Linköping Studies in Science and Technology
Licentiate Thesis No. 1937

A Composable and Extensible Environment for Equation-based Modeling and Simulation of Variable Structured Systems in Modelica

John Tinnerholm

LINKÖPING UNIVERSITY

Linköping Studies in Science and Technology
Licentiate Thesis No. 1937

A Composable and Extensible Environment for Equation-based Modeling and Simulation of Variable Structured Systems in Modelica

John Tinnerholm

Linköping University
Department of Computer and Information Science
Division of Software and Systems
SE-581 83 Linköping, Sweden

Linköping 2022

This is a Swedish Licentiate's Thesis

Swedish postgraduate education leads to a doctor's degree and/or a licentiate's degree.
A doctor's degree comprises 240 ECTS credits (4 years of full-time studies).
A licentiate's degree comprises 120 ECTS credits.

Edition 1:1

ISBN 978-91-7929-367-3 (Printed)
ISBN 978-91-7929-368-0 (Electronic)
ISSN 0280-7971
DOI: https://doi.org/10.3384/9789179293680

Typeset using XeTeX

Printed by LiU-Tryck, Linköping 2022

I dedicate this thesis to my family.

ABSTRACT

Modeling and Simulation are usually used to solve real-world problems safely and efficiently by constructing digital models of Cyber-Physical Systems. The models can be simulated and analyzed with respect to requirements, and decisions about their design can be based on this analysis. In the latest years, the field of Modeling and Simulation has grown massively and is tackling systems with increased complexity. Thus, the process of modeling and simulating Cyber-Physical systems is becoming more and more complex. This increase requires modeling languages that can express systems with increasing complexity.

Modelica is an open-standard declarative equation-based object-oriented language used to model various systems expressed using equations. Modelica tools can read the models, process them, and simulate them. However, the Modelica language and tools cannot express some concepts such as structural changes to the components or behavior of Cyber-Physical Systems during Simulation.

In this thesis, we propose extensions of the Modelica language to support modeling so-called variable structure systems, that is, systems where the structure of the system varies during Simulation. The full Modelica language and the new extensions are supported by a novel composable programming environment framework called OpenModelica.jl written in the Julia language. The proposed Modelica language extensions can handle explicit and implicit modeling of variable structure systems by introducing new operators and, consequently, new semantics to the Modelica language.

The explicit modeling is based on extensions that switch at runtime between continuous modes of operations with operators similar to the ones used in the specification of Modelica state-machines. The implicit modeling supports reconfiguration during runtime via recompilation. A Just-in-time compiler was implemented to handle the new semantics using the symbolic-numeric programming language Julia.

We investigate the performance of our new framework and compare it with existing state-of-the-art Modelica tools on models with thousands of equations and variables. The results show that our extensions and proposed runtime framework is viable for simulating both usual Modelica models and models with variable structure systems.

The conclusion is that the Modelica language can be extended further to support systems with variable structures with the addition of a few operators and JIT enhanced runtime system support. Based on the result of this thesis, we propose several directions for future work.

This work has been supported by the Swedish Government in the ELLIIT project and by Vinnova in the ITEA3 EMBRACE project. Support has also been received from the Swedish Strategic Research foundation (SSF) in the LargeDyn project. The development of OpenModelica is supported by the Open Source Modelica Consortium.

POPULÄRVETENSKAPLIG SAMMANFATTNING

Ekvationer har länge använts för att underlätta för mänskligheten inom flera olika fält. När analysen utvecklades under 1700-talet av Leibnitz och Newton började differentialekvationer användas i större utsträckning för att beskriva olika fysikaliska fenomen. När därefter först den analoga och sedan den digitala datorn uppfanns kunde ekvationer skrivas och simuleras med hjälp av datorer. Detta utgjorde grunden för ett ämne som sedan kom att kallas beräkningsvetenskap där beräkningsmetoder studeras för att med större noggrannhet och effektivitet kunna simulera fysikaliska system.

Kompilatorkonstruktion är ett annat ämne. I kompilatorkonstruktion undersöker forskare hur en textuell eller visuell programbeskrivning kan översättas till effektiv maskinkod för att köras på en digital dator. Denna licentiatavhandling undersöker hur ett ramverk för att konstruera kompilatorer kan realiseras för att kunna representera så kallade system med variabel struktur, som inte tidigare har kunnat representeras i det ekvationsorienterade programspråket Modelica. Exempel på system med variabel struktur är en pendel som går av eller en bil där ett hjul faller av under körning. Det vill säga situationer som förekommer i verkligheten men som inte med enkelhet går att modellera statiskt.

Licentiatavhandlingen visar att det möjligt av att kombinera simulering under kompilering på ett sådant sätt att enbart ett fåtal nya operatorer behövs för att kunna simulera mer dynamiska system i Modelica.

Detta har praktiska tillämpningar som exempelvis simulering av elnät där olika typer av energikällor såsom vindkraft och vattenkraft samverkar. Sådana system har tidigare gått att simulera i Modelica, men har krävt att att simuleringen konfigureras explicit för olika scenarier; med de utökningar som avhandlingen presenterar kan beteenden såsom att ett vindkraftverk slutar fungera representeras på ett mer precist sätt än tidigare. Vidare så har en helt ny modellerings och simuleringsmiljö utvecklas.

Författarens tack

Först och främst skulle jag vilja tacka min huvudhandledare Adrian Pop. Adrian har alltid varit tillgänglig, och vi har haft flera intressanta prestigelösa diskussioner där jag har kunnat diskutera allt mellan himmel och jord. Jag skulle också vilja tacka min bihandledare Martin Sjölund. Martin introducerade mig till kompilatorkonstruktion och till ekvationsbaserad modellering från ett datalogiskt perspektiv. Vidare skulle jag också vilja utbringa ett stort tack till min andra bihandledare Peter Fritzson. Peters långa erfarenhet och kunnande har varit ytterst hjälpsamt under forskarutbildningen samt i mitt akademiska skrivande. Vidare skulle jag skulle också vilja tacka mina kollegor vid PELAB. Tack till Mahder Gebremedhin och Lennart Ochel för intressanta lunchdiskussioner. Jag vill också uttrycka tacksamhet till Alachew Mengist för hans återkoppling på de sista utkasten av avhandlingen och den alltid tillgängliga och kompetente labchefen Kristian Sandahl.

Mina kollegor Lena Buffoni, Christoph Kessler och Ola Leifler har också varit viktiga. Att handleda masterarbeten tillsammans med er har varit väldigt givande vilket har hjälpt mig i mitt eget skrivande.

Jag skulle också vilja tacka Francesco Casella, Andreas Heuermann, Karim Abdelhak och Bernhard Bachman i OpenModelica-konsortiet för deras hjälp. Anne Moe förtjänar också att omnämnas, tack för att du alltid har tid och tack för allt du gör för att hjälpa och vägleda doktorander på IDA. Samma tack förtjänar också administratören på SaS, Lene Rosell, utan hennes hjälp hade det varit väldigt svårt att resa.

Sist men inte minst vill jag tacka min familj, speciellt min mor Angela och min far Anders, mina vänner och resten av min familj. Tack för att ni finns och har funnits!

Contents

List of Figures

List of Tables

List of Symbols

$\dot{x}$	Derivative of x with respect to time.
t	Time.
$\underset{\rightarrow}{x}$	Vector of state variables.
$\underset{\rightarrow}{\dot{x}}$	Vector of state derivatives.
$\underset{\rightarrow}{y}$	Vector of algebraic variables.
$\underset{\rightarrow}{u}$	Vector of input variables.
$\underset{\rightarrow}{p}$	Vector of parameters and constants.
$\hat{\mu}$	Sample mean.
$\hat{x}$	Sample median.
$\hat{\sigma}$	Sample standard deviation.
$\underset{\rightarrow}{0} = f(t, \underset{\rightarrow}{\dot{x}}(t), \underset{\rightarrow}{x}(t), \underset{\rightarrow}{y}(t), \underset{\rightarrow}{u}(t), \underset{\rightarrow}{p})$	System of differential algebraic equations.

List of Abbreviations

AOT-Compilation	Ahead-Of-Time Compilation. 26, 27, 47, 82
AOT-Compilers	Ahead-of-time Compilers. 26, 27
BDF	Backward differentiation formulas. 17
DAE	Differential Algebraic Equation 13, 15, 17
DASSL	Differential Algebraic System Solver 20
DFA	Deterministic Finite Automaton 9
DFSM	Deterministic Finite State Machine xvii, 9, 10
EOOL	Equation-based object-oriented modeling languages. 50
HiR	High-level intermediate representation. Denotes an intermediate language utillized between initial compiler translation phases. The High-level intermediate representation keeps control structures and other characteristics from the original language. 24
Interpretation	Interpretation, in this context is to be understood as a program, A interpreting the meaning of another program, B. That is program that exectutes a program described in another program. 47, 82
IR	Intermediate representation. Denotes an intermediate language used in compilers between translation phases. 43
JIT-Compilation	Just-In-Time Compilation. 26, 27, 41, 47, 51, 62, 71, 75, 78, 81–83, 86, 89
JIT-Compilers	Just-in-time Compilers. 26, 27
LiR	Low-level intermediate representation. Denotes an intermediate language used in the last phases of a typical compiler before code generation. The LiR representation is close to the target language but otherwise language independent. 26

LLVM-IR	The intermediate representation of the LLVM compilation framework. 26
MCL	Model Composition Language 43
MiR	Mid-level intermediate representation. Denotes an intermediate language used in compilers between translation phases that typically keep some structures from the original language. 24, 26
MKL	Model Kernel Language 43
MSL	Modelica Standard Library 57, 75
MTK	ModelingToolKit 53, 73, 81
ODE	Ordinary Differential Equation 16, 17, 19, 29
OMC	The OpenModelica Compiler. xvii, 34, 53, 55, 56, 73–75, 77, 86, 106, 107
RLC	Resistance (R), Inductance (L), and Capacitance (C). Used in the context RLC circuit 13, 15
UML	Unified Modeling Language 37
VSS	Variable Structure Systems xiii, 1–4, 40, 42, 46, 47, 58, 66, 85, 86, 89, 90

1. Introduction

The models scientists have proposed to describe our reality are continuously improving. With the dawn of computing, the necessity of constructing physical models such as the model that disproved the Reeber plan[1], has been reduced. Instead, models can be created and simulated using computers. Hence, the advent of the computer has proven to be a useful tool for both academia and industry.

Still, these developments would not be possible without the environments and tools to support them, such as MATLAB, Dymola, OpenModelica, Wolfram, et cetera. While some environments and associated tooling focus on specific domains, others aim for generality. In the past, models such as the model of the Reber Plan were physical; that is, they did not contain any digital sub-components. However, modern systems such as personal cars also contain digital subsystems. Such systems are called Cyber-Physical Systems (CPS). One example of a CPS is the modern car consisting of mechanical (physical) components and software components. Hence, modern CPS, such as a car, is often complex, exhibiting both discrete and continuous behavior. Furthermore, they are usually systems of systems.

The equation-based language Modelica is one way of modeling such systems. Modelica was developed as a unified object-oriented equation-based language. The development of Modelica is a continuous effort to create and maintain an open standard to model wide spectra of systems. However, there are some elements of modern systems that are hard to capture efficiently in Modelica. More specifically, Modelica lacks standardized support for highly dynamic systems, so-called Variable Structured Systems (VSS). While classes of VSS can be handled in Modelica[2], support for systems in which the index [3] of the differential-algebraic equations (DAE) varies at simulation time is not supported.

[1] John Reber conceived the Reber plan in 1949 to terraform the San Francisco bay into two freshwater lakes. A physical miniature of San Francisco Bay was constructed to simulate the effect of this change. Luckily, the simulation demonstrated that the implementation of the plan would lead to an environmental disaster, so the plan was abandoned (Weisberg 2012).

[2] For instance, compile-time structural components.

[3] The DAE index is the number of times a DAE must be differentiated to be transformed into an ODE, the index of the ODE being zero (Brenan, Campbell, and Linda Ruth Petzold 1995).

1.1 Motivation

While several modeling languages and environments support acausal modeling, support for variable structure modeling has yet to enter the mainstream. Existing proposals (Benveniste, Benoît Caillaud, and Malandain 2021; Benveniste, Benoît Caillaud, Elmqvist, Ghorbal, Otter, and Pouzet 2019; Höger 2019; Zimmer 2010; Giorgidze and Nilsson 2009) to handle variable structure system modeling and simulation are not yet integrated into any mainstream tool. Furthermore, there are no detailed studies concerning the practical implications of a modeling and simulation environment with VSS support.

This calls for an investigation on how to integrate support for VSS in a framework that is capable of supporting larger systems than previous approaches while adhering to existing standards and practices.

1.2 Aim

This thesis aims to provide a new environment for equation-based languages in general and Modelica in particular. This environment is then used to examine the practical implication of dynamic compilation within the context of equation-based languages empirically.

1.3 Research questions

The main question in this licentiate thesis is: How to integrate VSS support in an equation-based language to allow for efficient simulation of large systems?

While the question above constitutes the main theme, it can be divided into the following sub questions.

1. (RQ-I) What syntactic and semantic constructs are needed in an equation-based language for modeling and simulating VSS?
2. (RQ-II) What characteristics of a modeling and simulation framework are appropriate for achieving VSS support?
3. (RQ-III) How can VSS support for Modelica be realized to simulate large systems effectively?

1.4 Research Methodology

This thesis follows the principal methodology of a design study proposed by Peffers, Tuunanen, Rothenberger, and Chatterjee (2007) sketched in Figure 1.1.

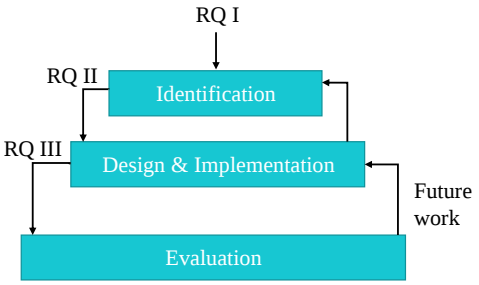

Figure 1.1: An overview of the research methodology of this thesis.

Starting with Research Question I (RQ I) relevant literature was examined and together with the initial experiments described in Tinnerholm, Sjölund, and Pop (2019) a framework was designed to support the syntactic and semantic constructs of VSS in Modelica.

During the process of answering *RQ II* Just-In-Time-Compilation was selected as the main technique to realize the simulation runtime.

To answer *RQ III* we investigated the feasibility of VSS simulation for large systems, and the subsequent impact on compiler design in a performance experiment where we gradually increased the number of equations and variables for a selection of models. As depicted in Figure 1.1 these experiments also influenced the design.

If we compare the methodology to that of the suggested guidelines for design studies by (Peffers, Tuunanen, Rothenberger, and Chatterjee 2007) and our research questions the first two steps contributed to answer *RQ I*:

- Problem identification and motivation.
- Define the objectives for a solution.

The process of answering *RQ II* corresponds to the *Design and Development* stage. Finally, the evaluation stage corresponds to the *Demonstration* and *Evaluation* stage.

While we follow the steps sequentially, some stages have been refined iteratively during the process of this research[4].

1.5 Contributions

The contributions of this licentiate thesis are to be understood in the context of equation-based languages. This thesis contributes to our understanding of designing a composable framework to model and simulate large systems

[4]Examples of this is the refinement of algorithms when inefficiencies have been noted during our experiments.

with variable structure using Modelica. Furthermore, this thesis provides a concrete framework to do so. Moreover, while an extension of Modelica is the target of this thesis, this framework provides the necessary retargetability so that it could be used to support new novel equation-based languages.

To summarize, the main contributions are:

- Firstly, an overview of current and past research within equation modeling of variable structured systems.
- Secondly, empirical insight regarding large-scale VSS simulation, and an initial proposal on how to design support for this paradigm in standard Modelica.
- Thirdly, a composable compiler framework that can be used to increase collaboration and further the design of other equation-based languages in the context of modeling and simulation.

1.6 List of publications

This monograph is based on the following publications.

1. Towards introducing just-in-time compilation in a Modelica compiler. In Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools (pp. 11-19) (Tinnerholm, Sjölund, and Pop 2019).
2. Towards an Open-Source Modelica Compiler in Julia. In Proceedings of Asian Modelica Conference (pp. 08-09) (Tinnerholm, Pop, Sjölund, Heuermann, and Abdelhak 2020).
3. OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl. In Modelica Conferences (pp. 109-117) (Tinnerholm, Pop, Heuermann, and Sjölund 2021).
4. A modular, extensible, and Modelica standard-compliant OpenModelica compiler framework in Julia supporting structural variability, J Tinnerholm, A Pop, M Sjölund (Submitted for publication)

1.6.1 Other publications not included in this thesis

Publications listed here are publications completed during my time writing this licentiate thesis that concern empirical software engineering and modeling and simulation but are not included:

1. A Failed attempt at creating Guidelines for Visual GUI Testing: An industrial case study. In 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST) (pp. 340-350)

2. The OpenModelica Integrated Environment for Modeling, Simulation, and Model-Based Development. Modeling, Identification and Control, P Fritzson et al.

1.7 Delimitations

This thesis aims not to prove the correctness of the implementation formally. Rather the aim is to provide insight concerning:

- How well the Julia language performs as an implementation language for a large application.
- The practical performance implications concerning the simulation of large dynamic systems.
- How support for models with dynamic structure can be realized.

This licentiate thesis is inductive and experimental; deductive approaches such as formal methods are not considered. Furthermore, while separate compilation of equation-based languages is a central concern in the context of efficient dynamic compilation, due to time limitations, such approaches are not considered. Notwithstanding these limitations, this thesis suggests that support for models with varying structures can be implemented in the core Modelica language with just a few modifications.

1.8 Structure

The structure of the remaining chapters is as follows: in Chapter 2 the concepts of systems, simulation, and some foundations within mathematical modeling are presented. In Chapter 3 we present the background concerning compilers and equation-based languages. Chapter 4 presents the framework developed as part of the thesis and the experimental framework used to answer the stated research questions. The result of the experiments evaluating the performance are presented in Chapter 5, and the answers to the research question are provided in Chapter 6. Finally, future research directions are presented in Chapter 7.

2. Systems and Simulation

In this chapter, we present the background of this thesis. In Section 2.1 we discuss continuous, discrete, and hybrid systems and how they can be represented using equations. This is followed by Section 2.2 where we discuss algorithms used to simulate equation-based systems. The examples in this chapter were simulated using *OpenModelica.jl* developed as a part of the thesis.

2.1 Systems and Equation-based modeling

What is a system? Donella Meadows gives one definition in Thinking in Systems (Meadows 2008):

> *"A set of elements or parts that is coherently organized and interconnected in a pattern or structure that produces a characteristic set of behaviors, often classified as its "function" or "purpose"." Meadows 2008. p. 188.*

This is similar to the definition given by Merriam Webster[1]:

> *"A regularly interacting or interdependent group of items forming a unified whole." Merriam-Webster 2020.*

This thesis is concerned with systems modeled by equations. We briefly describe Continuous Systems, Discrete Systems, and Hybrid Systems in the following subsections.

2.1.1 Continuous Systems

In continuous systems no events occur. When modeling continuous systems using equations it means that the values of the system variables at any time point t are decided by continuous functions. An example of a physical scenario represented with continuous equations can be seen in Figure 2.1. This can be described by a continuous system of equations as given by Equation (2.1). The system describes how the height, h and the velocity, v of a ball dropped from this tower behave where $h_0 = 1$ and $v_0 = 0$ when $t = 0, t \geq 0$.

[1] Merriam Webster is a well known American dictionary.

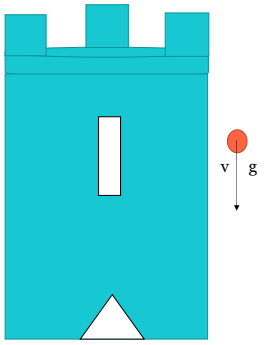

Figure 2.1: A ball being dropped from a high tower.

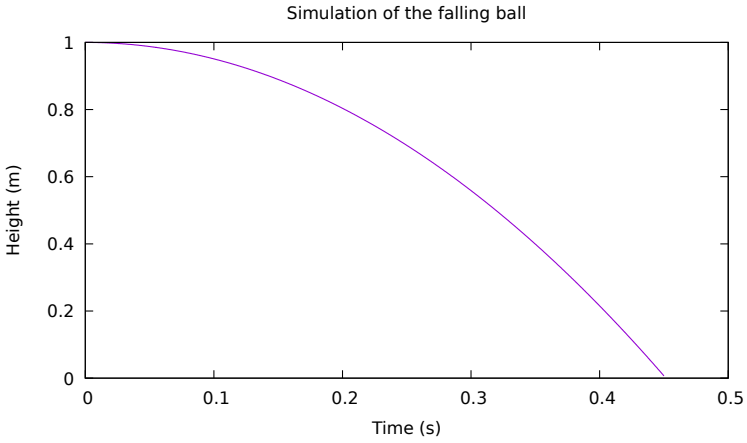

Figure 2.2: Plot of the h height of the ball as a function of time.

$$f(t) = \begin{cases} g = 9.81 \\ \dot{h} = v \\ \dot{v} = -g \end{cases} \tag{2.1}$$

In Figure 2.2 we can observe how this system behaves if it is simulated for 0.5 seconds. Equation (2.1) does not take the ground into account as that would imply using discrete behavior in our continuous model. Systems where continuous and discrete behavior is combined are discussed in Section 2.1.3.

2.1.2 Discrete Systems

While the continuous systems depicted in Figure 2.1 can be modeled using continuous functions by contrast, in a discrete system, the variables are controlled by discrete functions; hence the state of the system change at discrete points in time. One example of a discrete system is a digital computer[2] another is a deterministic finite state machine (DFSM)[3].

A DFSM can be represented with an alphabet Σ; a set of states Q; a transition function δ and a set of final states F (Hopcroft, Motwani, and Jeffrey D. Ullman 2007).

A DFSM that accepts the word *DISCRET* can be defined as follows:

$$\begin{aligned} \Sigma &= \{D, I, S, C, R, E, T\}, \\ Q &= \{S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}\}, \\ F &= \{S_9\} \end{aligned} \tag{2.2}$$

The state transition function, δ is represented using a table 2.1. Hence, as the table illustrates $\delta('D', S_1) = S_2, \delta('I', S_2) = S_3, \delta('S', S_3) = S_4$ and so on.

	D	I	S	C	R	E	T
$\rightarrow S_1$	S_2	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}
S_2	S_{10}	S_3	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}
S_3	S_{10}	S_{10}	S_4	S_{10}	S_{10}	S_{10}	S_{10}
S_4	S_{10}	S_{10}	S_{10}	S_5	S_{10}	S_{10}	S_{10}
S_5	S_{10}	S_{10}	S_{10}	S_{10}	S_6	S_{10}	S_{10}
S_6	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_7	S_{10}
S_7	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_8
S_8	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_9	S_{10}
$*S_9$	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}
S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}	S_{10}

Table 2.1: State transition table of a DFSM that accepts the string *DISCRETE*. S_1 is the starting state and S_9 is the accepting state.

A DFSM can be visualized as a directed graph with the set of states as vertices and the set of transitions represented as edges, see Figure 2.3.

The next section discusses hybrid systems that combine discrete and continuous behavior.

[2] These discrete systems can be used to simulate continuous systems.
[3] Also known as a Deterministic Finite Automaton (DFA).

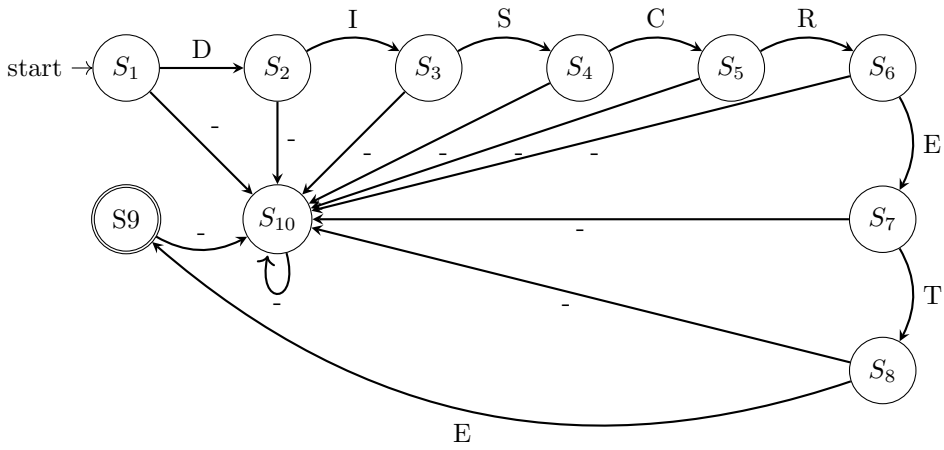

Figure 2.3: The state transition diagram for the DFSM defined in Equation 2.2 and Table 2.1. In this figure – denotes input that lead to our error state (S_{10}). So for state S_1 – represents some $x \in \{I, S, C, R, E, T\}$.

2.1.3 Hybrid Systems

Our previous model describes the change in height and velocity of a ball dropped from a tower, assuming that the ball would never hit the ground. If we consider the ground in our model, we need to introduce events. We introduce one *event* that happens when the ball hits the ground and $h = 0$. To model this behavior, we need to introduce the notion of state. Either the ball is in free fall as in Equation (2.1) or the ball made contact with the ground at some time t_n and changed its trajectory before falling again, see Figure 2.4.

The event that occurs when the ball hits the ground is a discontinuity. Therefore, the value of v changes at that instance, in Equation (2.3) e denotes the coefficient of restitution, and v^- represents the value of v before the event and v^+ the value after the event. During the event, the value of v changes abruptly to $-e * v^-$. This value is the new initial value of v after the event until some time $t + \delta$ when the ball once again hits the ground[4].

$$\begin{cases} e = 0.7, \\ v^+ = -e * v^- \end{cases} \tag{2.3}$$

Figure 2.5 illustrates the behavior of the system with the introduction of discrete events. By introducing the notion that the ball hits the ground and consequently bounce, we have introduced a discrete behavior in our previously continuous model. This is a hybrid system since it exhibits both discrete and continuous behavior (Cellier and Kofman 2006).

[4]In fact, this will continue indefinitely.

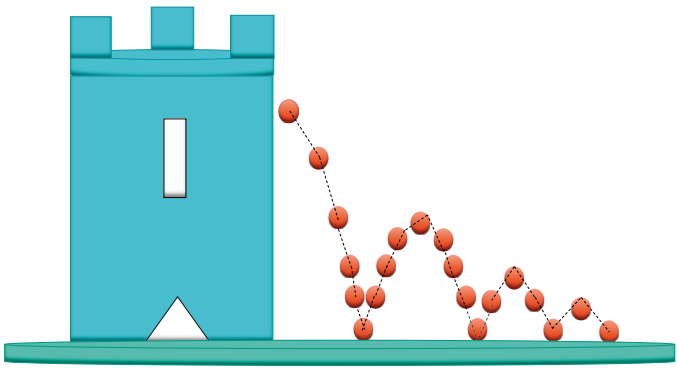

Figure 2.4: A ball being dropped from a high tower. On impact the ball bounces.

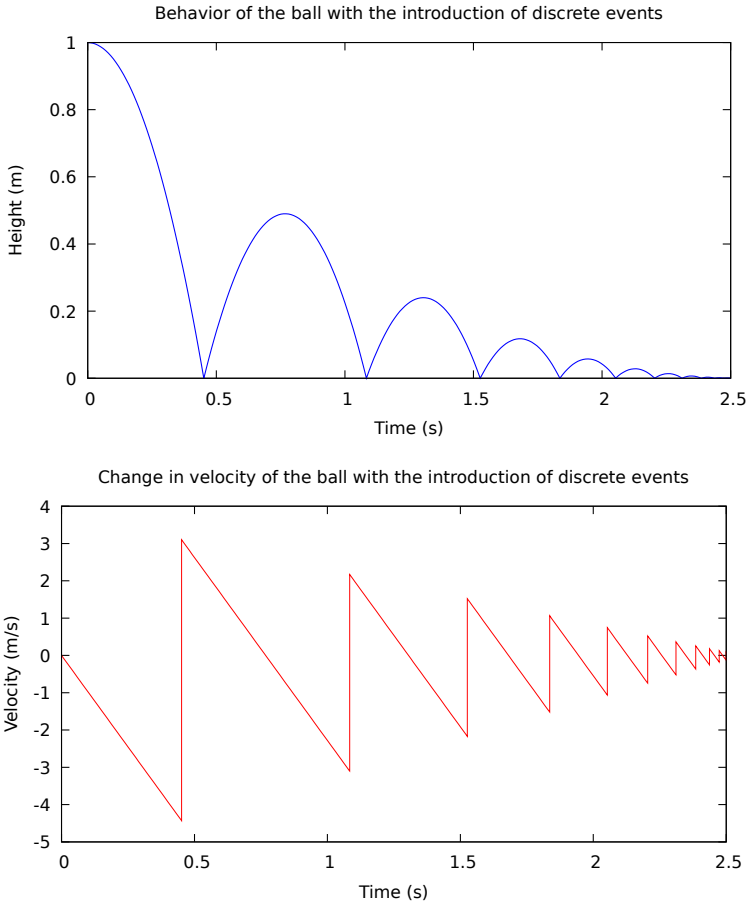

Figure 2.5: Plot of $h(t)$ and $v(t)$ respectively when extending the falling ball in Equation 2.1 with the discrete event in Equation 2.3.

2.1.4 Differential Algebraic Equations

In the previous sections, we discussed three types of systems that appear when modeling with equations. In this section, we discuss systems of differential-algebraic equations (DAE), a useful mathematical model when modeling continuous systems (Cellier and Kofman 2006). A DAE system can be formulated generally as:

$$\underset{\rightarrow}{0} = f(t, \underset{\rightarrow}{\dot{x}}(t), \underset{\rightarrow}{x}(t), \underset{\rightarrow}{y}(t), \underset{\rightarrow}{u}(t), \underset{\rightarrow}{p}) \tag{2.4}$$

In Equation (2.4) the arguments to the function f are divided into five different categories. All variables that are derived w.r.t the state of the system are named *state variables*, denoted $\underset{\rightarrow}{x}$ and the corresponding *state derivatives* are denoted $\underset{\rightarrow}{\dot{x}}$. Variables that are not differentiated w.r.t the state $\underset{\rightarrow}{y}$ are called *algebraic variables*.

A system might depend on parameters, constants and external input. For instance, a model of a Resistance (R), Inductance (L), and Capacitance (C), in short a RLC (RLC) circuit, can be reused if, for instance, the capacitance of the circuit is parameterized[5]. To reflect this, Equation (2.4) includes an input vector $\underset{\rightarrow}{u}$ and a parameter vector $\underset{\rightarrow}{p}$.

[5]The ability to reuse physical components defined using equations is one of the advantages with equation-based programming languages which we discuss in this thesis.

$$
\begin{aligned}
R2.p.v &= AC.p.v \\
R2.p.v &= R1.p.v \\
C.p.v &= R1.n.v \\
R2.n.v &= L.p.v \\
AC.n.v &= C.n.v \\
AC.n.v &= L.n.v \\
AC.n.v &= G.p.v \\
R1.n.i + C.p.i &= 0.0 \\
L.p.i + R2.n.i &= 0.0 \\
G.p.i + C.n.i + L.n.i + AC.n.i &= 0.0 \\
R1.p.i + R2.p.i + AC.p.i &= 0.0 \\
R1.R * R1.i &= R1.v \\
R1.v &= R1.p.v - R1.n.v \\
0.0 &= R1.p.i + R1.n.i \\
R1.i &= R1.p.i \\
C.i &= C.C * \frac{dC.v}{dt} \\
C.v &= C.p.v - C.n.v \\
0.0 &= C.p.i + C.n.i \\
C.i &= C.p.i \\
R2.R * R2.i &= R2.v \\
R2.v &= R2.p.v - R2.n.v \\
0.0 &= R2.p.i + R2.n.i \\
R2.i &= R2.p.i \\
L.L * \frac{dL.i}{dt} &= L.v \\
L.v &= L.p.v - L.n.v \\
0.0 &= L.p.i + L.n.i \\
L.i &= L.p.i \\
AC.v &= AC.A * sin(AC.w * t) \\
AC.v &= AC.p.v - AC.n.v \\
0.0 &= AC.p.i + AC.n.i \\
AC.i &= AC.p.i \\
G.p.v &= 0.0
\end{aligned}
\tag{2.5}
$$

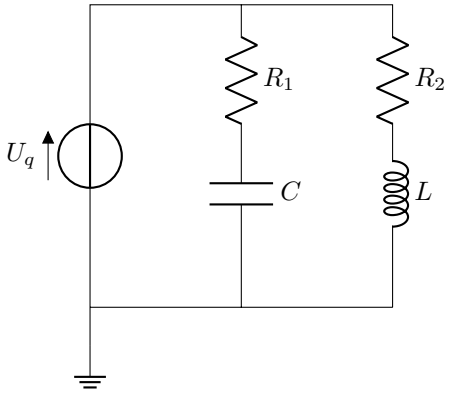

Figure 2.6: A RLC circuit connected to a sine voltage source. The circuit is based upon the simple circuit model in (P. Fritzson 2014, p. 43).

$$
\begin{aligned}
\dot{\underrightarrow{x}} &= \{\frac{dC.v}{dt}, \frac{dL.i}{dt}\},\\
\underrightarrow{x} &= \{C.v, L.i\},\\
\underrightarrow{y} &= \{AC.i, AC.n.v, AC.p.v, AC.v, C.i, C.n.v, C.p.v,\\
&\quad G.p.v, L.n.v, L.p.v, L.v, R1.i, R1.n.i, R1.n.v,\\
&\quad R1.p.i, R1.p.v, R1.v, R2.i, R2.n.v, R2.p.i, R2.p.v,\\
&\quad R2.v, AC.n.i, AC.p.i, C.n.i, C.p.i\\
&\quad G.p.i, L.n.i, L.p.i, R2.n.i\},\\
\underrightarrow{u} &= \{\},\\
\underrightarrow{p} &= \{R1.R, C.C, R2.R, L.L, AC.A, AC.w\},
\end{aligned}
\tag{2.6}
$$

In Equation (2.5) we have a set of equations representing the RLC circuit in Figure 2.6. This set of equations constitutes a DAE system, which is a system with mixed differential and algebraic equations. In this example, the set of state variables, state derivatives, algebraic variables, parameters, constants, and input variables, are assigned as in Equation (2.6). In the subsequent sections, we discuss how to numerically solve such systems.

2.2 Continuous System Simulation

To simulate DAE systems, we need to numerically solve systems such as Equation (2.4). However, we start by discussing how to solve systems of Ordinary

Differential Equations (ODEs), which are systems of the following form:

$$\begin{aligned} \dot{x}(t) &= f(t, x(t)) \\ x(t_0) &= x_0 \end{aligned} \tag{2.7}$$

Several methods for numerical integration have been developed to achieve an approximate solution. These methods can roughly be divided into three categories, Explicit methods, Implicit methods, and so-called Higher-Order methods (Cellier and Kofman 2006).

2.2.1 Explicit methods

The simplest explicit method is the Euler method[6]. Explicit methods consider the current state of the system, and from this current state, we calculate the value of the system at some time $t + \Delta t$

By using the definition of the derivative, the Euler method numerically approximate the true solution via extrapolation:

$$x(t_n + \Delta t) \approx x(t_n) + \Delta t \cdot f(t_n, x(t_n)) \tag{2.8}$$

The approximate equality in Equation (2.8) is used to formulate Algorithm 1.

Algorithm 1 The explicit Euler method for one timestep, $t + \Delta t$. The parameter t represents the current value of the time, Δt is the current timestep, H is the function to integrate and V is the set of state variables.

function Explicit Euler(t, Δt, H, V)
  $k_1 \leftarrow H(t, V)$
  $h_k \leftarrow \Delta t \cdot k_1$
  $V \leftarrow V + h_k$
  return V

This explicit integration algorithm assumes a fixed step-size; however, it might be difficult to know what step-size to select for a particular problem. To solve this issue, algorithms with adaptive step-sizes have been developed. One such being *Richardson Extrapolation* (Cheney and Kincaid 2003).

For a more in depth treatment of other explicit methods such as Runge-Kutta and Heun, see (Cheney and Kincaid 2003).

2.2.2 Implicit methods

While explicit methods and step-size control improve the accuracy, such improvements are sometimes not enough for certain systems. Furthermore, for

[6]This method is named after the mathematician Leonard Euler.

some systems using adaptive step-size in combination with a higher-order explicit method might result in small timesteps. Hence, implicit methods allow for greater step-size and stability (Cheney and Kincaid 2003).

An example of an implicit method is the implicit Euler method or backward Euler method, Equation (2.9):

$$x(t_n + \Delta t) = x(t_n) + \Delta t \cdot x(t_n + \Delta t) \tag{2.9}$$

In Equation (2.9) an algebraic equation needs to be solved for each timestep Δt, since $x(t_n + \Delta t)$ occurs on both sides of the equation. Since the algebraic equation might be nonlinear, integration using implicit methods is more computationally expensive in comparison to explicit methods such as the Euler method. Similar to the explicit methods, several implicit methods exist. A comprehensive overview of implicit and explicit methods and their applications in solving stiff and nonstiff ODE systems are available in (Norsett and Wanner 1993; Gerhard Wanner and Hairer 1996).

2.2.3 Multistep methods

Another category of integration algorithms are the multistep methods. While explicit and implicit methods disregard previously computed values, multistep methods also consider previously computed solutions. These methods are typically divided into three categories Adams-Bashforth methods, Adams-Moulton methods, and Backward differentiation formulas BDF. Adams-Bashforth methods are explicit; BDF and Adams-Moulton methods are implicit (Cellier and Kofman 2006).

2.2.4 Representing systems

In the previous sections, we presented a set of integration methods that can be applied to solve ODE systems numerically. However, the DAE system presented in Figure 2.6 and the resulting equations in Equation (2.5) is not an ODE. It is possible to transform a DAE system into a corresponding ODE system. This section briefly discusses the steps needed to go from one representation to the other. Recall Equation (2.5) and Equation (2.6). In this DAE system we got 32 equations and 32 variables.

We can start by simplifying the set of equations that constitutes this DAE system, removing trivial equations such as $R2.p.v = AC.p.v$. The simplification results in Equation (2.10).

$$
\begin{aligned}
G.p.i &= L.i - ((-C.i) - AC.i) \\
AC.i &= (-L.i) - C.i \\
C.i &= R1.v / R1.R \\
R1.v &= AC.v - C.v \\
\frac{C.v}{dt} &= C.i / C.C \\
R2.v &= R2.R * L.i \\
L.v &= AC.v - R2.v \\
\frac{L.i}{dt} &= L.v / L.L \\
AC.v &= AC.A * x \\
x &= sin(AC.w * t)
\end{aligned}
\tag{2.10}
$$

As in Equation (2.6) we divide our system into five sets:

$$
\begin{aligned}
\dot{\underrightarrow{x}} &= \{\frac{dC.v}{dt}, \frac{dL.i}{dt}\}, \\
\underrightarrow{x} &= \{C.v, L.i\}, \\
\underrightarrow{y} &= \{G.p.i, AC.i, AC.v, L.v, R2.v, C.i, R1.v\}, \\
\underrightarrow{u} &= \{x\}, \\
\underrightarrow{p} &= \{R1.R, C.C, R2.R, L.L, AC.A, AC.w\}
\end{aligned}
\tag{2.11}
$$

Recall from Equation (2.6) that $\dot{\underrightarrow{x}}$ are our state derivatives, $\underrightarrow{x}$ are our state variables, $\underrightarrow{y}$ are our algebraic variables and $\underrightarrow{p}$ are our parameters and constants. To solve this system using one of the integration algorithms within a procedural programming language, we transform our system of equations into the *explicit state space* representation see Equation (2.12) (Cellier and Kofman 2006).

$$
\begin{aligned}
\dot{\underrightarrow{x}} &= h(t, \underrightarrow{x}; \underrightarrow{u}, \underrightarrow{p}) \\
\underrightarrow{y} &= k(t, \underrightarrow{x}; \underrightarrow{u}, \underrightarrow{p})
\end{aligned}
\tag{2.12}
$$

We also reformulate the equations for the state derivatives so that they only depend on the state, input variables and parameters. That is $\frac{C.v}{dt} = \frac{(AC.A \cdot x - C.v)/R1.R}{C.C}$ and $\frac{L.i}{dt} = \frac{AC.A \cdot x - R2.R \cdot L.i}{L.L}$

Using the explicit state space representation we get the following equations:

$$h(t, \underset{\rightarrow}{x}; \underset{\rightarrow}{u}, \underset{\rightarrow}{p}) = \begin{cases} \frac{C.v}{dt} = \frac{(AC.A \cdot x - C.v)/R1.R}{C.C} \\ \frac{L.i}{dt} = \frac{AC.A \cdot x - R2.R \cdot L.i}{L.L} \end{cases}$$

$$k(t, \underset{\rightarrow}{x}; \underset{\rightarrow}{u}, \underset{\rightarrow}{p}) = \begin{cases} G.p.i = L.i - ((-C.i) - AC.i) \\ AC.i = (-L.i) - C.i \\ C.i = R1.v/R1.R \\ R1.v = AC.v - C.v \\ R2.v = R2.R * L.i \\ L.v = AC.v - R2.v \\ AC.v = AC.A * x \\ x = sin(AC.w * t) \end{cases} \tag{2.13}$$

This representation enable us to reformulate the system such that it can be solved using a procedural algorithm. That is the system h is a ODE system that can be integrated in separation from the algebraic equations that constitute system k. Algorithm 2 describes the solution procedure. A concrete implementation is available in Listing A.1.3.

Algorithm 2 Procedural integration of the systems in Equation (2.13).

procedure SOLVE SYSTEM(t, Δt, h, k)
  Initialize parameters, states and algebraic variables
  while *Simulation not finished* **do**
    $\underset{\rightarrow}{x} \leftarrow$ *Explicit Euler*$(t, h, \underset{\rightarrow}{x}, \underset{\rightarrow}{u})$
    $\underset{\rightarrow}{y} \leftarrow k(t, \underset{\rightarrow}{x}, \underset{\rightarrow}{u})$
    $t \leftarrow t + \Delta t$

In Equation (2.13), we rewrote the system using forward substitution. However, not all systems of equations are possible to solve using forward substitution. One way to classify the difficulty of these systems is to use the DAE-index[7].

For example, the set of equations for the previously discussed DAE-System is a *index-0* system. The set of equations can be ordered[8] using forward substitution. However, sometimes the inter-dependencies between variables and equations can not be solved using forward substitution alone (Cellier and Kofman 2006).

[7] The meaning of the term DAE-index varies. An overview of the relationship to matching is available in *Determination of perturbation index of a DAE with maximum weighted matching algorithm* (Bujakiewicz and Bosch 1994).

[8] Causalized, that is, the equalities can be treated as assignments.

To visualize these inter-dependencies, let us consider the following nonlinear system of equations with initial values omitted:

$$
\begin{aligned}
N &= 5 \\
\underrightarrow{x} &= \{x_1, x_2, x_3, x_4, x_5\} \\
(1) N + 1 &= e^{(t*1+x_1)} + \Sigma \underrightarrow{x} \\
(2) N + 1 &= e^{(t*2+x_2)} + \Sigma \underrightarrow{x} \\
(3) N + 1 &= e^{(t*3+x_3)} + \Sigma \underrightarrow{x} \\
(4) N + 1 &= e^{(t*4+x_4)} + \Sigma \underrightarrow{x} \\
(5) N + 1 &= e^{(t*5+x_5)} + \Sigma \underrightarrow{x} \\
(6) \dot{y} &= \Sigma \underrightarrow{x} * t
\end{aligned}
\tag{2.14}
$$

In this example, we can observe that all algebraic variables in $\underrightarrow{x}$ are interdependent; only the state variable y is independent. We can visualize this relationship by using a directed graph, as in Figure 2.7. These interdependencies are called algebraic loops. Systems that contain algebraic loops are called *index-1* systems.

To handle algebraic loops, a method called tearing is employed. An algorithm for reducing high-index systems of DAE to lower index is the *Pantelides* algorithm (Pantelides 1988). In some cases, the algorithm for index reduction needs to be applied more than once. These systems are denoted higher index systems and have a DAE-index strictly greater than one, and the index of the system is reduced by one after each application of the algorithm of Pantelides (Cellier and Kofman 2006).

2.3 Summary

A numerical solver may implement several numerical algorithms (Hindmarsh, Brown, K. E. Grant, S. L. Lee, Serban, Shumaker, and Woodward 2005) or one specific configuration of an algorithm, which is the case, for example, in the DASSL(Differential-Algebraic System Solver) solver (Linda R Petzold 1982). Some frameworks abstract several solvers behind a single interface. A recent example is DifferentialEquations.jl (Christopher Rackauckas and Nie 2017).

Transforming systems of equations into executable code is a central concept of equation-based compilers and to a degree within the context of solver frameworks. The next chapter provides an overview of equation-based programming languages and how compilers automate the transformation process which is done in *OpenModelica.jl* for the Modelica language.

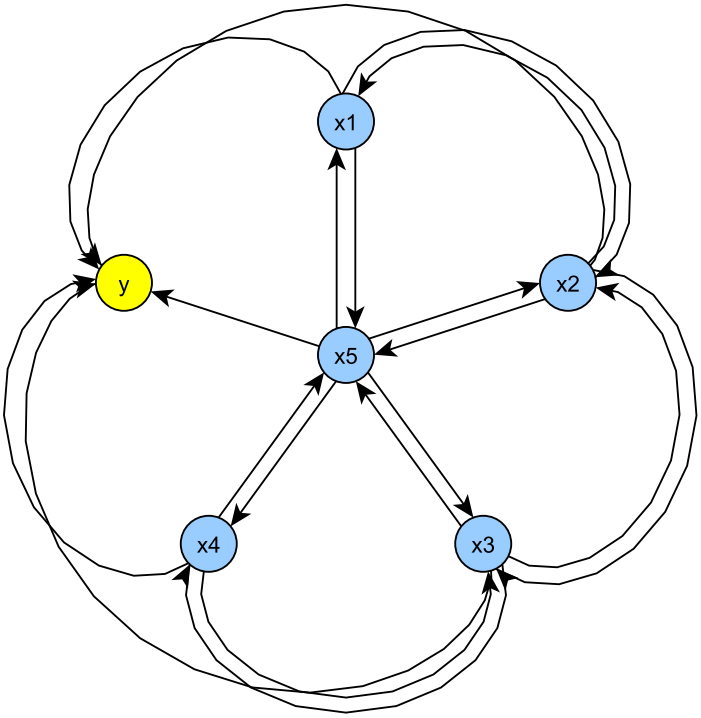

Figure 2.7: The relationship between the six variables and six equations in Equation 2.14. The edges represent equations and the vertices represent the variables. The independent variable y is colored yellow to distinguish it from $\underrightarrow{x}$. To solve for y the algebraic loop between the variables in $\underrightarrow{x}$ must be resolved.

3. Compilers and Equation-based modeling languages

In this chapter, we introduce compiler construction in Section 3.1 and in Section 3.2 we provide an overview of both past and present equation-based languages. Finally, we discuss equation-based languages and frameworks that supports variable structure in Section 3.3.

Section 3.3 is partly based upon:

- A modular, extensible, and Modelica standard compliant OpenModelica compiler framework in Julia supporting structural variability, J Tinnerholm, A Pop, M Sjölund (Submitted for publication)

3.1 Compilers and Interpreters - an overview

This section provides an overview of modern compiler design and associated compiler phases. We briefly elaborate on interpreters and their differences compared to compilers. Finally, in the last subsection, we discuss just-in-time compilation, its implications and possibilities.

3.1.1 Compilers

A *compiler* is a computer program designed to transform code[1] from one language to another[2] (Aho, Lam, Sethi, and Jeffrey D Ullman 2007).

The transformation from the source language (original language) into a target language is commonly subdivided into a sequence of compiler phases. Each phase has its purpose; a high-level overview of a typical compiler for a procedural language with associated phases is presented in Figure 3.1. The high-level overview of Figure 3.1 consists of three phases transforming the input program into a sequence of tokens and, from this sequence of tokens, constructing a syntax-tree and from the syntax-tree generating code for the target language. While the high-level overview of Figure 3.1 succinctly describes different components of a compiler, a modern optimizing compiler typically consists of more phases (Aho, Lam, Sethi, and Jeffrey D Ullman 2007).

[1] This program code may or may not be a high-level program.

[2] A common target language is a sequence of instructions that can be executed on some target architecture.

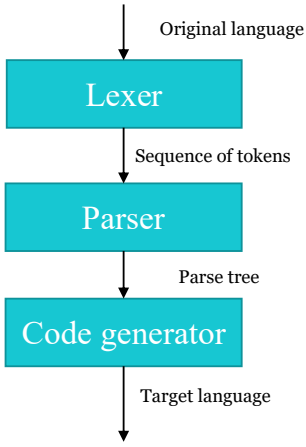

Figure 3.1: A high-level overview of compiler phases.

In Figure 3.2 a more detailed overview of an optimizing compiler is provided. The first part of the compiler is the frontend. For example, in Figure 3.2 the frontend consists of three modules, the Lexer, the Parser, and the Semantic Analyzer. The *Lexer* is responsible for transforming the textual program description into a set of tokens; this is called lexical analysis from which the name lexer is derived. This set of tokens produced by the Lexer is then fed into the Parser; the task of the Parser is to construct a *High-level intermediate representation* (HiR)[3] such that the sequence of tokens now represent a syntactically valid program w.r.t some grammar[4]. Certain optimizations can be performed on the intermediate representation of the frontend, the so-called, HiR one example being automatic parallelization (Muchnick 1997).

The midend is dedicated to platform-independent optimization; typically, the high-level intermediate representation is transformed into a midlevel intermediate representation (MiR). According to (Muchnick 1997) the MiR is designed to be a language-independent flat representation with the control structure defined as a basic block graph, still, as with the HiR, the exact characteristics of the MiR representation varies in the literature.

To give an example, some modern MiR such as Rust Intermediate Representation (MIR)[5] and the Swift Intermediate Representation (SIL)[6] are not fully language independent.

[3]The high-level intermediate representation typically keeps language-specific constructs such as loops.

[4]Depending on the compiler, this can be done during parsing or in a separate module. In Figure 3.2 this analysis is performed by the Semantic Analyzer.

[5]URL: https://rustc-dev-guide.rust-lang.org/mir/index.html, accessed 2022-05-03.

[6]URL: https://github.com/apple/swift/blob/main/docs/SIL.rst, accessed 2022-05-03.

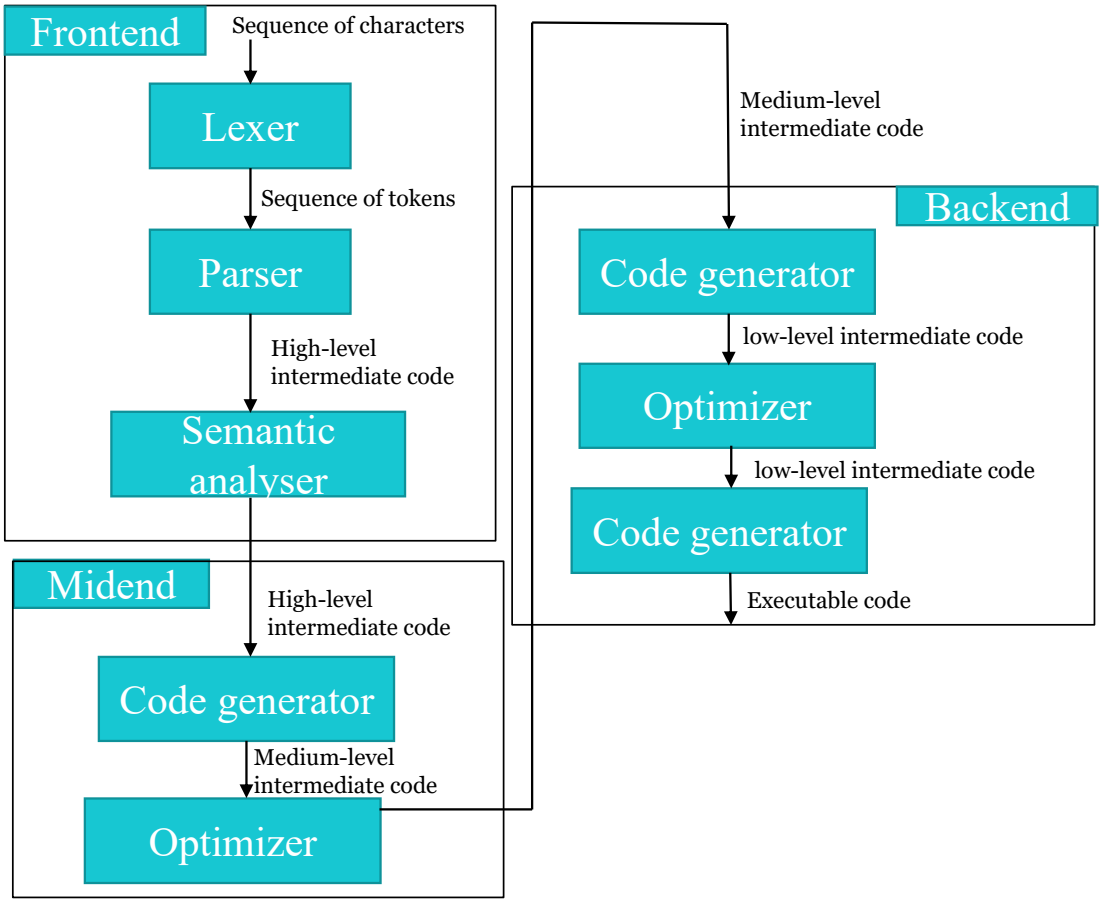

Figure 3.2: An overview of the different stages in an optimizing compiler, from the initial sequence of tokens from the target program to the low-level intermediate code and finally the target language, in this case machine code. In some literature, such as (Muchnick 1997; Aho, Lam, Sethi, and Jeffrey D Ullman 2007; Cooper and Torczon 2011) many of the phases of the *midend* are grouped into the backend phase of the compiler. This figure is based upon Figure 1.5 in (Muchnick 1997).

The backend is the last module of a compiler, and its responsibility is code generation (Muchnick 1997; Aho, Lam, Sethi, and Jeffrey D Ullman 2007; Cooper and Torczon 2011). The intermediate representation of the backend is the low-level intermediate representation (LiR). This representation is similar to MiR; however, it is designed to be close to a possible target architecture or target language while still being target-independent (Muchnick 1997). A modern example of a LiR is the LLVM intermediate representation LLVM-IR first introduced in *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation* (Lattner and Adve 2004). Several modern compilers employ LLVM-IR as a LiR, examples include *the Clang compiler*, the *Rust Compiler* and *the Julia compiler* (Bezanson, Edelman, Karpinski, and V. B. Shah 2017).

3.1.2 Interpreters

The distinction between interpreters and compilers is not always clear. However, compilers that exhibit the behavior of interpreters and perform code generation as a part of running the program while generating machine code are called Just-in-time compilers. Interpreters may have more or less the same structure as a compiler. However, what sets them apart from compilers is that while compilers typically generate native code at the end of the compilation process, interpreters instead produce a result based on the direct interpretation of some final intermediate representation such as a LiR (Romer, D. Lee, Voelker, Wolman, Wong, Baer, Bershad, and Levy 1996).

In the next section, we present some of the characteristics of Just-in-time compilers.

3.1.3 Just-in-time Compilers

Just-in-time compilers (JIT-Compilers) are contrasted with so called Ahead-of-time compilers (AOT-Compilers). The previous discussion in Section 3.1.1 illustrated in Figure 3.2 describes Ahead-of-time compilation (AOT-Compilation). In AOT-Compilation the result of the compilation process is a semantically equivalent program in some target language. This program can later be executed native on the machine CPU or in a virtual machine.

In contrast, during Just-In-Time Compilation[7] (JIT-Compilation), code is not compiled immediately. Instead, the compilation is postponed until that particular code is required. The compilation process occurs during program execution. The operations of JIT-Compilers are similar to certain interpreter configurations in the sense that code is executed while the program is running. However, what makes JIT-Compilers distinct from interpreters is that while the interpreter interprets the code, a JIT compiler typically generates native machine code for some target architecture or virtual machine.

[7] Also known as dynamic compilation.

JIT-Compilers have several advantages compared to AOT-Compilers; one is that they can support more expressive languages allowing constructs for self-modifying code[8]; another is that JIT-Compilers allow runtime optimization. What is meant by runtime optimization is that the compiler can dynamically optimize the program according to some heuristic while the program is executing (Aycock 2003).

However, the flexibility of JIT-Compilation comes with a cost. Since compilation occurs during runtime, typical schemes that are used during AOT-Compilation such as *register allocation* (Chaitin, Auslander, Chandra, Cocke, Hopkins, and Markstein 1981) may be to expensive to employ. Thus, the number of static optimization passes employed in JIT-Compilers are typically limited. Historically this has led to compilers that can employ just-in-time compilation to use schemes such as only compiling a fraction of the program or using JIT-Compilation in combination interpretation for functions that are seldom called (Aycock 2003).

In this section, we provided a brief overview of compilers for typical procedural languages. In the next section we introduce *equation-based modeling languages* and *equation-based object-oriented modeling languages*. While compiler construction principles remain the same for these languages, there are domain-specific characteristics and challenges. We start by discussing equation-based modeling languages in general and proceed by discussing the compilation of such languages in particular and how this process is related to the theory in the previous chapter.

3.1.4 Compilers for Declarative Languages

Declarative equation-based languages differ from procedural languages in the sense that you describe problems that the computer is to solve rather than specifying the steps involved in solving the problem. The equation-based language Modelica is an example of a declarative language; another example is the Prolog language. Some equation-based languages, such as Omola (Andersson 1990) are fully declarative, whereas others, such as Modelica and MetaModelica, contain both declarative and procedural elements (P. Fritzson 2014). This means that compilers for equation-based languages need to handle both symbolical and numerical methods as a part of the compiler pipeline.

Consequently, the model as presented in Figure 3.2 apply to compilers for equation-based languages the contents of each *box* is slightly different. For instance, the final program needs to generate code such that the equations of the program can be solved numerically. Either by using an external numerical solver or it needs to generate logic that implements a solver algorithm.

[8]The Lisp programming language is regarded to be the first example (Aycock 2003).

3.2 Equation-based modeling languages

The advent of computers and later programming languages made it possible to implement and run mathematical models using computers. While the first programming languages were procedural, dedicated domain-specific modeling languages were developed shortly after in the 1960s.

An equation-based modeling language is a language that allows the user to model systems using equations rather than writing algorithms or, in other ways, predefine causality[9,10]. Thus, equation-based languages fall in the category of declarative languages. Declarative languages are written according to the paradigm of declarative programming. In declarative programming, the programmer specifies the problem to solve, not how to solve the problem through an algorithm. In imperative programming, the programmer provides an explicit algorithm describing the steps to solve a given problem, e.g., providing the computer with the sequence of steps to execute.

In equation-based programming languages, this sequence of steps is instead derived from the language's textual representation of the equations[11]. This derivation is achieved by translating a given textual or graphical representation into one or more intermediate representations. The process is similar to that described by Figure 3.2[12]. This section restricts the discussion to such languages and provides a brief overview of equation-based modeling languages. We start by enumerating some historical languages and discuss the Modelica language.

3.2.1 Historical modeling languages

In this section, we briefly summarize some historical modeling languages. We start by discussing early procedural languages for modeling and simulation. We then discuss object-oriented languages and how they relate to simulation.

3.2.1.1 Early languages

Equation-based modeling languages have been around since the 1960s. One early example of a standardized modeling language for scientific computing is the Continuous System Simulation Language (CSSL) which provided extensions to the earlier *MIMIC* (H.E and F.J Sansom 1965) language (Augustin, Fineberg, Johnsson, Linebarger, and F.John Sansom 1967). The language was

[9] The causality of an equation-based language is inferred by the compiler.

[10] There exist modeling frameworks with fixed causality, a notable example being Simulink; however, modeling languages, in general, are outside the scope of this thesis (Chaturvedi 2017).

[11] While some programming environments provide a graphical user interface that allows the user to specify a model using graphical components, this representation is converted to a textual description before compilation.

[12] A compiler for an equation-based declarative language is different from a procedural in some aspects, some of these will be treated later in this chapter.

Listing 3.2.1 Small CSSL example program (Augustin, Fineberg, Johnsson, Linebarger, and F.John Sansom 1967, p .283).

```
DX = INTEG[F - B*X - A*DX, DX0]
X = INTEG[DX, X0]
```

designed to allow users to solve ODE systems and simulate them on digital computers[13].

$$\begin{aligned} \dddot{x} + a\dot{x} + bx &= f(t) \\ \dot{x}_0 &= DX0 \\ x_0 &= X0 \end{aligned} \tag{3.1}$$

In Listing 3.2.1 the first equation of Equation (3.1) has first been manually preprocessed and rewritten explicitly as $\dddot{x} = f(t) - a\dot{x} - bx$ which is required of the integration operator *INTEG*. Note that programming in CSSL is not declarative as described in (Augustin, Fineberg, Johnsson, Linebarger, and F.John Sansom 1967). However, it is equation-based in that it allows the user to specify numerical operators to solve equations and combine these with procedural procedures mixing procedural code together with simulation operators such as *INTEG*.

A CSSL program was structured according to three regions of operation:

- Initial region
- Dynamic Region
- Terminal Region

Where the initial region handled initial conditions, the computations were run each time step in the dynamic region, and the final region was responsible for the final postprocessing (Augustin, Fineberg, Johnsson, Linebarger, and F.John Sansom 1967).

The Advanced continuous simulation language ACSL extended the functionality of CSSL. It introduced another region, the derivative region. This region was responsible for solving the equations for the state variables. Unlike the initial CSSL design the order of the equations in this region did not matter. Instead, the equations were automatically sorted according to the dependencies between them (Mitchell and Gauthier 1976).

[13] Previously analog computers had been the main platform for mathematical simulations.

Another language that CSSL inspired was Simnon (Elmqvist 1977). Similar to ACSL, Simnon allowed unordered equation sections. One feature introduced by Simnon was the possibility to connect subsystems so that systems could be defined in isolation as individual components (Elmqvist 1977). While tendencies towards object orientation in terms of macros to facilitate reuse of systems and procedures in the case of CSSL and ACSL and the notion of subsystems in Simnon can be observed in these earlier languages, they were not explicitly object-oriented.

With inspiration from Simula (Capretz 2003) and the previously mentioned language Simnon, Dymola was developed (Elmqvist 1978). Dymola built upon some of the concepts introduced by Simnon. To give an example, in Simnon, a model might have one submodel; however, a model in Dymola might have a submodel which in turn might have other submodules. Dymola makes use of object-oriented concepts such as composition[14] and abstractions to connect subsystems together using the *cut* to denote the sets of variables involved and *connect* to connect them. However, Dymola as described in (Elmqvist 1978) was not explicitly object oriented (Jobling, P. W. Grant, Barker, and Townsend 1994). In the next section, we discuss object-oriented equation-based languages.

3.2.1.2 Object-oriented modeling languages

The following section introduces some historical object-oriented equation-based modeling languages. An explicit object-oriented modeling language was Omola (Andersson 1992) with concepts such as abstract classes[15] and abstractions to enforce encapsulation.

In Listing 3.2.2 we can see an example of the object-oriented features of Omola. In the listing, the basic tank model extends the tank model. Like Dymola, Omola had support for concepts such as model parameters, connections, and submodels. However, a drawback of Omola was that it lacked constructs for modelers to express procedural algorithms (P. Fritzson and D. Fritzson 1992; Andersson 1990).

Another early object-oriented programming language was ObjectMath (Viklund, Herber, and P. Fritzson 1992). ObjectMath combined features of languages dedicated to computer algebra with features of object-oriented simulation languages and matrix languages (Viklund and P. Fritzson 1995). In the article *The need for High-Level programming Support in Scientific Computing applied to Mechanical Analysis*, *Omola* and *Dymola* was mentioned as influences (P. Fritzson and D. Fritzson 1992) along with the Mathematica language (Wolfram 1991).

An additional early object oriented modeling environment was *ASCEND* (Piela, Epperly, K. M. Westerberg, and A. W. Westerberg 1991). Andersson

[14] Via something called submodel decomposition.

[15] In the licentiate thesis by Andersson referred to as primitive models.

Listing 3.2.2 Example of a Omola model for a tank model. The Tank model extends the basic tank. (Andersson 1990, p. 35).

```
Basic_Tank ISA Model WITH
  terminals:
    inflow ISA Terminal;
    outflow ISA Terminal;
   parameter:
     tank_area TYPE Real := 5.0;
   END;
  Tank ISA Basic_Tank WITH
    realization:
      mass_balance ISA SetDAE WITH
      variable:
        level TYPE Real :=0;
  equation:
    tank_area * dot(level) = inflow - outflow;
  END;
END;
```

(1990) states that the ASCEND language is similar to Omola in that it follows the object-oriented modeling paradigm.

3.2.1.3 Other languages

While the languages discussed in the previous sections deals with the simulation of physical systems, another modeling paradigm was that of system dynamics developed by J.W Forrester (Forrester 1993). Industrial dynamics attempts to model feedback loops through differential equations describing social systems such as a companies (Forrester 1968). The methodology was also applied to other contexts outside the industry, such as urban planning, demonstrating the practical use of system dynamics (Forrester 1970).

We have reviewed aspects concerning the development of equation-based language and provided a brief historical overview. For a more exhaustive treatment concerning the evolution of languages for simulation see *Evolution of continuous-time modeling and simulation* (Åström, Elmqvist, Mattsson, et al. 1998); *A perspective on modeling and simulation of complex dynamical systems* (Åström 2011) and (Jobling, P. W. Grant, Barker, and Townsend 1994).

3.2.1.4 Modelica

Version 1.0 the Modelica language specification was introduced in 1998 (Elmqvist, Mattsson, and Otter 1998; P. Fritzson and Engelson 1998) with

Listing 3.2.3 The Modelica HelloWorld model (P. Fritzson 2014, p. 20)

```
model HelloWorld
  parameter Real a = 1;
  Real x(start = 1, fixed = true;
equation
  der(x) = - a * x;
end HelloWorld;
```

Listing 3.2.4 The Pin model in A.2.1

```
connector Pin
  Real v;
  flow Real i;
end Pin;
```

the goal of provided a unified language for system modeling and simulation[16]. Modelica is a declarative and object-oriented equation-based modeling language (P. Fritzson 2014). Similar to the languages presented in the previous section, the Modelica language was designed to describe model behavior using equations and then compose these models to define systems using an object-oriented paradigm. Moreover, unlike the earlier Omola language, the first version of Modelica included the capability of expressing parts of models using non-declarative (imperative and causal) algorithms and functions.

The key construct in Modelica is the model or the class. To illustrate the object-oriented features of Modelica, the circuit in the previous chapter Figure 2.6 can be expressed using subclasses, one for each component.

In Listing A.2.1 in the appendix several electrical components are defined using Modelica. Firstly the Pin model (Listing 3.2.4) is defined. Pin is declared as a connector. A connector is a specialized class in Modelica, and it can be thought about as a datatype that specifies a set of components that may interface with other connectors using the *connect* statement[17].

The Pin class, has two variables, voltage v represented using a Real variable and one *flow* Real variable i.

The flow construct in Modelica indicates that the sum of the connections that use the variable must sum to zero. Recall that in the previous chapter the system of equations for the RLC circuit included equations such as $G.p.i + C.n.i + L.n.i + AC.n.i = 0$ these equations are the results of Kirchhoff's circuit laws which states $\sum_{k=1}^{n} I_k = 0$.

[16] It is a unifying language in the sense that it standardized features present in existing languages such as Dymola, Omola and ObjectMath.

[17] The connector type is similar to the cut in Dymola (Elmqvist 1978).

Listing 3.2.5 The TwoPin model in A.2.1

```
partial model TwoPin
  Real v;
  Real i;
  Pin p;
  Pin n;
equation
  v = p.v - n.v;
  0 = p.i + n.i
  i = p.i;
end TwoPin;
```

Listing 3.2.6 The Resistor model in A.2.1

```
model Resistor
  extends TwoPin;
  parameter Real R;
equation
  R * I = V
end Resistor;
```

Modelica supports both inheritance and composition. Using inheritance we declare the model TwoPin, similarly to the primitive classes of Omola. Modelica, supports an equivalent construct, so-called partial models. The keyword *partial* indicates that the model is not complete and can not be used on its own; rather, the model is to be extended by some concrete model. Using these models we can represent a resistor as in Listing 3.2.6 by inheriting from the TwoPin model in Listing 3.2.5. The single equation of the resistor model is derived using Ohms law: $V = R \cdot I$.

Similarly, by using physical laws combined with the Pin and TwoPin constructs, we can declare models for other electrical components, see Listing A.2.1. These are combined to construct a model for the circuit in Figure 2.6. In the example, we can see the heritage that Modelica shares with previously discussed languages, such as the connections, inheritance, and composition. To model the circuit a new model is created using these basic elements as building blocks.

The final Modelica model of the circuit is illustrated in Listing 3.2.7.

While the language is mainly declarative, the Modelica language also supports a procedural subset via functions and algorithm sections. In this way, Modelica is capable of formulating discrete procedural models.

Similar to previous equation-based languages, Modelica is not limited to electrical circuits. Modelica has an extensive standard library encompass-

Listing 3.2.7 A Modelica model of the RLC circuit in Figure 2.6 (P. Fritzson 2014, p. 35).

```
model RLCCircuit
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  Source AC(A = 1.0, w = 1.0);
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, C.p);
  connect(C.n, AC.n);
  connect(R1.p, R2.p);
  connect(R2.n, L.p);
  connect(L.n, C.n);
  connect(AC.n, G.p);
end RLCCircuit;
```

ing several domains, from electronics to mechanics. Several modeling and simulation environments supports the Modelica language. Examples include OpenModelica (P. Fritzson, Pop, Abdelhak, Asghar, Bachmann, W. Braun, Bouskela, R. Braun, Buffoni, Casella, et al. 2020), Modelon Impact[18] and Dymola[19].

3.2.1.5 MetaModelica

While Modelica does provide elements of procedural languages such as functions and algorithms such that Modelica is Turing complete, the language was not explicitly designed for semantic modeling. MetaModelica extends the Modelica language with features common in functional programming languages, such as pattern matching and recursive datatypes (Pop and P. Fritzson 2006).

MetaModelica is at the time of writing the implementation language of the OpenModelica Compiler OMC (P. Fritzson, Pop, Abdelhak, Asghar, Bachmann, W. Braun, Bouskela, R. Braun, Buffoni, Casella, et al. 2020).

Listing 3.2.8 illustrates the uniontype data type of MetaModelica. In this listing a real expression is defined the *Exp* type is defined to either be a node representing a real number constant such as 3.14 or one of five arithmetic operators.

[18] URL: https://www.modelon.com/modelon-impact/, accessed 2022-02-15

[19] URL: https://www.3ds.com/products-services/catia/products/dymola/, accessed 2022-02-15

Listing 3.2.8 Use of the uniontype construct in MetaModelica to define a syntax tree for arithmetic expressions (Pop and P. Fritzson 2006, p. 220).

```
uniontype Exp
  record RCONST Real x1; end RCONST;
  record PLUS Exp x1; Exp x2; end PLUS;
  record SUB Exp x1; Exp x2; end SUB;
  record MUL Exp x1; Exp x2; end MUL;
  record DIV Exp x1; Exp x2; end DIV;
  record NEG Exp x1; end NEG;
end Exp;
```

Listing 3.2.9 A function to evaluate expressions defined using the uniontype in Listing 3.2.8 (Pop and P. Fritzson 2006, p. 222).

```
function eval
  input Exp in_exp;
  output Real out_real;
algorithm
  out_real := match in_exp
    local Real v1,v2,v3; Exp e1,e2;
    case RCONST(v1) then v1;
    case ADD(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 + v2;
      then v3;
    case SUB(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 - v2;
      then v3;
    case MUL(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 * v2;
      then v3;
    case DIV(e1,e2)
      equation
        v1 = eval(e1); v2 = eval(e2); v3 = v1 / v2;
      then v3;
    case NEG(e1)
      equation
        v1 = eval(e1); v2 = -v1;
      then v2;
  end match;
end eval;
```

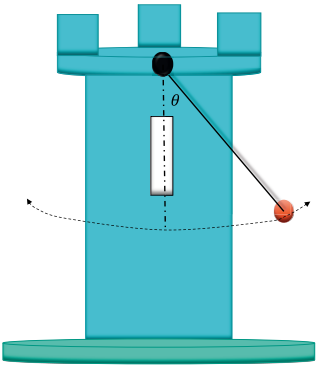

Figure 3.3: A ball attached to a pendulum located at the top of a tower.

To evaluate constructs represented by the *uniontype* data types, MetaModelica pattern matching can be used. Listing 3.2.9 demonstrates how pattern matching together with recursion is used to define a function for evaluating expressions defined by the uniontype in Listing 3.2.8.

3.3 Equation-based languages with variable structure

In the previous section we have enumerated several historical languages and we have introduced the Modelica language, along with the MetaModelica extension. In this section we discuss another subset of equation-based languages that supports a variable structure. A language with a variable structure means that the program's structure may change during simulation. Examples of structural variability could be a ball joint breaking on a automobile, a revolute joint breaking on a door handle or if the pendulum depicted in Figure 3.3 breaks. The scenario with the breaking pendulum, can be described more formally by Equation (3.2).

$$f = \begin{cases} g(t, \underset{\rightarrow}{\dot{x}}(t), \underset{\rightarrow}{x}(t), \underset{\rightarrow}{y}(t), \underset{\rightarrow}{p}), & \text{if } t \geq 1 \\ h(t, \underset{\rightarrow}{\dot{\chi}}(t), \underset{\rightarrow}{\chi}(t), \underset{\rightarrow}{\gamma}(t), \underset{\rightarrow}{p}), & \text{Otherwise} \end{cases} \tag{3.2}$$

When the pendulum breaks, in this example when $t \geq 1$, it is no longer a pendulum, but a falling object as in Figure 2.1. When representing this scenario with Equation (3.2) the dynamics of the pendulum is defined by g and when the pendulum breaks the dynamics are defined by h.

All systems around us have some structural variability. Besides providing increased modeling capabilities, support for varying the model's structure during simulation also has other advantages. For example, the model can change the granularity of a specific subsystem during the simulation. With this approach, the dynamics of the entire system need not be specified from

the beginning. Instead, the model can make use of low complexity idealization at the start of the simulation and switch to a more granular description when some conditions are met. For example, if we model a water damn, we might not be interested in calculating the mechanics of materials until the amount of water has reached a certain threshold. Thus, computational resources can be saved since only part of the entire system needs to be specified initially.

Due to the advantages of a modeling language that supports systems with varying structure, several languages and environments have been developed to support this paradigm. However, as it turns out, extending an equation-based language to support variable structure leads to additional complications in terms of language design.

In this section, we present a brief overview of languages and environments dedicated to extending or providing support for modeling structural variability within the context of equation-based modeling.

3.3.1 Mosilab

An extension to the Modelica language to allow models with variable structure is the Modeling and Simulation Language (MOSILA) within the programming environment Mosilab (Nytsch-Geusen, Ernst, Nordwig, Schneider, Schwarz, Vetter, Wittwer, Holm, Nouidui, Leopold, et al. 2005). Mosilab is considered to be the first programming environment with the explicit goal of being capable of modeling systems with variable structure (Zimmer 2010).

To support the modeling of systems with variable structure, Mosilab introduces *Dynamical object structures*. These structures can be activated or deactivated during discrete time events. The Mosilab environment also provides a visual methodology to represent these systems using statecharts based on UML (Nytsch-Geusen, Ernst, Nordwig, Schneider, Schwarz, Vetter, Wittwer, Holm, Nouidui, Leopold, et al. 2005).

In Figure 3.4 we can see an example of a state chart describing the behavior of a landing device. At the start of the simulation, the model enters the state *moving*; if the speed by which the device is descending becomes too great, boosters are activated to stabilize it. Finally, when the device has landed, it enters the state *stop*.

From the state chart depicted in Figure 3.4 the code in Listing 3.3.1 is derived. In the figure we can see how the system transitions between the different states as depicted in Figure 3.4. However, as can be seen in the example above, a disadvantage of the Mosilab approach is that the dynamic objects of the model must be specified explicitly before simulation.

Listing 3.3.1 The Model describing the state chart in Figure 3.4 (Nytsch-Geusen, Ernst, Nordwig, Schneider, Schwarz, Vetter, Wittwer, Holm, Nouidui, Leopold, et al. 2005, p. 528).

```
model System
...
statechart
  state SystemSC extends State;
  state Moving extends State;
  state SlowDown extends State;
    exit action
      body.remove(boost);
    end exit;
  end SlowDown;

  State falling, start(isInitial=true);
  SlowDown slowDown;

  transition start -> falling end transition;
  transition t2 : falling -> slowDown
      event sw guard sw==1 action
      body.add(boost);
  end transition;
  transition t3 : slowDown -> falling
      event sw guard sw==0
  end transition;
  end Moving;

  State stop, start(isInitial=true);
  Moving moving;

  transition t1 : start -> moving action
    body.add(gr);
  end transition;
  transition t4 : moving -> stop
    event landed action
    body.remove(gr);
  end transition;
  end SystemSC;
end System;
```

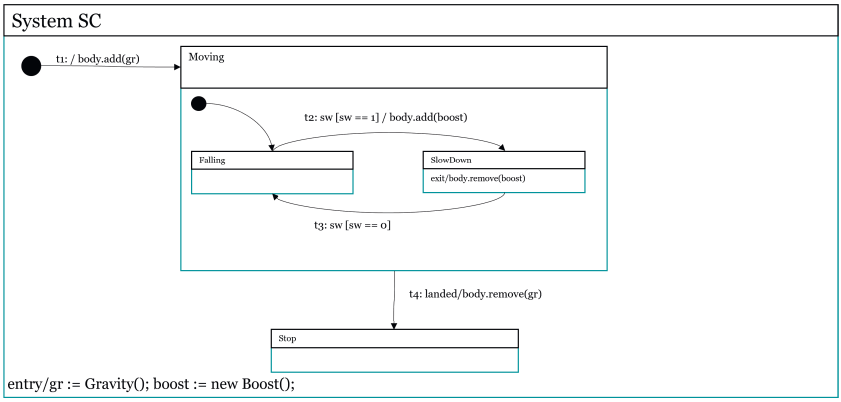

Figure 3.4: State chart describing a landing system in Mosilab. The figure is adapted from Figure 1. in (Nytsch-Geusen, Ernst, Nordwig, Schneider, Schwarz, Vetter, Wittwer, Holm, Nouidui, Leopold, et al. 2005)

3.3.2 The Sol language

While Mosilab solves the problem by allowing models to exhibit dynamic behavior during simulation, it requires the modeler to describe the system's state prior to simulation. Individual components of the system may also not modify themselves (Zimmer 2010).

To overcome the limitations of the Mosilab approach and extend the expressiveness of Modelica, Zimmer proposed the Sol language (Zimmer 2010). The Sol language, while superficially similar to Modelica in many aspects, is a separate language designed to support the handling of VSS using the framework *SolSim*. The key addition of the sol language is the addition of new operators to change component during discrete time events:

- The copy transmission operator, «
- The move transmission operator, <-

The copy transmission works like an assignment in procedural languages such as C. That is, the value of the right-hand side is assigned to the object on the left-hand side. The move transmission operator is similar to the copy operator; however, it transfers the ownership of the object to the destination component [20]. The move operator in Sol can be used, for instance, to indicate that one object should replace another given that certain conditions are met.

One example of how structural variability is achieved in Sol can be seen in Listing 3.3.2. The model consists of two states, Engine1 and Engine2, where Engine2 is more computationally expensive to simulate compared to Engine1. However, during the simulation, the dynamics of the engine change due to the relationship between the inertia and the torque. Because of this, the level of detail of Engine2 is no longer needed, and Engine1 can be used instead to speed up the simulation process. This translation is captured by the $F.w > 40$ condition in the when equation which in turn results in the exchange of the engine model.

3.3.2.1 Handling structural change in Sol

During simulation the following cases can occur and is handled by the simulator for the Sol language, *SolSim*.

- Introducing a new variable
- Removing a variable
- Introducing a new relation
- Removing a relation

[20] This is similar to how the move semantics work in C++ (Stroustrup 2013).

Listing 3.3.2 A Machine model with structural change from (Zimmer 2010, p. 78)

```
model Machine
  implementation:
    static Mechanics.FlyWheel F{inertia << 1}
    static Mechanics.Gear G{ ration << 1.8}
    connection{a << G.f2, b << F.f};
    static Boolean fast;
    if fast then
      static Mechanics.Engine1 E{meanT << 10};
      connection{a << E.f, b << G.f1};
    else then
      static Mechanics.Engine2 E{meanT << 10};
      connection{a << E.f, b << G.f1};
    end;
    if initial() then
      fast << false;
    end;
    when F.w > 40 then
      fast << true;
    end;
end Machine;
```

The SolSim simulator handles these changes by dynamic DAE processing. The simulator calculates these using symbolic operations to dynamically account for changes in the set of equations and variables using dynamic topological sorting (Pearce and Kelly 2004).

3.3.3 Hydra

Hydra (Giorgidze 2012) is a deeply embedded language implemented in Haskell according to the paradigm of functional hybrid modeling (Nilsson, Peterson, and Hudak 2003). Hydra supports most operations of acausal modeling languages but lacks the object-oriented features present in languages such as Modelica. Hydra compensates for the lack of these capabilities by enabling more flexibility compared to the relatively static Modelica language. One example of this flexibility is handling systems where the set of equations and variables change during simulation. Hydra handles this issue by utilizing JIT-Compilation, see Section 3.1.3.

3.3.3.1 Recompilation during mode change in Hydra

Hydra, as presented in the thesis by Giorgidze (Giorgidze 2012), does not cache the equations between mode changes. Instead, the system is recompiled. This means that equations that were not subject to change during the mode switch are recompiled regardless. Conversely, in the SolSim modeling and

Listing 3.3.3 A breaking pendulum model described using Hydra[21].

```
g :: Double
g = 9.81
freeFall :: Body -> SR Body
freeFall ((x0,y0),(vx0,vy0)) = [rel| ((x,y),(vx,vy)) ->
                                 init (x,y)   = ($x0$,$y0$)
                                 init (vx,vy) = ($vx0$,$vy0$)
                                 (der x,der y) = (vx,vy)
                                 (der vx,der vy) = (0.0, - $g$)|]
pendulum :: Double -> Double -> SR Body
  pendulum l phi0 = [rel| ((x,y),(vx,vy)) ->
      local phi
      local phid
      init phi  = $phi0$
      init phid = 0
      init (x,y) = ($l$ * sin $phi0$, - $l$ * cos $phi0$)
      phid = der phi
      (x,y) = ($l$ * sin phi, - $l$ * cos phi)
      (vx,vy) = (der x, der y)
      der phid + ($g$ / $l$) * sin phi = 0|]
breakingPendulum :: Double -> Double ->
  Double -> SR Body
  breakingPendulum t l phi0 = switch (pendulum l phi0)
                              [fun| ((_,_),(_,_)) -> time - $t$ |] freeFall
....
```

simulation environment, heuristics are used to estimate which equations need to be modified during a mode switch (Zimmer 2010).

Another difference is that while the Sol language is interpreted by an interpreter, the realization of the Hydra language instead generates machine code. While caching equations would improve performance, the generation of machine code was discovered to be the main bottleneck (Giorgidze 2012).

Listing 3.3.3 illustrates a breaking Pendulum in Hydra. Initially the pendulum is active until $t = 5.0$ seconds when the active model change from the pendulum model to the free fall model.

3.3.4 The Model Composition Language and Nano Modelica

In *Compiling Modelica* Höger presents both a theoretical framework and an experimental prototype that is capable of handling systems with a dynamic structure as well as separate compilation (Höger 2019). In this section, we focus on the extensions to the language and the computational framework to achieve VSS.

[21] URL: https://github.com/giorgidze/Hydra/blob/e5b59d0baff27f06368caeb1e9860f1395913ca5/examples/BreakingPendulum.hs accessed 2022-05-03

Höger introduces two new operators to a subset of the Modelica language, *Nano Modelica*:

- *resume*
- *resuming*

Along with the attribute *restart* and a new class *Checkpoint*.

A Modelica model specified in Nano Modelica is translated to some Hybrid DAE representation. This representation is then translated to the Model Composition Language (MCL). MCL is inspired from a previous core language, the Model Kernel Language (MKL) (Broman 2010). The purpose of the kernel language is to provide a formal framework to describe the semantics of modeling languages[22]. It is used as an IR in the translation process before finally being transformed into OCaml[23].

To be noted that MCL is not Modelica; rather, it is a concise language meant to encapsulate the behavior of a subset of Modelica called Nano Modelica. It is also used to implement the simulation runtime of the models that are being simulated.

The breaking pendulum example is also discussed in the work of Höger, the use of the *Checkpoint*, *resume* and *resuming* can be seen in Listing 3.3.4. Initially the constraint equation $x^2 + y^2 = 1$ is active along with the equations governing the dynamics of the pendulum:

$$\begin{cases} \dot{x} = vx \\ \dot{y} = vy \\ \dot{vx} = F \cdot x \\ \dot{vy} = F \cdot y - 9.81 \end{cases} \tag{3.3}$$

until y exceeds 0.6, after which the new equation $F = 0$ is activated, which results in the object attached to the pendulum entering the free fall state. This approach to representing the model is different when contrasted with the model in Hydra. See Listing 3.3.3. In the latter case, the free fall state and the attached state are explicitly encoded, whereas in the example by Höger, activating and deactivating equations change the behavior.

3.3.5 Comparing languages and programming environments for Variable Structured Modeling

In the previous section, we have discussed different languages and programming environments that attempt to achieve support for systems with variable

[22] In the context of the thesis this is used to describe the semantics of the Modelica subset mapped to MKL, however, it is not limited to Modelica alone, since conceivably similar equation-based languages could be mapped to it.

[23] URL: https://ocaml.org/, accessed 2022-02-15.

Listing 3.3.4 The Breaking Pendulum from (Höger 2019, p. 171).

```
model Pendulum
  constant Real pi = 2*asin(1.0);
  Real x(start=sin(3.*pi/4.));
  Real y(start=cos(pi/4.));
  Real vx, vy ,F;
  Checkpoint cp;
initial equation
  vx=0; vy=0; x = sin(3.*pi/4.); y = cos(pi/4.);
equation
  der(x) = vx; der(y) = vy;
  der(vx) = F*x; der(vy) = F*y - 9.81;
  if (not resuming(cp)) then
    x*x + y*y = 1;
    when (y > 0.6) then
      resume(cp);
    end when;
  else
    F = 0;
  end if;
end Pendulum;
```

structure. Within the context of Modelica, there seems to be a consensus that Mosilab represents the first attempt to extend Modelica for this purpose (Zimmer 2010; Giorgidze 2012; Höger 2019). Concerning the Hydra language, Zimmer states:

> *Hydra is based on the paradigm of functional hybrid modeling. This makes it a powerful language. In principle, it is possible to state arbitrary equation systems with Hydra and to formulate arbitrary changes. Also new elements can be generated at runtime. Practically, the simulation engine is currently not able to support higher-index systems to a sufficient extent. Also the language has not been tested on complex modeling examples. The way Hydra is processed is rather unique in the field of M&S. Hydra features a just-in-time compilation. At each structural change, the model is completely recompiled in order to enable a fast evaluation of the system. This processing scheme makes Hydra interesting with respect to Sol since it represents a contemplative approach. Whereas Sol, being an interpreter, is efficient in handling the changes in the system of equations but inefficient in the evaluation stage, Hydra represents the opposite case. A*

> *combination of both approaches would therefore lead to an optimal trade-off between flexibility and efficiency. Zimmer 2010*

p. 49

Concerning the Sol language Giorgidze states:

> *Sol is a Modelica-like language [Zimmer, 2007, 2008]. It introduces language constructs that enable the description of systems where objects are dynamically created and deleted, thus supporting modelling of unbounded structurally dynamic systems. The work on Sol is complementary to ours in a number of respects outlined in the following. Sol explores how structurally dynamic systems can be modelled in an object-oriented, noncausal language. Hydra extends a purely functional programming language with constructs for structurally dynamic, noncausal modeling.*
>
> *The implementation of Sol makes use of symbolic methods that for each structural change aim to identify the smallest number of equations that need to be modified or added in order to model the structural change. It would be interesting to combine these symbolic methods with the runtime code generation approach used in Hydra in order to reduce the JIT compilation overheads by only compiling the modifid and added equations for each structural change.*
>
> *Sol features only an interpreted implementation. The dynamic compilation techniques featured in the implementation of Hydra would be of interest in the context of Sol to enable it to target high-end simulation tasks. Giorgidze 2012*

p. 108

Furthermore, both Zimmer and Giorgidze states that the main disadvantage of Mosilab is the explicit structure, moreover, it does not allow the modeler to model an unbounded system with structural variability (Zimmer 2010; Giorgidze 2012).

Concerning MCL, the thesis by Höger leaves the following final remark:

> *Throughout this thesis, we have deviated quite far from the topics that are usually discussed in the context of*

> *Modelica. We have only sketched the surface of the numerical and symbolical treatment of equations and done so only to provide the necessary framework for some very specific aspects of its operational semantics. While we considered it important to fully support the simulation of at least a very small subset of the language, we did not bother with realistic models or aspects like simulation performance. Instead, we focused on a specific architecture of implementation, namely that of a rigorous, separate compiler, and how its choice enables a new modeling paradigm. Höger 2019*
>
> *p. 206*

From this, we can infer that while the work of Höger provides an extensive formal overview concerning separate compilation and the modeling of highly dynamic systems, it does not provide any empirical evaluation concerning the efficiency of his approach.

A summary of the characteristics of the approaches by (Höger 2019; Giorgidze 2012; Zimmer 2010; Nytsch-Geusen, Ernst, Nordwig, Schneider, Schwarz, Vetter, Wittwer, Holm, Nouidui, Leopold, et al. 2005) in Table 3.1.

In Table 3.1 we categorize the approaches according to the following characteristics:

- Type
- Paradigm
- Boundness
- Variability
- Declaration Scheme
- Higher-Order Models

The *Type* and the *Paradigm* are discussed in the previous sections, along with the description of the compilation technique. What is more interesting is the concept *Boundness*. Boundness in the context of VSS is defined not in terms of how many distinct modes of operation models might have but rather if the language is capable of formulating models that may in turn, generate new modes. In the case of Mosilab, a model may have infinite modes, but this set of modes must be specified before simulation, which is unfeasible. However, in the case of *Sol*, *MCL* and *Hydra* new modes may be created during simulation.

Table 3.1: Characteristics of languages that are able to express system with structural variability.

	Mosilab	Sol	Hydra	MCL
Type	Modelica extension	Modelica variant	Embedded in Haskell	Intermediate Representation
Paradigm	Declarative	Declarative	Functional	Functional
Compilation technique	AOT-Compilation	Interpretation	JIT-Compilation	AOT-Compilation
Variability	Static	Dynamic	Dynamic	Static
Declaration Scheme	Explicit	Implicit	Explicit	Implicit
Boundness	Bounded	Unbounded	Unbounded	Unbounded
Higher-order-models	No	Yes	Yes	Yes

Variability, concerns the modeling and simulation framework. Framework that are able either to extend or add to the program itself via metaprogramming after starting the simulation are categorized as *Dynamic*, otherwise we say that it is *Static*.

Declaration Scheme, concerns how structural variability is expressed syntactically, ether explicitly or implicitly. One example of an explicit declaration is in Listing 3.3.3 where the structural dynamics are expressed using the switch operator. Another is Listing 3.3.1 where structural changes are expressed via structural transitions. This can be contrasted with the works of Höger and Zimmer where an operator in the code affects the resulting mode.

An example of implicit transitions is how the constraint equation in Listing 3.3.4 change the active mode.

The last category is *Higher-order-models*, it implies that simulated model is able to directly reconfigure models as if models themselves are variables, either explicitly or implicitly.

The themes identified in this overview have illustrated that simulation of variable structured systems is possible. However, how these changes are expressed and the expressive power of proposed constructs vary. Nevertheless, researchers have not treated the practical implication of a programming environment with VSS support in detail. In general, no attempt was made to quantify the implication of modeling systems according to this paradigm on large examples. Only Giorgdize provides empirical insight regarding the practicality of his approach.

3.4 Other frameworks

Besides the frameworks discussed above, variable structure systems in equation-based languages are discussed in (Mehlhase 2014; Elmqvist, Matsson, and Otter 2014; Elmqvist and Otter 2017). The work by (Mehlhase 2014) is an interesting approach in terms of tool interoperability. However, since it abstracts existing modeling and simulation environments rather than defining or extending a language, we omitted it in Section 3.3. The ideas by Elmqvist, Matsson, and Otter (2014) are similar to those in (Nytsch-Geusen, Ernst,

Nordwig, Schneider, Schwarz, Vetter, Wittwer, Holm, Nouidui, Leopold, et al. 2005). We implement and discuss a similar scheme in Section 4.6.1.

4. OpenModelica.jl a Composable Modelica Environment

This chapter is closely based on the following publications:

- Towards introducing just-in-time compilation in a Modelica compiler (Tinnerholm, Sjölund, and Pop 2019)
- Towards an Open-Source Modelica Compiler in Julia (Tinnerholm, Pop, Sjölund, Heuermann, and Abdelhak 2020)
- OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl (Tinnerholm, Pop, Heuermann, and Sjölund 2021)
- A modular, extensible, and Modelica standard-compliant OpenModelica compiler framework in Julia supporting structural variability, J Tinnerholm, A Pop, M Sjölund (Submitted for publication)

In this chapter we present *OpenModelica.jl*. OpenModelica.jl is a composable modeling and simulation environment written in the Julia language. The term composable means that software components within this programming environment can be used interchangeably. We start by providing an overview of OpenModelica.jl without any implementation specifics in Section 4.1. We then introduce the Julia language and highlight some of the key advantages of providing a Modelica compiler written in Julia and the implications of using such an environment. Section 4.3 presents the MetaModelica extension of Modelica. The section ends with a presentation of MetaModelica.jl a component of OpenModelica.jl and how the homoiconic[1] characteristics of the Julia language is leveraged. In Section 4.4 and Section 4.5 we introduce the compiler frontend and backend respectively. Furthermore, in Section 4.6 we discuss how a set of minimal modifications to the Modelica language allows for more expressive and novel modeling. We end this chapter in Section 4.7 by providing a partial answers to the stated research questions.

4.1 Introducing OpenModelica.jl

OpenModelica.jl is a modeling and simulation environment written in the Julia language dedicated to Modelica modeling and simulation. As of this

[1] Homoiconicity is a characteristic of the Lisp programming language. Homoiconicity means that code is data and data is code (McIlroy 1960).

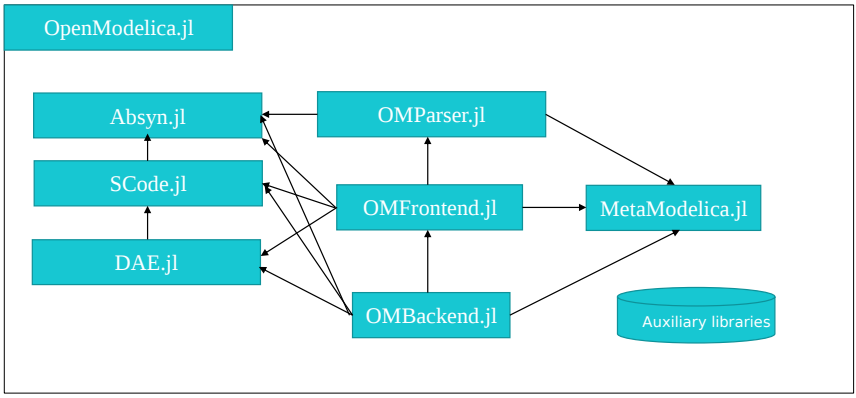

Figure 4.1: An overview of the dependencies between the components in OpenModelica.jl. Absyn.jl, SCode.jl and DAE.jl are existing intermediate representations encoded in the Julia language. The compiler runtime is implemented by MetaModelica.jl. The frontend is provided by the OMFrontend.jl module and the backend by the OMBackend.jl module.

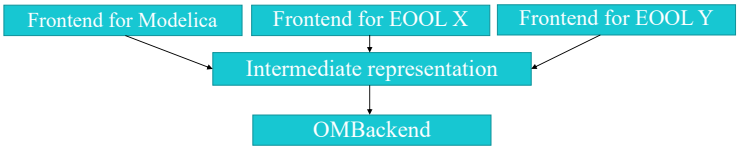

Figure 4.2: A high level overview of a design separating the intermediate representation from the frontend to allow several hypothetical frontends to use the same backend.

writing the OpenModelica.jl programming environment consists of several integrated modules that make up separate pieces of a Modelica compiler.

An overview of the various components of OpenModelica.jl is available in Figure 4.1. In the figure we can see that the backend depends on the frontend. The reason for this dependency is to enable the compiler to dynamically recompile models during simulation. We discuss this feature and its implications in Section 4.6. Following the principles of LLVM (Lattner and Adve 2004) the frontend and the intermediate representation were separated so that additional frontends or backends can be provided to support other EOOL given that they target the same intermediate representation.

An illustration is provided in Figure 4.2.

Listing 4.2.1 An example of a Julia function, addition that adds two variables.

```
function addition(a, b)
  return a + b
end
addition(1.0, 1.0)
addition(1, 1)
```

4.2 The Julia Programming language

The Julia programming language was created to combine the expressive power and flexibility of scientific computing environments such as those existing for Python and Matlab with the performance of compiled procedural languages such as Fortran and C. The Julia language achieves this by utilizing, multiple dispatch, dataflow type inference and runtime JIT-Compilation (Bezanson, Edelman, Karpinski, and V. B. Shah 2017).

Consider the Julia code in Listing 4.2.1. In this example a single function, addition is defined with two arguments a and b. Similar to the Python language which support *duck-typing* this function can be called with different arguments in this case with integer and float arguments. The language differ in while Python interprets the program, Julia will instead infer a function to generate based on the arguments to the function and create native code for each type specialization. In the case of the *addition* function two such specializations will be created one for integer arguments and one for floating point arguments.

In this way Julia retains the flexibility of scripting languages such as Python while at the same time generating high performance code (Bezanson, Edelman, Karpinski, and V. B. Shah 2017). However, the drawback of this approach is that there is an initial overhead in terms of compilation time (Tinnerholm, Sjölund, and Pop 2019).

4.2.1 Scientific computation in Julia

As previously stated, the Julia language was designed to provide a flexible yet performant environment for scientific computing. Consequently, several packages[2] have been developed to facilitate scientific computing. One example of such a package is *DifferentialEquations.jl* (Christopher Rackauckas and Nie 2017). This package allows the user to interface with a number of solver libraries such as SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic Equation Solvers) (Hindmarsh, Brown, K. E. Grant, S. L. Lee, Serban, Shu-

[2] A Julia package is a piece of reusable Julia code. It is similar to modules in other programming languages.

maker, and Woodward 2005). As of 2022, *DifferentialEquations.jl* provides support for the following features[3]:

- Discrete equations (function maps, discrete stochastic (Gillespie/-Markov) simulations)
- Ordinary differential equations (ODEs)
- Split and Partitioned ODEs (Symplectic integrators, IMEX Methods)
- Stochastic ordinary differential equations (SODEs or SDEs)
- Stochastic differential-algebraic equations (SDAEs)
- Random differential equations (RODEs or RDEs)
- Differential algebraic equations (DAEs)

4.2.2 Equation based modeling in Julia

As of 2022, several modeling environments that provide the option of causal and acausal modeling within the Julia ecosystem exists and *DifferentialEquations.jl* (Christopher Rackauckas and Nie 2017) is one such environment. It provides a seamless foreign function interface that allows interfacing algorithmic Julia code and a variety of different solvers. A user of DifferentialEquations.jl writes imperative code in the Julia language to conform to systems such as Nonlinear-systems, ODE-systems, and DAE-systems. Tinnerholm, Pop, Sjölund, Heuermann, and Abdelhak (2020) selected DifferentialEquations.jl as the default backend target. A model of a hybrid system representing a bouncing ball using DifferentialEquations.jl can be studied in Listing 4.2.2.

While DifferentialEquations.jl provides the abstractions necessary to write causal models in Julia, it does not provide the abstractions of a full-fledged modeling language. *ModelingToolkit.jl* (MTK) aims address this issue (Ma, Gowda, Anantharaman, Laughman, V. Shah, and Chris Rackauckas 2021). MTK is a recent modeling and simulation framework to automate symbolic operations common for equation-based languages, such as methods for index reduction. It does so by using the symbolic-numerical capabilities of Julia to preprocess a description into a format that can be solved using the set of solvers provided by DifferentialEquations.jl. In other words, the translation process from an acausal description based on equations to a causal representation acceptable for a solver is similar to that of a typical Modelica Compiler.

Two studies (Ma, Gowda, Anantharaman, Laughman, V. Shah, and Chris Rackauckas 2021; Elmqvist, Neumayr, and Otter 2018) have begun to examine the usefulness of the Julia language within the context of equation-based

[3]URL: https://github.com/SciML/DiffEqDocs.jl/blob/5ab30570f91a69820e02449f71b59d2f786c3dd4/docs/src/index.md accessed 2022-04-27

modeling. However, no study has - to our knowledge - implemented and investigated the implications of developing a full compiler for an equation-based language in Julia. Instead, they provide the possibility of Modelica-like acausal modeling using Julia as a host language. In (Tinnerholm, Pop, Sjölund, Heuermann, and Abdelhak 2020) we presented the first Modelica compiler written in Julia. We constructed the compiler by translating the old frontend of the OMC and by providing a MetaModelica-Julia compatibility layer, *MetaModelica.jl*. In addition, a backend was implemented and the first backend target was DifferentialEquations.jl (Christopher Rackauckas and Nie 2017).

The main difference between Modelica and the language defined by MTK is the level of abstraction. To give an example, as of this writing, MTK requires users to specify the application of index reduction explicitly; it also requires systems to be specified explicitly with the state derivatives on the left-hand side. Thus, the user specifies the transformation from a DAE-System into an ODE-system, whereas in a Modelica compiler, these decisions are generally abstracted away. Still, as we illustrate, this flexibility makes MTK suitable as a backend framework for Modelica Compilers or other equation-based languages frameworks in Julia.

Modia.jl (Elmqvist and Otter 2017; Elmqvist, Otter, Neumayr, and Hippmann 2021) is another framework for acausal modeling in Julia. Syntactically it is more similar to Modelica when compared to the language defined by MTK. However, it is different from the work presented here because its constructs are implemented using Julia metaprogramming, primarily a set of macros, rather than traditional compiler phases.

Yet another modeling framework is *Causal.jl* (Sarı and Günel 2021) a causal modeling framework reminiscent of Simulink.

4.3 MetaModelica and MetaModelica.jl

MetaModelica is an extension to the Modelica language. It extends Modelica with several constructs to provide the Modelica language with greater expressive power (P. Fritzson, Pop, Abdelhak, Asghar, Bachmann, W. Braun, Bouskela, R. Braun, Buffoni, Casella, et al. 2020). To automatically translate the OpenModelica Compiler into Julia, MetaModelica constructs such as *matchcontinue* and *match* had to be mapped into Julia. To define these constructs we implemented MetaModelica.jl.

4.3.1 MetaModelica.jl

MetaModelica.jl[4] provides a compatibility layer between Julia and MetaModelica (P. Fritzson, Pop, and Sjölund 2011; Pop and P. Fritzson 2006) and an

[4] URL: https://github.com/OpenModelica/MetaModelica.jl accessed 2022-05-03.

Listing 4.2.2 Automatically generated Julia code for a simple hybrid system. The Julia code presented in this listing is targeting the IDA solver in Sundials (Hindmarsh, Brown, K. E. Grant, S. L. Lee, Serban, Shumaker, and Woodward 2005) using DifferentialEquations.jl.

```
function BouncingBallRealsStartConditions(aux, t)
  local x = zeros(2)
  local dx = zeros(2)
  local p = aux[1]
  local reals = aux[2]
  reals[1] = 1.0
  dx[1] = reals[2]
  dx[2] = -(p[2])
  x[2] = reals[2]
  x[1] = reals[1]
  return (x, dx)
end
function BouncingBallRealsDifferentialVars()
  return Bool[1, 1]
end
function BouncingBallRealsDAE_equations(res, dx, x, aux, t)
  local p = aux[1]
  local reals = aux[2]
  res[1] = dx[2] - -(p[2])
  res[2] = dx[1] - reals[2]
  reals[2] = x[2]
  reals[1] = x[1]
end

function BouncingBallRealsParameterVars()
  local aux = Array{Array{Float64}}(undef, 2)
  local p = Array{Float64}(undef, 2)
  local reals = Array{Float64}(undef, 2)
  aux[1] = p
  aux[2] = reals
  p[2] = 9.81
  p[1] = 0.7
  return aux
end
saved_values_BouncingBallReals = SavedValues(Float64, Tuple{Float64,Array})
function BouncingBallRealsCallbackSet(aux)
  local p = aux[1]
  function condition1(x, t, integrator)
    x[1] - 0.0
   end
  function affect1!(integrator)
    integrator.u[2] = -(p[1] * integrator.u[2])
  end
  cb1 = ContinuousCallback(condition1, affect1!, rootfind = true,
                           save_positions = (true, true), affect_neg! = affect1!)
  savingFunction(u, t, integrator) = let
    (t, deepcopy(integrator.p[2]))
  end
  cb2 = SavingCallback(savingFunction, saved_values_BouncingBallReals)
  return CallbackSet(cb1, cb2)
end
```

Listing 4.3.1 Original code written in MetaModelica to typecheck array expressions.

```
function matchArrayExpressions
  input output Expression exp1;
  input Type type1;
  input output Expression exp2;
  input Type type2;
  input Boolean allowUnknown;
  output Type compatibleType;
  output MatchKind matchKind;
protected
  Type ety1, ety2;
  list<Dimension> dims1, dims2;
algorithm
  Type.ARRAY(elementType = ety1, dimensions = dims1) := type1;
  Type.ARRAY(elementType = ety2, dimensions = dims2) := type2;
  // Check that the element types are compatible.
  (exp1, exp2, compatibleType, matchKind) :=
    matchExpressions(exp1, ety1, exp2, ety2, allowUnknown);
  // If the element types are compatible, check the dimensions too.
  (compatibleType, matchKind) :=
    matchArrayDims(dims1, dims2, compatibleType, matchKind, allowUnknown);
end matchArrayExpressions;
```

extension to the Julia language via Julia metaprogramming. It re-implements several constructs of MetaModelica such as *match* and *matchcontinue*. Furthermore, MetaModelica.jl replicates the existing runtime of OMC. This Julia extension is used extensively in the translated modules.

For a more-in-depth comparison between Julia and MetaModelica we refer to (P. Fritzson, Pop, Sjölund, and Asghar 2019).

Listing 4.3.2 illustrates an example of automatically translated code using *@match* equations. The original MetaModelica version of this function can be seen in Listing 4.3.1.

4.4 OMFrontend

OMFrontend.jl is used to flatten Modelica code. OMFrontend was automatically generated from the high-performance frontend (Pop, Östlund, Casella, Sjölund, Franke, et al. 2019) of the OMC. Previously, we used the old frontend (Tinnerholm, Pop, Sjölund, Heuermann, and Abdelhak 2020); however, as part of the work presented here, the *MetaModelica-Julia translator* was used to automatically generate a Julia implementation of the high-performance frontend (Pop, Östlund, Casella, Sjölund, Franke, et al. 2019).

While the translation of the old frontend[5] was achieved without any major modifications, we had to manually resolve cases of mutually circular module

[5] The old frontend is the frontend the *high-performance frontend* replaced (Pop, Östlund, Casella, Sjölund, Franke, et al. 2019).

Listing 4.3.2 A function used in the type checking phase of our Modelica compiler. This code make use of the @match equation constructs from MetaModelica, for comparison the original MetaModelica code is available in Listing 4.3.1.

```
function matchArrayExpressions(
  exp1::Expression,
  type1::NFType,
  exp2::Expression,
  type2::NFType,
  allowUnknown::Bool,
  )::Tuple{Expression, Expression, NFType, MatchKindType}
  local matchKind::MatchKindType
  local compatibleType::NFType
  local ety1::NFType
  local ety2::NFType
  local dims1::List{Dimension}
  local dims2::List{Dimension}
  @match TYPE_ARRAY(elementType = ety1, dimensions = dims1) = type1
  @match TYPE_ARRAY(elementType = ety2, dimensions = dims2) = type2
  #= Check that the element types are compatible. =#
  (exp1, exp2, compatibleType, matchKind) =
  matchExpressions(exp1, ety1, exp2, ety2, allowUnknown)
  #= If the element types are compatible, check the dimensions too. =#
 (compatibleType, matchKind) =
  matchArrayDims(dims1, dims2, compatibleType, matchKind, allowUnknown)
  return (exp1, exp2, compatibleType, matchKind)
end
```

dependencies for the new frontend since Julia does not handle mutually circular module dependencies while MetaModelica does.

The design and implementation of this frontend remains the same as described by Pop et al. in (Pop, Östlund, Casella, Sjölund, Franke, et al. 2019), however a few modifications where made that we elaborate on later in this text.

4.4.1 Validating the frontend by using Flat-Modelica

Similarly to the frontend in OMC *OpenModelica.jl* is capable of generating flat Modelica from a Modelica model. By generating flat Modelica using OMC and comparing it to OMFrontend, we established the frontend's correctness to an extent[6]. In Listing 4.4.1 we can see a model representing a water tank, and in Listing 4.4.2 we can see the corresponding flat model generated by OMFrontend where the object-orientation is gone.

[6]Software testing can not prove the absence of errors.

Listing 4.4.1 A model of a water tank.

```
connector Stream   //Connector class
  Real pressure;
  flow Real volumeFlowRate;
end Stream;

model Tank
  parameter Real area = 1;
  replaceable connector TankStream = Stream;
  TankStream inlet, outlet;
  Real level(start=2);
equation
  inlet.volumeFlowRate = 1;
  inlet.pressure = 1;
  area * der(level) = inlet.volumeFlowRate + outlet.volumeFlowRate;
  outlet.pressure = inlet.pressure;
  outlet.volumeFlowRate = 2;
end Tank;
```

Listing 4.4.2 The flat model of the water tank.

```
class Tank
  parameter Real area = 1.0;
  Real inlet.pressure;
  flow Real inlet.volumeFlowRate;
  Real outlet.pressure;
  flow Real outlet.volumeFlowRate;
  Real level(start = 2.0);
equation
  inlet.volumeFlowRate = 0.0;
  outlet.volumeFlowRate = 0.0;
  inlet.volumeFlowRate = 1.0;
  inlet.pressure = 1.0;
  area * der(level) = inlet.volumeFlowRate + outlet.volumeFlowRate;
  outlet.pressure = inlet.pressure;
  outlet.volumeFlowRate = 2.0;
end Tank;
```

4.4.2 Modelica library support

The compiler presented in this text provide support for users to write their own custom libraries. We have also tested our compiler on existing Modelica libraries, such as the ScalableTestSuite testsuite and the MSL. While OpenModelica.jl does not currently cover all models of the Modelica Standard Library, the Electrical sub-library of electrical components and example models is currently supported. In Chapter 5 we provide one example where we use a transmission line model from the ScalableTestSuite (Casella 2015) to estimate the current performance of our frontend.

4.5 OMBackend

The module responsible for code generation is the backend module, *OMBackend.jl*. Current backend targets include both MTK and DifferentialEquations.jl. The DifferentialEquations.jl backend uses the Sundials IDA solver (Hindmarsh, Brown, K. E. Grant, S. L. Lee, Serban, Shumaker, and Woodward 2005) and it roughly follows the DAE-mode implementation by (W. Braun, Casella, Bachmann, et al. 2017). OMBackend currently supports continuous systems and initial support for hybrid systems. The backend currently performs matching and sorting on the equations; however, the process of symbolic index reduction and other compiler optimizations such as algebraic simplification is outsourced to the MTK-framework. Furthermore, the backend integrates other Julia facilities such as *Plots.jl* (Christ, Schwabeneder, and Christopher Rackauckas 2022) for plotting and animation.

4.6 Extending the Modelica language to support Variable Structured Systems

The literature overview presented in Chapter 3 highlighted that for an equation-based language to support VSS, the language would need to have the syntax and semantics necessary to describe and capture structural changes in the systems of equations. Furthermore, if the system is object oriented, the language also needs such syntax and semantics to capture the change in the components constituting these systems as well. In this section we discuss two extensions to enable the Modelica language to represent such systems.

4.6.1 Explicit Variable Structured Systems

We define *Explicit Variable Structured Systems* as system where the transitions between states of the system are explicitly encoded by the *modeler*. In this case the equations and variables of the system is known before the system is simulated. To illustrate this we reused the process of representing state machines in the Modelica language by providing support for *continuous state machines*.

However, while state machines in Modelica does not support continuous-time equations or algorithms[7] our representations allows a modeler to represent structural transitions between separate continuous-time states.

To be able to encode such explicit structural transitions we introduced one new keyword *structuralmode* along with two operators:

- initialStructuralState(state)
- structuralTransition(fromState, toState, condition)

[7]URL: https://specification.modelica.org/v3.4/Ch17.html accessed 2022-04-20

Listing 4.6.1 An example of a simple explicit variable structured systems with two modes of operation.

```
model SimpleTwoModes
  model Single
    parameter Real a = 1.0;
    Real x (start = 1.0);
  equation
    der(x) = 2 * x + a;
  end Single;
  model HybridSingle
    parameter Real a = 1.0;
    Real x (start = 0.0);
  equation
    der(x) = x  - a;
  end HybridSingle;
  structuralmode Single firstMode;
  structuralmode HybridSingle secondMode;
equation
  // We start in this initial mode
  initialStructuralState(firstMode);
  // We switch the mode when time is greater than or equal to 0.7
  structuralTransition(firstMode, secondMode, time  >=  0.7);
end SimpleTwoModes;
```

The operator *initialStructuralState* is used to represent an initial structural state while *structuralTransition* is used to specify the transition between one structural state to another structural state.

Listing 4.6.1 illustrates an example of a system modeled using these constructs. The model *SimpleTwoModes* consists of two states *Single* and *HybridSingle*. The model starts in the *Single* state and after 0.7 second the model transition to the next state *HybridSingle*. This transition is modeled using the *structuralTransition* operator and the initial structural state is specified using the *initialStructuralState* operator.

The code for simulating and plotting this model is available in Listing 4.6.1, the plot is available in Figure 4.3.

4.6.1.1 Modeling the breaking pendulum explicitly

Using these constructs we can simulate models where the equations and variables change, given that this change is encoded by the modeler. We can use this methodology of explicitly encoding the states to model the breaking pendulum model discussed in Chapter 3.

Simulating this system results in the plot seen in Figure 4.4. To summarize, using an explicit approach we can increase the flexibility concerning what is possible to express in Modelica. However, there are some disadvantages to this approach. The first is that the representation is causal, that is the

Listing 4.6.2 A program to simulate and plot *SimpleTwoModes* from Listing 4.6.1 using OpenModelica.jl with associated modules.

```
using Revise
import Absyn
import DAE
import OM
import OMBackend
import OMFrontend
import SCode
using MetaModelica
using Plots
function runModelMTK(model,
                     file;
                     timeSpan = (0.0, 1.0))
  @info "Running : " model
  @time OM.simulate(model,
                    file,
                    mode = OMBackend.MTK_MODE,
                    startTime = first(timeSpan),
                    stopTime = last(timeSpan))
end
res = runModelMTK("SimpleTwoModes",
                  "./Models/SimpleTwoModes.mo";
                  timeSpan=(0.0, 1.0))
p = plot(res; legend = :topleft)
Plots.pdf(p,
          "./Plots/SimpleTwoModesPlot")
```

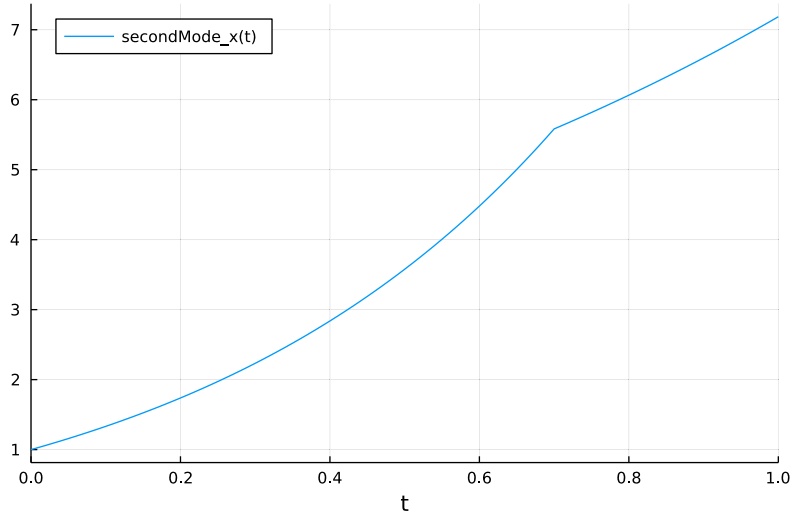

Figure 4.3: The result of simulating Listing 4.6.1.

Listing 4.6.3 An example of the breaking pendulum model using structural transitions.

```
model FreeFall
  Real x;
  Real y;
  Real vx;
  Real vy;
  parameter Real g = 9.81;
  parameter Real vx0 = 0.0;
equation
  der(x) = vx;
  der(y) = vy;
  der(vx) = vx0;
  der(vy) = -g;
end FreeFall;
model Pendulum
  parameter Real x0 = 10;
  parameter Real y0 = 10;
  parameter Real g = 9.81;
  parameter Real L = sqrt(x0^2 + y0^2);
  // Common variables
  Real x(start = x0);
  Real y(start = y0);
  Real vx;
  Real vy;
  // Model specific variables
  Real phi(start = 1.0, fixed = true);
  Real phid;
equation
  der(phi) = phid;
  der(x) = vx;
  der(y) = vy;
  x = L * sin(phi);
  y = -L * cos(phi);
  der(phid) =  -g / L * sin(phi);
end Pendulum;

model BreakingPendulum
  structuralmode Pendulum pendulum;
  structuralmode FreeFall freeFall;
equation
  initialStructuralState(pendulum);
  structuralTransition(pendulum,
                       freeFall,
                       time - 5.0 <= 0);
end BreakingPendulum;
```

Listing 4.6.4 A program to simulate and plot the explicit breaking pendulum model.

```
using Revise
import Absyn
import DAE
import OM
import OMBackend
import OMFrontend
import SCode
using MetaModelica
using Plots

function runModelMTK(model, file; timeSpan = (0.0, 1.0))
  @info "Running : " model
  @time OM.runModelFM(model,
                      file,
                      mode = OMBackend.MTK_MODE,
                      startTime = first(timeSpan),
                      stopTime = last(timeSpan))
end

res = runModelMTK("BreakingPendulum",
                  "./Models/BreakingPendulumExplicit.mo";
                  timeSpan=(0.0, 7.0))
p = plot(res;
         legend = :bottomleft,
         xlim=(0.0, 7.0),
         ylim = (0.0, 10.0),
         vars = [(0,2)])
Plots.pdf(p, "./Plots/BreakingPendulumExplicit")
```

transition between the states need to be encoded sequentially. The second drawback is that all equations need to be known and represented a priori simulation. The disadvantage of this representation is that the compiler and the simulation runtime need to process the entire model at the same time, and while transition between states can be achieved dynamically the model may not modify itself during simulation. The implicit VSS discussed in the next section does not have this disadvantage. Consequently we do not need to explicitly encode the model's entire behavior.

4.6.2 Implicit Variable Structured Systems

In the previous section we discussed systems that we denoted *Explicit Variable Structure Systems*. These are models where the variables and equations change during simulation according to some explicit stated scheme. In this section we provide examples of implicit systems where we lift the restriction of explicit encoding. We do so by introducing a single new operator *recompilation*. To achieve recompilation during simulation we introduce JIT-

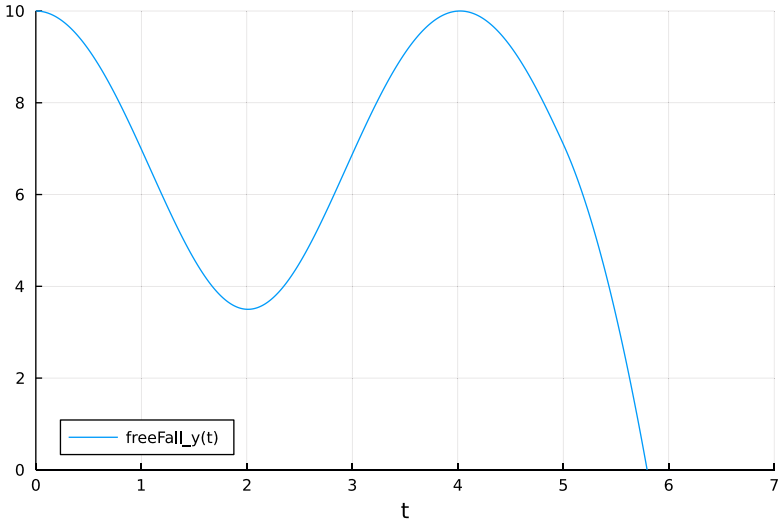

Figure 4.4: The result of simulating Listing 4.6.3.

Compilation in our compiler. Figure 4.5 illustrates compilation our compiler extended with dynamic capabilities.

Recompilation allows structural events to trigger a modification and subsequently recompile the model under simulation. To achieve this we extended the flat Modelica representation to also contain a *MetaModel* of itself[8]. During recompilation the model may query itself and change the values of its parameters or constants. In this way the different sets of equations and variables need not to be explicitly encoded before structural transitions. Consequently, values computed by some model during simulation may be used to modify the model itself. Note that any part of the model could be changed if additional meta-programming operators would be introduced or even load a completely different model during recompilation.

To illustrate consider the two examples in Listing 4.6.6 and Listing 4.6.5 respectively. At the start of the simulation *ArrayShrink* consists of ten equations and variables. However, after 0.5 seconds the system changes radically, and the number of equations and variables shrinks to five. For the second example in Listing 4.6.5 the system initially consists of 10 equations however, during the course of the simulation the set of equations and variables double.

The code and the resulting plot of this system is presented in Listing 4.6.7. The benefit of this approach is that it is also capable of modeling the explicit models discussed previously. Using the *recompilation* construct we can modify

[8] As of this writing this meta model consists of the SCode IR.

Listing 4.6.5 The ArrayGrow model.

```
// This is an example of a model with structural variability
// We initially start with 10 equations, however during the simulation
// the amount of equations are doubled.
model ArrayGrow
   parameter Integer N = 10;
   Real x[N](start = {i for i in 1:N});
equation
  for i in 1:N loop
    x[i] = der(x[i]);
  end for;
  when time > 0.5 then
    // Recompilation with change of parameters.
    // the name of this function is the subject of change.
    // What is changed depends on the argument passed to this function.
    recompilation(
     N /* What we are changing */,
     2 * N /* The value of the change */
    );
  end when;
end ArrayGrow;
```

Listing 4.6.6 The ArrayShrink model.

```
// This is an example of a model with structural variability
// We initially start with 10 equations, however during the simulation
// the amount of equations are decreased to 5.
model ArrayShrink
   parameter Integer N = 10;
   Real x[N](start = {i for i in 1:N});
equation
  for i in 1:N loop
    x[i] = der(x[i]);
  end for;
  when time > 0.5 then
    // Recompilation with change of parameters.
    // the name of this function is the subject of change.
    // What is changed depends on the argument passed to this function.
    recompilation(
      N /* What we are changing */,
      5 /* The value of the change */
    );
  end when;
end ArrayShrink;
```

Listing 4.6.7 Program to simulate the implicit breaking pendulum model, *ArrayGrow* and *ArrayShrink* models.

```
using Revise

import Absyn
import DAE
import OM
import OMBackend
import OMFrontend
import SCode
using DifferentialEquations
using MetaModelica
using Plots

function runModelMTK(model, file; timeSpan = (0.0, 1.0))
  @info "Running : " model
  @time OM.simulate(model, file,
                    mode = OMBackend.MTK_MODE,
                    startTime = first(timeSpan),
                    stopTime = last(timeSpan),
                    solver = :(Rodas5()))
end

function plotCombined(res, name, limX, limY)
  #= Plot array grow=#
  p1 = plot(res[1]; legend = false)
  p2 = plot(res[2]; legend = false)
  p3 = plot(p1, p2)
  #= Plot array grow change from 10 to 15 equations =#
  Plots.pdf(p3, "./Plots/$name")
  #= Construct a merged plot =#
  p1 = plot(res[1]; legend = false, xlim=limX, ylim = limY)
  p2 = plot!(res[2]; legend = false, xlim=limX, ylim = limY)
  Plots.pdf(p2, "./Plots/$(name)SinglePlot")
end

function plotPendulum(res, name, limX, limY)
  #= Plot array grow=#
  p1 = plot(res[1]; legend = true)
  p2 = plot(res[2]; legend = true)
  p3 = plot(p1, p2)
  #= Plot array grow change from 10 to 15 equations =#
  Plots.pdf(p3, "./Plots/$name")
  #= Construct a merged plot =#
  p1 = plot(res[1]; legend = :bottomleft, xlim=limX, ylim = limY, vars = [(0,3)])
  p2 = plot!(res[2]; legend = :bottomleft, xlim=limX, ylim = limY, vars = [(0,2)])
  Plots.pdf(p2, "./Plots/$(name)SinglePlot")
end

res = runModelMTK("BreakingPendulum",
                  "./Models/BreakingPendulumRecompilation.mo"; timeSpan=(0.0, 7.0))
plotPendulum(res, "BreakingPendulum", (0.0, 7.0), (-10, 10.0))

res = runModelMTK("ArrayGrow", "./Models/ArrayGrow.mo"; timeSpan=(0.0, 1.0))
plotCombined(res, "ArrayGrow", (0.0, 1.0), (0.0, 20))
res = runModelMTK("ArrayShrink", "./Models/ArrayShrink.mo"; timeSpan=(0.0, 1.0))
plotCombined(res, "ArrayShrink", (0.0, 1.0), (0.0, 20))
```

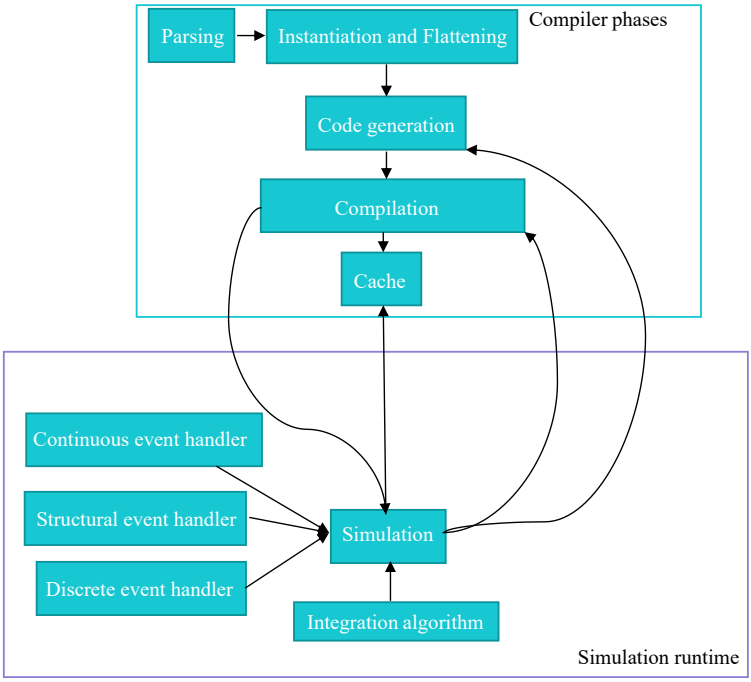

Figure 4.5: The compilation and simulation process of a Modelica compiler with dynamic capabilities. A dynamic Modelica compiler differs from a static compiler in the sense that the line between system simulation and compilation are blurred. Instead such a framework unifies the simulation process and the translation phase of a system being simulated.

such parameters during simulation by querying and updating the meta model. With this proposed extension we can reformulate this model see, Listing 4.6.8.

4.7 Summary

In this chapter we have presented OpenModelica.jl a Modelica environment in Julia and discussed some of the implications of having such a framework written in this language. We have demonstrated that:

- Automatically translating MetaModelica to Julia is possible
- A Modelica compiler written in the Julia language is possible
- The Modelica language can be extended to simulate VSS with only minor changes

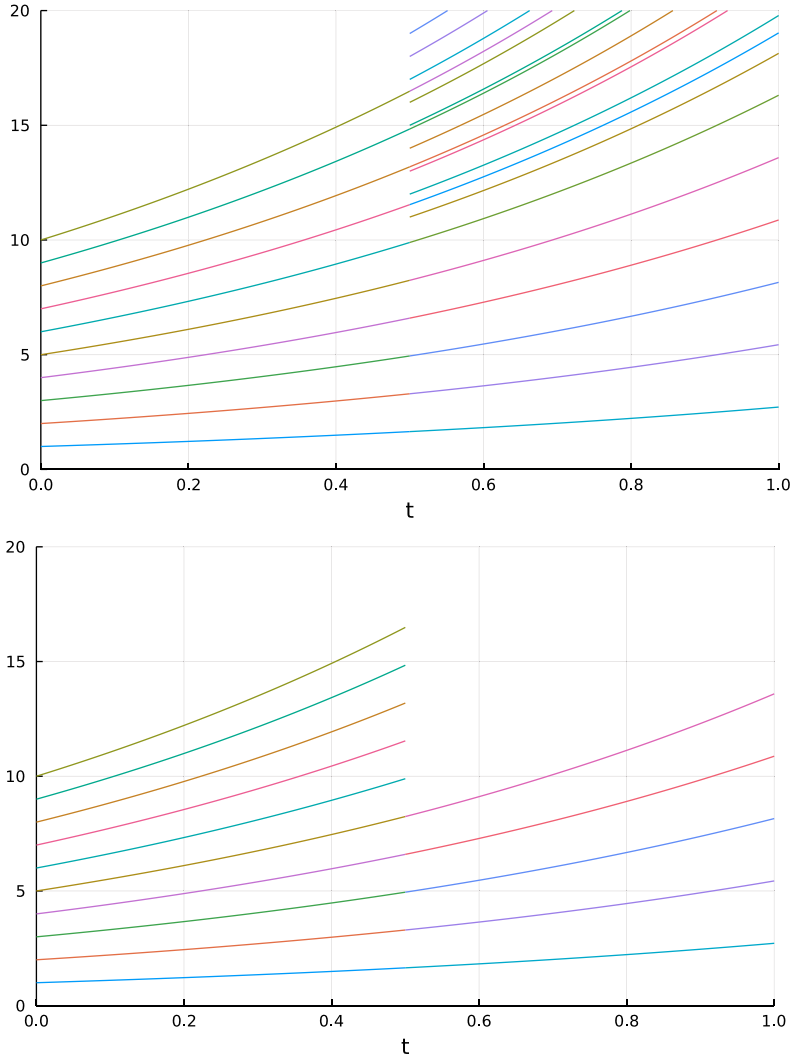

Figure 4.6: The result of simulating Listing 4.6.5 (Top) and Listing 4.6.6 (Bottom). The structural change and subsequent recompilation of the model occurs at $t = 0.5$ seconds. The curve in the graph represents all x_i variables of $\underrightarrow{x}$ in Listing 4.6.5 and Listing 4.6.6 respectively. In the top graph we can see how the system grows and in the bottom graph we can see how the system shrinks at $t = 5$.

Listing 4.6.8 The breaking pendulum model using the new recompilation keyword to activate and deactivate components via Just-in-time compilation during simulation.

```
model BreakingPendulum

model FreeFall
  parameter Real e=0.7;
  parameter Real g=9.81;
  Real x;
  Real y;
  Real vx;
  Real vy;
equation
  der(x) = vx;
  der(y) = vy;
  der(vy) = -g;
  der(vx) = 0.0;
end FreeFall;

model Pendulum
  parameter Real x0 = 10;
  parameter Real y0 = 10;
  parameter Real g = 9.81;
  parameter Real L = sqrt(x0^2 + y0^2);
  // Common variables
  Real x(start = x0);
  Real y(start = y0);
  Real vx;
  Real vy;
  // Model specific variables
  Real phi(start = 1., fixed = true);
  Real phid;
equation
  der(phi) = phid;
  der(x) = vx;
  der(y) = vy;
  x = L * sin(phi);
  y = -L * cos(phi);
  der(phid) =  -g / L * sin(phi);
end Pendulum;

  parameter Boolean breaks = false;
  FreeFall freeFall if breaks;
  Pendulum pendulum if not breaks;
equation
  when 5.0 <= time then
    recompilation(breaks, true);
  end when;
end BreakingPendulum;
```

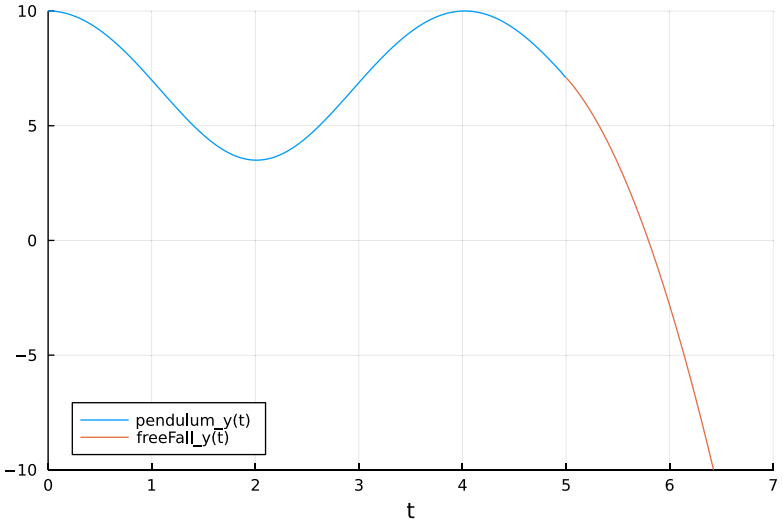

Figure 4.7: The result of simulating Listing 4.6.3. Note that the legend provides the prefix of the last active model instance which was the $freeFall$. The behavior before the structural transition at $t = 5$ is described by the equations of the Pendulum model with instance *pendulum*.

- There is an advantage in terms of expressive power of a modeling language when blurring the line between compilation, modeling and simulation

At the time of writing, to my best knowledge, there is no other framework for object-oriented equation-based languages that is capable of simulating systems with structural change consisting of thousands of variables and equations where JIT-Compilation is the main technique. Furthermore, it can be argued, that our proposed framework is the first composable implementation of a compiler for the Modelica language.

5. Results

This chapter is closely based upon:

- OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl (Tinnerholm, Pop, Heuermann, and Sjölund 2021)
- A modular, extensible, and Modelica standard-compliant OpenModelica compiler framework in Julia supporting structural variability, J Tinnerholm, A Pop, M Sjölund (Submitted for publication)

The previous chapter presented a novel framework for equation-based simulation and modeling. In this chapter, we evaluate the performance of this framework. The instrumentation of our experiments is given in Section 5.1.

In Section 5.2 we analyze the simulation performance of OpenModelica.jl and compare it to the OpenModelica environment (P. Fritzson, Pop, Abdelhak, Asghar, Bachmann, W. Braun, Bouskela, R. Braun, Buffoni, Casella, et al. 2020).

This is followed by Section 5.3 where we present some numbers concerning initial compilation time performance when compiling a set of Modelica models.

Recall that in Section 4.6 we explained how we extended the Modelica language to allow the simulation of systems with variable structure via JIT-Compilation. To examine the overhead and possible advantages of this method, we compare the costs induced by this method in Section 5.4. In Section 5.5 we compare our solution to the related research presented in Section 3.3. Finally, we end the chapter in Section 5.6 where we summarize our results.

5.1 Instrumentation

The experiments were run with the following system specifications:

Table 5.1: Hardware used in the performance experiments.

Operating System	Processor	System memory
Ubuntu 20.04.4 LTS	AMD Ryzen[1]	130 GiB

Table 5.2: Software packages used.

OpenModelica	ModelingToolkit	Julia
1.18.1	ModelingToolkit v8.5.0	1.7.2

5.2 Simulation of large Modelica models

This experiment evaluates the simulation time performance when simulating large Modelica models using our proposed compiler. The model selected for this experiment is the CascadingFirstOrder system from the scalable testsuite (Casella 2015), see Listing 5.2.1.

Listing 5.2.1 The Cascading first Order system from the scalable testsuite.

```
package CascadingFirstOrder
  model Casc
    parameter Integer N = 100 "Order of the system";
    final parameter Real tau = T/N "Individual time constant";
    parameter Real T = 1 "System delay";
    Real x[N] (each start = 0, each fixed = true);
  equation
    tau*der(x[1]) = 1 - x[1];
    for i in 2:N loop
      tau*der(x[i]) = x[i-1] - x[i];
    end for;
  end Casc;

  model Casc10
    Casc(N = 10);
  end Casc10;

  model Casc100
    Casc(N = 100);
  end Casc100;

  model Casc200
    Casc(N = 200);
  end Casc200;

  model Casc400
    Casc(N = 400);
  end Casc400;

  model Casc800
    Casc(N = 800);
  end Casc800;
...
end CascadingFirstOrder;
```

In our experiment, we gradually increased N[2] from 10 to 25600 and simulated the system using the MTK backend of our proposed compiler with the TSIT5 solver and the IDA solver. The resulting simulation time was evaluated using the standard benchmark suite of Julia, BenchmarkTools.jl (Chen and Revels 2016). We also performed the same experiments using the OMC compiler with the IDA solver. The IDA solver was selected since the OMC did not support a solver similar to TSIT5. The benchmarking program was set to use 1000 samples for each level of N. The timeout over all samples for each N was set to 500 seconds.

The OMC was used with the standard settings and the IDA solver.

The resulting simulation time performance is presented in Table B.1. From this experiment, we can see that the simulation time performance of our proposed compiler is on par with one state-of-the-art Modelica compiler. Furthermore, since the MTK environment supports more solvers compared to the OMC we can also leverage this difference and achieve better performance than the OMC. The feasibility of the MTK framework has also been demonstrated in (Chris Rackauckas, Anantharaman, Edelman, Gowda, Gwozdz, Jain, Laughman, Ma, Martinuzzi, Pal, Rajput, Saba, and V. Shah 2021) where MTK models accelerated with machine learning outperformed the commercial Dymola compiler in terms of simulation performance in a specific case. However, due to the high memory requirements of Julia and MTK, we are currently unable to go further than 25600 equations in this benchmark.

If we examine Figure 5.1 we conclude that simulation time performance is similar between OpenModelica and OpenModelica.jl when simulating the system with around a thousand equations and variables. In Table B.1 we see that OpenModelica.jl initially performs better. However, as the value of N is increased, the performance between the two becomes more similar, as illustrated in Figure 5.1. To conclude, the experiment highlights that the MTK can achieve similar performance to that of an existing state-of-the-art Modelica compiler.

It should be noted, however, that both MTK and subsequently DifferentialEquations.jl can be configured using several solvers and employ parallel processing to solve systems faster. Similar configurations can also be employed by the OMC. This experiment aimed not to test all possible permutations of backend configurations but rather to examine the behavior of the new Modelica compiler presented here and one existing state-of-the-art compiler using typical configurations in both environments.

[2] N in Listing 5.2.1 directly governs the number of equations and variables in the model.

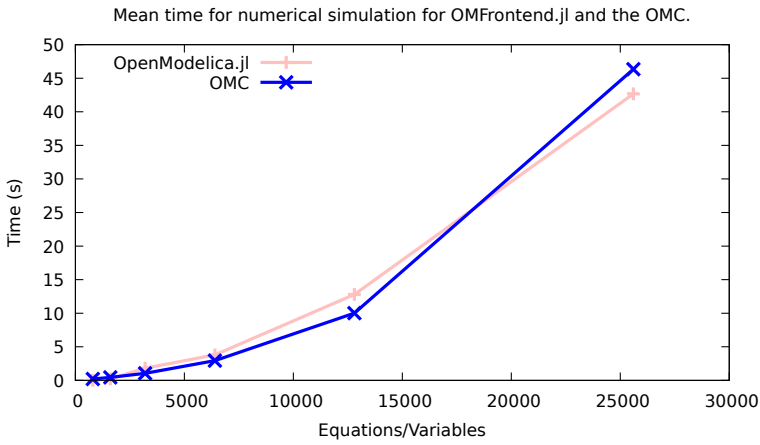

Figure 5.1: Time spent during numerical simulation in OMFrontend.jl and for the OMC. Lower is better.

5.3 Evaluating compile-time overhead

This section presents the compile-time overhead when flattening large Modelica models. For this experiment, we selected the Transmission line model from the scalable testsuite, since it represents a typical Modelica model and uses the Modelica Standard Library. The full model is presented in Listing 5.3.1.

We generated scalarized flat Modelica code by gradually increasing the N in the transmission line model, starting with $N = 10$ and ending with $N = 1280$. For each N we ran the experiments for 500 seconds with the maximum number of replications configured to 100. The compilation time is presented in Tables B.3 and B.5 and memory consumption is presented in Table B.4. Figure 5.2 illustrates how the two frameworks compare.

In this experiment, we have demonstrated that although OpenModelica.jl is still under development, we can see that the partially automatically translated frontend is capable of translating large Modelica models up to 26915 equations and variables using standard Modelica components from the MSL with competitive performance compared to the OpenModelica Compiler. Overall, these results indicate that while the frontend demonstrates lower performance compared to the OMC by a factor of 3.7 in the case of 26915 equations and variables. It can be argued that frontend performance is not significantly worse, especially considering *OpenModelica.jl* is still in its development phase.

These results must also be understood in that the current frontend is mostly automatically generated from the OMC so performance regressions were expected. Still, as discussed in (Tinnerholm, Sjölund, and Pop 2019), the Julia language is superior in terms of performance for certain cases in comparison to the MetaModelica language. The Julia language is also actively developed and improved by a large team while the MetaModelica language is in maintenance phase only.

We expect that with manual tuning *OMFrontend.jl* can achieve similar performance or even outperform the OMC. However, additional research is needed to establish the benefits of using Julia to implement a compiler for an equation-based language. In the next section, we highlight these issues in more detail, in an experiment where we measure the impact of JIT-Compilation when simulating a system with variable structure.

Listing 5.3.1 A Modelica model representing an electrical transmission line.

```
// Transmission line model from the Scalable testsuite by Francesco Casella Politecnico
↪  Milano
model TransmissionLine "Modular model of an electrical transmission line"
  import Modelica.SIunits;
  import Modelica.Electrical.Analog;
  SIunits.Voltage vpg "voltage of pin p of the transmission line";
  SIunits.Voltage vng "voltage of pin n of the transmission line";
  SIunits.Current ipin_p
    "current flows through pin p of the transmission line";
  SIunits.Current ipin_n
    "current flows through pin n of the transmission line";
  Analog.Interfaces.Pin pin_p;
  Analog.Interfaces.Pin pin_n;
  Analog.Interfaces.Pin pin_ground "pin of the ground";
  Analog.Basic.Ground ground "ground of the transmission line";
  parameter Integer N = 1 "number of segments";
  parameter Real r "resistance per meter";
  parameter Real l "inductance per meter";
  parameter Real c "capacitance per meter";
  parameter Real length "length of tranmission line";
  Analog.Basic.Inductor L[N](L = fill(l * length / N, N)) "N inductors";
  Analog.Basic.Capacitor C[N](C = fill(c * length / N, N)) "N capacitors";
  Analog.Basic.Resistor R[N](R = fill(r * length / N, N)) "N resistors";
initial equation
  for i in 1:N loop
    C[i].v = 0;
    L[i].i = 0;
  end for;
equation
  vpg = pin_p.v - pin_ground.v;
  vng = pin_n.v - pin_ground.v;
  ipin_p = pin_p.i;
  ipin_n = pin_n.i;
  connect(pin_p, R[1].p);
  for i in 1:N loop
    connect(R[i].n, L[i].p);
    connect(C[i].p, L[i].n);
    connect(C[i].n, pin_ground);
  end for;
  for i in 1:N - 1 loop
    connect(L[i].n, R[i + 1].p);
  end for;
  connect(L[N].n, pin_n);
  connect(pin_ground, ground.p);
end TransmissionLine;
```

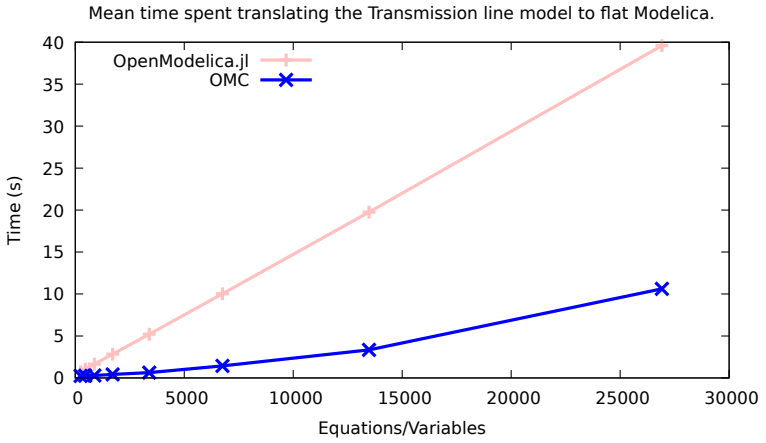

Figure 5.2: Time spent translating the Transmission line model in Listing 5.3.1 to flat Modelica for OMFrontend.jl and for the OMC. Lower is better.

Listing 5.4.1 SimpleClockArrayGrow, this model initially starts out with N equations and variables, however, each 10 seconds the structure of the model changes and K new equations and variables are added to the system.

```
// This model a exhibits the same behavior as
// ArrayGrow, except that it resizes several times
model SimpleClockArrayGrow
  parameter Integer N = 1000;
  parameter Integer K = 2000;
  Real x[N](start = {i for i in 1:N});
equation
  when sample(0.0, 15.0) then
    recompilation(N, N + K);
  end when;
  for i in 1:N loop
    x[i] = der(x[i]);
  end for;
end SimpleClockArrayGrow;
```

5.4 Evaluating the cost of structural changes

When introducing JIT-Compilation in a Modelica compiler it is important to examine the introduced overhead. In this section we evaluate the cost of compilation during simulation using a modified variant of the previously described *ArrayGrow* model (see Listing 4.6.5), *SimpleClockArrayGrow* see Listing 5.4.1.

This model differs from the previous *ArrayGrow* in that it gradually grows the system of equations during simulation instead of just once. A more realistic model with a similar behavior could be a nuclear power plant where different reactors are scheduled to be active at specific times or some other system where the dynamics change abruptly at regular intervals.

In this experiment we simulate *SimpleClockArrayGrow* for 60 seconds. Initially, the model consists of 1000 equations and variables, but after 15 seconds, the structure of the system changes, and the number of equations and variables increases to 2000. This process is repeated continuously until the system reaches 7000 equations.

Table 5.3 presents the median time in seconds required by each phase in the compiler when processing the structural changes in the model, compared to the cost of numerical simulation.

The median value was computed by running the model 5 times. The solver used in this experiment was Rodas5, a Rosenbrock method for stiff problems, with the tolerance set to $1e-6$. This solver was selected to emulate computationally expensive simulation.

The reason for not using a standardized benchmark suite for this example was that it was not possible to configure *BenchmarkTools.jl* with the granularity necessary to estimate the cost of the various phases. If we examine the

Table 5.3: The total time in seconds between the different phases of simulating the system with variable structure presented in Listing 5.4.1. Note that the reported compilation stages when $N = 1000$ is the initial time of compiling the model, that is the simulation in the initial interval between 0.0 seconds and 15.0 seconds. Since only five replications were used, the numbers in this graph were derived from the median.

Equations and Variables	1000	3000	5000	7000
Frontend Processing	0.21 s	2.65 s	2.86 s	3.24
Backend Processing	0.3	2.16	6.12	13.7
Machine Code Generation	3.62	9.06	16.4	25.4
Numerical simulation	8.81	52.02	153.44	335

data in Table 5.3 we can see that most of the significant cost of recompiling during simulation is caused by the Julia compiler and the later machine code generation done by LLVM. A separate sequence of experiments was used to establish the initial compilation time before the first simulation.

In Figure 5.3 we can see that the total compilation time is only a fraction of the total time spent when simulating this model. Furthermore, we can see that process of translating Modelica to Julia code is only a small fraction of the total compilation time. The main bottleneck is compile time machine code generation to LLVM by Julia. Comparing the results, it can be seen that the feasibility of runtime compilation depends on how often the system undergoes structural changes.

From this experiment, it is clear that a system undergoing such changes every other time-step would suffer from extensive overhead caused by excessive recompilation. This issue, however, could be circumvented by relying on interpretation instead of machine code generation. Hence, simulation time performance can be improved by combining the two approaches, generate machine code for large systems with few structural changes, and use interpretation for smaller systems. However, heuristics need to be developed to decide when to generate machine code and when to interpret the system under simulation.

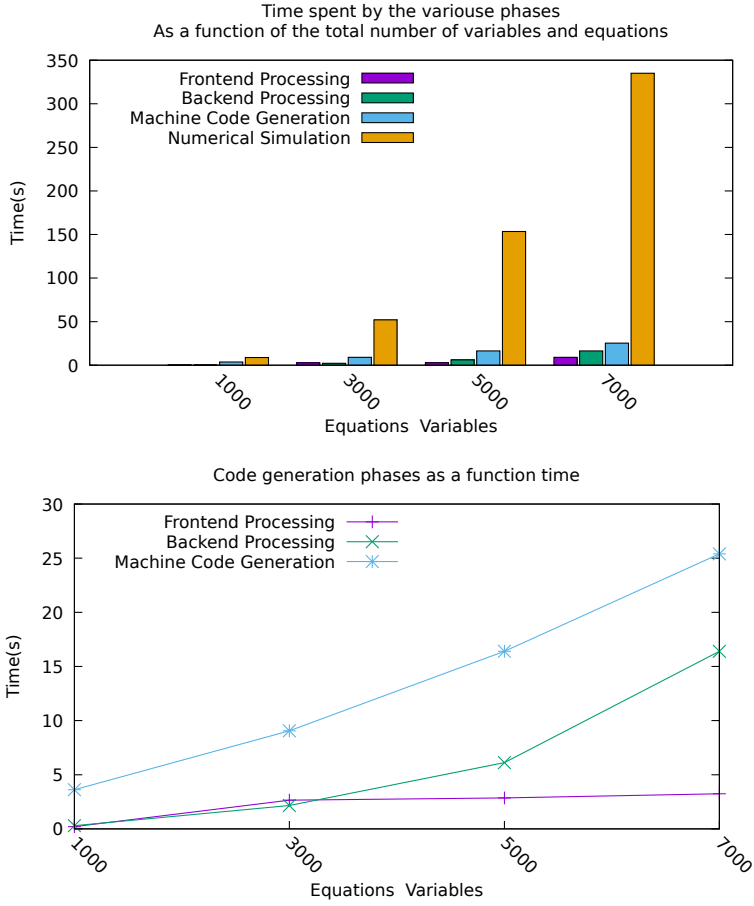

Figure 5.3: Time spent (in seconds) in the various phases when simulating the model in Listing 5.4.1.

5.5 Comparison To Related work

As discussed in Chapter 3 there are other frameworks capable of handling Modelica systems with variable structure. However, none of these frameworks can handle standard Modelica to the same extent, with the same size and scale.

In Table 5.4 we compare some characteristics of our proposed framework with other existing frameworks. Our extensions to Modelica (in Table 5.4 called *VariableModelica*) can be viewed as a combination of *Sol* and *Hydra*. Still, neither *Sol* nor *Hydra* are capable of handling standard Modelica. In this sense, the extension presented here is more similar to the work presented by Höger (2019). However, since our extension relies upon JIT-Compilation it is not necessary to set explicit checkpoints to mark where structural changes occur. Furthermore, while the frontend presented by Höger (2019) supports a subset of Modelica, *Nano Modelica* our solution supports much more of the Modelica standard[3].

On the technical side, our work is based upon MTK. Hence, we have access to more solvers combined with the symbolic numerical capabilities of Julia. Furthermore, our proposed compiler is not limited to a single platform. It works on Windows, macOS, and Linux.

[3]However, in theory the work presented by Höger (2019) could be extended with such support.

Table 5.4: Characteristics of languages and frameworks that are able to express system with structural variability. Our extension supported by OpenModelica.jl is called *VariableModelica*.

	Mosilab	Sol
Type	Modelica extension	Modelica variant
Paradigm	Declarative	Declarative
Compilation technique	AOT-Compilation	Interpretation
Variability	Static	Dynamic
Declaration Scheme	Explicit	Implicit
Boundness	Bounded	Unbounded
Higher-order-models	No	Yes

Hydra	MCL	VariableModelica
Embedded in Haskell	Intermediate Representation	Modelica Extension
Functional	Functional	Declarative
JIT-Compilation	AOT-Compilation	JIT-Compilation
Dynamic	Static	Dynamic
Explicit	Implicit	Implicit
Unbounded	Unbounded	Unbounded
Yes	Yes	Yes

5.6 Summary

We have presented an experimental overview on how a Modelica compiler capable of JIT-Compilation behaves both when compiling and simulating large dynamic systems. Furthermore, in our experiment, we have highlighted the performance characteristics concerning compiling large models using components from the Modelica Standard Library, and we have provided initial estimates concerning the novel capabilities concerning JIT-Compilation.

From the experiments in Section 5.4, we conclude that the large-scale simulation of variable structured systems in the context of equation-based languages is both possible and feasible. However, while recompilation of the models only took a fraction of the total simulation time, the compiler currently recompiles the entire system, not just the part that was impacted by some structural change. A recommendation for future work would be to integrate some of the symbolic techniques proposed by Zimmer (2010) and aspects of separate compilation proposed by Höger (2019), to minimize the number of variables and equations that need to be recompiled when a model undergoes structural changes.

Furthermore, modelers would experience a significant reduction in modeling time formulating systems using these new constructs along with having new abstractions to express more dynamic models than previously possible. Such a reduction would be possible because instead of having separate models, different behavior could be captured in the same model.

6. Conclusion & Discussion

This thesis set out to provide answers to the following research questions:

1. What syntactic and semantic constructs are needed in an equation-based language for modeling and simulating VSS?
2. What characteristics of a modeling and simulation framework are appropriate for achieving VSS support?
3. How can VSS support for Modelica be realized to simulate large systems effectively?

In this section, we discuss each of the stated research questions.

6.1 What syntactic constructs are needed in a language to simulate VSS?

In this text, we have illustrated two possible approaches; the first is to express the system as a set of continuous-time state machines and express the transitions between these. While this approach can be realized without compilation during simulation, it requires the entire model to be processed. As Zimmer (2010) discussed, this might not be feasible since it requires the modeler to enumerate all the states ahead of time. Also, this approach is causal[1], and it diverges from the acausal design of equation-based languages such as Modelica. The Modelica language needs to be extended with constructs to support systems with a variable structure to express either explicit or implicit changes to the system. As discussed earlier, the advantage of the explicit approach is that a graphical representation of such transitions is straightforward.

However, as discussed, this diverges from the acausal design principles of Modelica. Ideally, the Modelica language should support both explicit and implicit use of VSS. The other approach is to provide the ability to express implicit transitions. In this thesis, we implemented support for this by introducing compilation during simulation. However, this is not strictly necessary; in some cases, implicit transitions could be realized using static analysis. One recent example of this is the work by Benveniste, Benoît Caillaud, and Malandain (2021) where they handle multi-mode models via static analysis and subsequently generate additional code for the new states that are introduced.

[1] That is, the transitions of the system are specified as a set of causal transitions between different model states

Still, a disadvantage of this approach is that all states need to be enumerated, which might result in an exponential increase in the model's size.

6.2 What kind of computational framework is suitable for achieving VSS support?

In this thesis, we have illustrated that for a Modelica compiler to support variable structure systems successfully; it seems that JIT-Compilation is advantageous. As discussed in the result section, this comes with an additional cost. It might be the case for smaller models that the cost of recompiling the system is more expensive than the simulation. Instead, for small models, it might be suitable to combine interpretation with JIT-Compilation. Ideally, such a framework should be capable of utilizing both interpretation and JIT-Compilation.

6.3 How can VSS support for Modelica be realized to simulate large systems effectively?

In this thesis, we have illustrated that it is possible to design such a framework by writing a Modelica environment in the Julia programming language. As the experiments in Chapter 5 illustrate that while frontend performance is still not on par with state-of-the-art compilers such as the OMC we believe that we can achieve better performance by tuning the frontend and improving the MetaModelica-Julia compatibility layer, *MetaModelica.jl*. For example, the performance of the final generated Julia code can be improved both in terms of compilation time and in term of simulation time. One suggestion would be for MTK to introduce *descalarization* or avoid *scalarization* during symbolic processing. Techniques for unscalarized processing is described in (Marzorati, Fernández, and Kofman 2022). Another alternative, is MTK support for *DAE-Mode* as described in (W. Braun, Casella, Bachmann, et al. 2017; Henningsson, Olsson, and Vanfretti 2019).

6.4 The work in a wider context

The research presented in this thesis should enable modelers to express cyber-physical models with greater accuracy where it is possible to express the system in the Modelica language where the system's structure is changing during simulation. By implementing these extensions to the Modelica language in a compiler that tries to adhere to the standards of the existing language, modeling know-how from existing libraries can be reused. We believe this would enable the work presented in this thesis to impact a wider audience than what is typical of a research framework.

Regarding research ethics, the source code to replicate the experiments is available, and the code written as a part of this thesis is licensed under an open-source license, adhering to the scientific principle of openness.

As discussed in the previous paragraph, the result of this work enables modelers to express new kinds of models in the Modelica language. This can be used both for good and nefarious purposes. For instance, by modeling harmful systems to humanity and society in general. I would argue that such applications of the results obtained in this thesis are morally wrong and that the duty to make such judgments falls on the individual scientist.

7. Future Work

In this chapter, we propose future research directions. Starting with discussing separate compilation in Section 7.1 and ending with debugging in Section 7.7.

7.1 Separate Compilation

While the experiments illustrated that the JIT-Compilation is inexpensive compared to other phases when simulating variable structure systems, it still results in increased costs when employing this scheme. Previous work such as (Zimmer 2010) shows that causalisation when simulating systems with variable structure can be done in steps so that the impact of a change in one part of the system should not affect the whole system. (Höger 2019) presents a theoretical framework to deal with the issue of compiling models separately and composing them at runtime. One direction for future research is to examine the practical implications of using such schemes within the context of *OpenModelica.jl*.

7.2 Graphical presentation

While this thesis has primarily focused on the application of VSS on textual descriptions of models, further studies need to be carried out to examine the impact of the work presented here concerning how to represent and present models with structural variability in a graphical modeling environment. While this has been done before in Mosilab, using state charts representing implicit variability has to my knowledge, not been investigated.

Broman (2021) discusses this issue. However, it remains to investigate the impact of such frameworks in terms of empirical software engineering.

7.3 Initialization

Another problem not addressed in this thesis is the problem of initialization when systems undergo structural changes. Currently, the modeler is not aided by the language. The current compiler runtime assumes that the user specifies a valid structural change. This is similar to how Modelica allows the user to specify initial equations to ensure that the system is in or near a steady state at the start of a simulation. Similar features should be introduced for structural transitions. Benveniste et al. (Benveniste, Benoît Caillaud, Elmqvist, Ghorbal, Otter, and Pouzet 2019; Benveniste, Benoît Caillaud, and

Malandain 2021) present some approaches to initialization in the context of systems with several modes.

7.4 Dynamic optimization & Model Reduction

The capability of having the simulation affect the model being simulated allows the model to be modified based on the model's behavior. This is possible in a framework that supports VSS. Some static approaches demonstrating this have been discussed in the thesis. However, it would be interesting to examine how to devise a scheme to handle it more dynamically. One example could be using predefined surrogate models instead of using a predefined model as done by (Chris Rackauckas, Anantharaman, Edelman, Gowda, Gwozdz, Jain, Laughman, Ma, Martinuzzi, Pal, Rajput, Saba, and V. Shah 2021). In the context of Modelica, Tinnerholm, Pop, Heuermann, and Sjölund (2021) conducted some experiments that illustrated how algebraic loops or other components of a system could be replaced with surrogate models. One application of the techniques introduced in this paper could be to experiment and see if parts of a model could be replaced with a surrogate if the conditions during the simulation allow parts of a model to be simulated with less detail. In that case, surrogatization techniques could possibly be employed in conjunction with VSS to improve simulation efficiency.

7.5 Verification

Extending Modelica with initial support for VSS and combining the process of simulation and compilation results in the possibility of new types of runtime errors. However, the new capabilities of formulating models might also aid the user. For example, it should be possible to run simulations interactively and see if adding additional components to the model might cause an error. To conclude, verifiability is a wide area of research in its own right and further research is required in the context of equation-based languages that supports VSS.

7.6 Cloud computing

Cloud Computing is a relatively new trend. Another area of future work would be to experiment with cloud computing and examine the benefits of, for example, distributing the simulation on several computing nodes.

7.7 Debugging

Debugging declarative equation-based languages is a difficult problem because the programmer's view of the program, the model, is different from what the

compiler for the language generates for final execution on the target machine. Debugging in the context of equation-based languages has been examined by (Pop, Sjölund, Ashgar, P. Fritzson, and Casella 2014; Sjölund 2015) However, adding support for varying model structure to the Modelica language complicates debugging further. One direction of future research could be to investigate how to combine features introduced in this thesis with efficient and user-friendly debugging.

Bibliography

[1] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson, 2007. ISBN: 0321486811.

[2] Mats Andersson. *An Object-Oriented Language for Model Representation*. Licentiate thesis. 1990.

[3] Mats Andersson. "Discrete event modelling and simulation in Omola." In: *IEEE Symposium on Computer-Aided Control System Design*. IEEE. 1992, pp. 262–268.

[4] K. J. Åström. "A perspective on modeling and simulation of complex dynamical systems." In: *Integrated Modeling of Complex Optomechanical Systems*. Ed. by Torben Andersen and Anita Enmark. Vol. 8336. International Society for Optics and Photonics. SPIE, 2011, pp. 13–22. DOI: 10.1117/12.916687.

[5] Karl Johan Åström, Hilding Elmqvist, Sven Erik Mattsson, et al. "Evolution of continuous-time modeling and simulation." In: *ESM*. 1998, pp. 9–18.

[6] Donald Augustin, Mark Fineberg, Bruce Johnsson, Robert Linebarger, and F.John Sansom. "The SCi Continuous System Simulation Language (CSSL)." In: *SIMULATION* 9.6 (1967). Ed. by Jon C Strauss, pp. 281–303. DOI: 10.1177/003754976700900601.

[7] John Aycock. "A brief history of just-in-time." In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113. DOI: 10.1145/857076.857077.

[8] Albert Benveniste, Benoît Caillaud, Hilding Elmqvist, Khalil Ghorbal, Martin Otter, and Marc Pouzet. "Multi-Mode DAE Models - Challenges, Theory and Implementation." In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 283–310. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_16. URL: https://doi.org/10.1007/978-3-319-91908-9_16.

[9] Albert Benveniste, Benoît Caillaud, and Mathias Malandain. "Handling Multimode Models and Mode Changes in Modelica." In: *Proceedings of the 14th International Modelica Conference*. Ed. by Martin Sjölund, Lena Buffoni, Adrian Pop, and Lennart Ochel. Linköping Electronic Conference Proceedings 181. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, Sept. 2021, pp. 507–517. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181507.

[10] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A Fresh Approach to Numerical Computing." In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: 10.1137/141000671. eprint: https://doi.org/10.1137/141000671. URL: https://doi.org/10.1137/141000671.

[11] Willi Braun, Francesco Casella, Bernhard Bachmann, et al. "Solving large-scale Modelica models: new approaches and experimental results using OpenModelica." In: *12 International Modelica Conference*. Linkoping University Electronic Press. 2017, pp. 557–563.

[12] Kathryn Eleda Brenan, Stephen L Campbell, and Linda Ruth Petzold. *Numerical solution of initial-value problems in differential-algebraic equations*. SIAM, 1995.

[13] David Broman. "Meta-Languages and Semantics for Equation-Based Modeling and Simulation." PhD thesis. Linköping University Electronic Press, 2010. ISBN: 978-91-7393-335-3.

[14] David Broman. "Interactive Programmatic Modeling." In: *ACM Trans. Embed. Comput. Syst.* 20.4 (May 2021). ISSN: 1539-9087. DOI: 10.1145/3431387. URL: https://doi.org/10.1145/3431387.

[15] P Bujakiewicz and PPJ van den Bosch. "Determination of perturbation index of a DAE with maximum weighted matching algorithm." In: *Proceedings of IEEE Symposium on Computer-Aided Control Systems Design (CACSD)*. IEEE. 1994, pp. 129–136.

[16] Luiz Fernando Capretz. "A Brief History of the Object-Oriented Approach." In: *SIGSOFT Softw. Eng. Notes* 28.2 (Mar. 2003), p. 6. ISSN: 0163-5948. DOI: 10.1145/638750.638778. URL: https://doi.org/10.1145/638750.638778.

[17] Francesco Casella. "Simulation of large-scale models in modelica: State of the art and future perspectives." In: *11th International Modelica Conference*. 2015, pp. 459–468.

[18] François E Cellier and Ernesto Kofman. *Continuous system simulation*. Springer Science & Business Media, 2006. ISBN: 0387261028.

[19] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. "Register Allocation via Coloring." In: *Comput. Lang.* 6.1 (Jan. 1981), pp. 47–57. ISSN: 0096-0551.

[20] Devendra K Chaturvedi. *Modeling and simulation of systems using MATLAB® and Simulink®*. CRC press, 2017. ISBN: 9781351834223.

[21] Jiahao Chen and Jarrett Revels. "Robust benchmarking in noisy environments." In: *arXiv e-prints*, arXiv:1608.04295 (Aug. 2016). arXiv: 1608.04295 [cs.PF].

[22] E Ward Cheney and David R Kincaid. *Numerical mathematics and computing*. Cengage Learning, 2003. ISBN: 0534389937.

[23] Simon Christ, Daniel Schwabeneder, and Christopher Rackauckas. "Plots. jl–a user extendable plotting API for the julia programming language." In: *arXiv preprint arXiv:2204.08775* (2022).

[24] Keith Cooper and Linda Torczon. *Engineering a compiler*. Elsevier, 2011. ISBN: 9780120884780.

[25] Hilding Elmqvist. "SIMNON-An Interactive Simulation Program for Non-Linear Systems." In: *Simulation'77: Proceedings of the international symposium, Montreaux, June 22-24, 1977*. Acta Press. 1977, pp. 85–89.

[26] Hilding Elmqvist. "A structured model language for large continuous systems." PhD thesis. Lund University, 1978.

[27] Hilding Elmqvist, Sven Erik Matsson, and Martin Otter. "Modelica extensions for multi-mode DAE systems." In: *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*. 096. Linköping University Electronic Press. 2014, pp. 183–193.

[28] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. "Modelica: The new object-oriented modeling language." In: *12th European Simulation Multiconference, Manchester, UK*. Vol. 5. 1998.

[29] Hilding Elmqvist, Andrea Neumayr, and Martin Otter. *Modia-dynamic modeling and simulation with julia*. 2018.

[30] Hilding Elmqvist and Martin Otter. "Innovations for future Modelica." In: *Proceedings of 12th International Modelica Conference*. Linköping University Electronic Press. 2017, pp. 693–702.

[31] Hilding Elmqvist, Martin Otter, Andrea Neumayr, and Gerhard Hippmann. "Modia - Equation Based Modeling and Domain Specific Algorithms." In: *Proceedings of the 14th International Modelica Conference*. Ed. by Martin Sjölund, Lena Buffoni, Adrian Pop, and Lennart Ochel. Linköping Electronic Conference Proceedings 181. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, Sept. 2021, pp. 73–86. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp2118173.

[32] Jay W Forrester. "Industrial dynamics—after the first decade." In: *Management science* 14.7 (1968), pp. 398–415.

[33] Jay W Forrester. "Urban dynamics." In: *IMR; Industrial Management Review (pre-1986)* 11.3 (1970), p. 67.

[34] Jay W Forrester. "System dynamics and the lessons of 35 years." In: *A systems-based approach to policymaking*. Springer, 1993, pp. 199–240.

[35] Peter Fritzson. *Principles of object-oriented modeling and simulation with Modelica 3.3: a cyber-physical approach*. John Wiley & Sons, 2014. ISBN: 978-1-118-85912-4.

[36] Peter Fritzson and Vadim Engelson. "Modelica—A unified object-oriented language for system modeling and simulation." In: *European Conference on Object-Oriented Programming*. Springer. 1998, pp. 67–90.

[37] Peter Fritzson and Dag Fritzson. "The need for high-level programming support in scientific computing applied to mechanical analysis." In: *Computers & structures* 45.2 (1992), pp. 387–395.

[38] Peter Fritzson, Adrian Pop, Karim Abdelhak, Adeel Asghar, Bernhard Bachmann, Willi Braun, Daniel Bouskela, Robert Braun, Lena Buffoni, Francesco Casella, et al. "The OpenModelica integrated environment for modeling, simulation, and model-based development." In: *Modeling, Identification and Control* 41.4 (2020), pp. 241–295.

[39] Peter Fritzson, Adrian Pop, and Martin Sjölund. *Towards Modelica 4 Meta-Programming and Language Modeling with MetaModelica 2.0*. Tech. rep. 2011:10. Linköping University, PELAB - Programming Environment Laboratory, May 2011. 297 pp. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-68361 (visited on 04/01/2013).

[40] Peter Fritzson, Adrian Pop, Martin Sjölund, and Adeel Asghar. "MetaModelica – A Symbolic-Numeric Modelica Language and Comparison to Julia." In: *Proceedings of the 13th International Modelica Conference*. Regensburg, Germany: Modelica Association and Linköping University Electronic Press, Mar. 2019. DOI: 10.3384/ecp19157289.

[41] George Giorgidze. "First-class models: On a noncausal language for higher-order and structurally dynamic modelling and simulation." PhD thesis. University of Nottingham, 2012.

[42] George Giorgidze and Henrik Nilsson. "Higher-order non-causal modelling and simulation of structurally dynamic systems." In: *Proceedings of the 7th International Modelica Conference; Como; Italy; 20-22 September 2009*. 043. Linköping University Electronic Press. 2009, pp. 208–218.

[43] Peterson H.E and F.J Sansom. *MIMIC - A digital simulator program.* 1965.

[44] Erik Henningsson, Hans Olsson, and Luigi Vanfretti. "DAE Solvers for Large-Scale Hybrid Models." In: *Proceedings of the 13th International Modelica Conference, Regensburg, Germany, March 4–6, 2019*. 2019, pp. 491–500.

[45] Alan C Hindmarsh, Peter N Brown, Keith E Grant, Steven L Lee, Radu Serban, Dan E Shumaker, and Carol S Woodward. "SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers." In: *ACM Transactions on Mathematical Software (TOMS)* 31.3 (2005), pp. 363–396.

[46] Christoph Höger. “Compiling Modelica : about the separate translation of models from Modelica to OCaml and its impact on variable-structure modeling.” Doctoral Thesis. Berlin: Technische Universität Berlin, 2019. DOI: 10.14279/depositonce-8354. URL: http://dx.doi.org/10.14279/depositonce-8354.

[47] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages and computation.* Pearson Addison-Wesley, 2007. ISBN: 0321476174.

[48] CP Jobling, Phil W Grant, HA Barker, and Peter Townsend. “Object-oriented programming in control system design: a survey.” In: *Automatica* 30.8 (1994), pp. 1221–1261.

[49] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for lifelong program analysis & transformation.” In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[50] Yingbo Ma, Shashi Gowda, Ranjan Anantharaman, Chris Laughman, Viral Shah, and Chris Rackauckas. *ModelingToolkit: A Composable Graph Transformation System For Equation-Based Modeling.* 2021. arXiv: 2103.05244 [cs.MS].

[51] Denise Marzorati, Joaquin Fernández, and Ernesto Kofman. “Efficient connection processing in equation–based object–oriented models.” In: *Applied Mathematics and Computation* 418 (2022), p. 126842. ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2021.126842.

[52] M Douglas McIlroy. “Macro instruction extensions of compiler languages.” In: *Communications of the ACM* 3.4 (1960), pp. 214–220.

[53] Donella H Meadows. *Thinking in systems: A primer.* chelsea green publishing, 2008. ISBN: 9781603580557.

[54] Alexandra Mehlhase. “A Python framework to create and simulate models with variable structure in common simulation environments.” In: *Mathematical and Computer Modelling of Dynamical Systems* 20.6 (2014), pp. 566–583.

[55] Merriam-Webster. *System.* In: *Merriam-Webster.com dictionary.* URL: https://www.merriam-webster.com/dictionary/system (visited on 11/08/2020).

[56] Edward EL Mitchell and Joseph S Gauthier. “Advanced continuous simulation language (ACSL).” In: *Simulation* 26.3 (1976), pp. 72–78.

[57] Steven Muchnick. *Advanced compiler design & implementation.* Morgan Kaufmann, 1997. ISBN: 9789814066242.

[58] Henrik Nilsson, John Peterson, and Paul Hudak. “Functional hybrid modeling.” In: *International Symposium on Practical Aspects of Declarative Languages.* Springer. 2003, pp. 376–390.

[59] Hairer SP Norsett E and G Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, 1993. ISBN: 978-3-540-78862-1.

[60] Christoph Nytsch-Geusen, Thilo Ernst, André Nordwig, Peter Schneider, Peter Schwarz, Matthias Vetter, Christof Wittwer, Andreas Holm, Thierry Nouidui, Jürgen Leopold, et al. "MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics." In: *Proceedings of the 4th International Modelica Conference TU Hamburg-Harburg*. Vol. 2. Citeseer. 2005.

[61] Constantinos C Pantelides. "The consistent initialization of differential-algebraic systems." In: *SIAM Journal on Scientific and Statistical Computing* 9.2 (1988), pp. 213–231.

[62] David J Pearce and Paul HJ Kelly. "A dynamic algorithm for topologically sorting directed acyclic graphs." In: *International Workshop on Experimental and Efficient Algorithms*. Springer. 2004, pp. 383–398.

[63] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. "A design science research methodology for information systems research." In: *Journal of management information systems* 24.3 (2007), pp. 45–77.

[64] Linda R Petzold. *Description of DASSL: a differential/algebraic system solver*. Tech. rep. Sandia National Labs., Livermore, CA (USA), 1982.

[65] Peter C Piela, Thomas G Epperly, Karl M Westerberg, and Arthur W Westerberg. "ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language." In: *Computers & chemical engineering* 15.1 (1991), pp. 53–72.

[66] Adrian Pop and Peter Fritzson. "Metamodelica: A unified equation-based semantical and mathematical modeling language." In: *Joint Modular Languages Conference*. Springer. 2006, pp. 211–229.

[67] Adrian Pop, Per Östlund, Francesco Casella, Martin Sjölund, Rüdiger Franke, et al. "A new openmodelica compiler high performance frontend." In: *13th International Modelica Conference*. Vol. 157. 2019, pp. 689–698.

[68] Adrian Pop, Martin Sjölund, Adeel Ashgar, Peter Fritzson, and Francesco Casella. "Integrated Debugging of Modelica Models." In: (2014).

[69] Chris Rackauckas, Ranjan Anantharaman, Alan Edelman, Shashi Gowda, Maja Gwozdz, Anand Jain, Chris Laughman, Yingbo Ma, Francesco Martinuzzi, Avik Pal, Utkarsh Rajput, Elliot Saba, and Viral Shah. "Composing Modeling and Simulation with Machine Learning in Julia." In: *Proceedings of the 14th International Modelica Conference*. Ed. by Martin Sjölund, Lena Buffoni, Adrian Pop, and Lennart Ochel.

Linköping Electronic Conference Proceedings 181. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, Sept. 2021, pp. 97–107. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp2118197.

[70] Christopher Rackauckas and Qing Nie. "Differentialequations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia." In: *Journal of Open Research Software* 5.1 (2017).

[71] Theodore H Romer, Dennis Lee, Geoffrey M Voelker, Alec Wolman, Wayne A Wong, Jean-Loup Baer, Brian N Bershad, and Henry M Levy. "The structure and performance of interpreters." In: *ACM SIGPLAN Notices* 31.9 (1996), pp. 150–159.

[72] Zekeriya Sarı and Serkan Günel. "Causal.jl: A Modeling and Simulation Framework for Causal Models." In: *Proceedings of the JuliaCon Conferences* 1.1 (2021), p. 71. DOI: 10.21105/jcon.00071. URL: https://doi.org/10.21105/jcon.00071.

[73] Martin Sjölund. "Tools and Methods for Analysis, Debugging, and Performance Improvement of Equation-Based Models." PhD thesis. 2015. ISBN: 978-91-7519-071-6.

[74] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley Educational Publishers Inc, 2013. ISBN: 9780321958327.

[75] John Tinnerholm, Adrian Pop, Andreas Heuermann, and Martin Sjölund. "OpenModelica.jl: A modular and extensible Modelica compiler framework in Julia targeting ModelingToolkit.jl." In: *Proceedings of the 14th International Modelica Conference*. Ed. by Martin Sjölund, Lena Buffoni, Adrian Pop, and Lennart Ochel. Linköping Electronic Conference Proceedings 181. Linköping, Sweden: Modelica Association and Linköping University Electronic Press, Sept. 2021, pp. 109–117. ISBN: 978-91-7929-027-6. DOI: 10.3384/ecp21181109.

[76] John Tinnerholm, Adrian Pop, Martin Sjölund, Andreas Heuermann, and Karim Abdelhak. "Towards an Open-Source Modelica Compiler in Julia." In: *Proceedings of Asian Modelica Conference Tokyo, Japan, October 08-09*. 2020. DOI: 10.3384/ecp2020174143.

[77] John Tinnerholm, Martin Sjölund, and Adrian Pop. "Towards introducing just-in-time compilation in a modelica compiler." In: *Proceedings of the 9th International Workshop on Equation-based Object-oriented Modeling Languages and Tools*. 2019, pp. 11–19.

[78] Lars Viklund and Peter Fritzson. "Objectmath–an object-oriented language and environment for symbolic and numerical processing in scientific computing." In: *Scientific Programming* 4.4 (1995), pp. 229–250.

[79] Lars Viklund, Johan Herber, and Peter Fritzson. "The implementation of ObjectMath—a high-level programming environment for scientific computing." In: *International Conference on Compiler Construction*. Springer. 1992, pp. 312–318.

[80] Gerhard Wanner and Ernst Hairer. *Solving ordinary differential equations II*. Vol. 375. Springer Berlin Heidelberg, 1996. ISBN: 978-3-642-05221-7.

[81] Michael Weisberg. *Simulation and similarity: Using models to understand the world*. Oxford University Press, 2012. ISBN: 0190265124.

[82] Stephen Wolfram. *Mathematica: a system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.

[83] Dirk Zimmer. "Equation-based modeling of variable-structure systems." PhD thesis. ETH Zürich, 2010.

A. Source code examples

A.1 Models and source code for Chapter 2

This sections contains models and source code for chapter 2.

Listing A.1.1 The Modelica code of the bouncing ball model in Section 2.1.1

```
model FreeFall
  Real h(start = 1.0);
  Real v(start = 0.0);
  parameter Real g = 9.81;
equation
 der(h) = v;
 der(v) = -g;
end FreeFall;
```

Listing A.1.2 The Modelica code of the bouncing ball model in Section 2.1.3

```
model BouncingBallReals
  parameter Real e=0.3;
  parameter Real g=9.81;
  Real h(start = 1);
  Real v(start = 0);
equation
  der(h) = v;
  der(v) = -g;
  when h <= 0 then
    reinit(v, -e*pre(v));
  end when;
end BouncingBallReals;
```

Listing A.1.3 Julia program to simulate the RLC circuit in Figure 2.6

```
using Plots
R1R = 1.0
CC = 0.01
R2R = 1.0
LL = 0.1
ACA = 1.0
ACw = 1.0
function Runge(t::Real, Δt::Real, f::Function, vars...)
  local k1 = f(t, vars...)
  local k2 = f(t + Δt/2, (vars .+ (Δt .* (k1 ./ 2)))...)
  local k3 = f(t + Δt/2, (vars .+ (Δt .* (k2 ./ 2)))...)
  local k4 = f(t + Δt, (vars .+ (Δt.*k3)...))
  vars = (Δt .* ((k1 .+ (2 .* k2) .+ (2 .* k3) .+ k4) ./ 6)) .+ vars
  return vars
end
function H(t, Cv, Li)
  x = ACA * sin(ACw * t)
  DCv = ((ACA * x - Cv) / R1R) / CC
  DLi = (ACA * x - R2R * Li) / LL
  return (DCv, DLi)
end
function K(t, Cv, Li)
  x = sin(ACw * t)
  ACv = ACA * x
  R2v = R2R * Li
  Lv = ACv - R2v
  R1v = ACv - Cv
  Ci = R1v / R1R
  ACi = (-Li) - Ci
  Gpi = Li - ((-Ci) - ACi)
  return (ACv, R2v, Lv, R1v, Ci, ACi, Gpi)
end
function simulate()
  tArr = []
  CvArr = []
  LiArr = []
  (Cv, Li) = (0.0, 0.0)
  (x, ACv, R2v, Lv, R1v, Ci, ACi, Gp) = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)
  Δt = 0.001
  for i in 0.00:Δt:3.0
    local t = i
    (Cv, Li) = Runge(t, Δt, H, Cv, Li)
    (ACv, R2v, Lv, R1v, Ci, ACi, Gpi) = K(t, Cv, Li)
    push!(tArr, t)
    push!(CvArr, Cv)
    push!(LiArr, Li)
  end
  return [tArr, CvArr, LiArr]
end
#= Simulate and plot =#
res = simulate()
fig = plot(res[1],res[2:3])
Plots.plot(fig)
```

A.2 The Electrical component library

The different components that constitute the electrical component library is presented here. Adapted from (P. Fritzson 2014).

Listing A.2.1 Definition of some basic electrical components using Modelica.

```
package ElectricalComponents
  connector Pin
    Real v;
    flow Real i;
  end Pin;

  partial model TwoPin
    Real v;
    Real i;
    Pin p;
    Pin n;
  equation
    v = p.v - n.v;
    0 = p.i + n.i;
    i = p.i;
  end TwoPin;

  model Resistor
  extends TwoPin;
    parameter Real R;
  equation
    R*i = v;
  end Resistor;

  model Inductor
    extends TwoPin;
    parameter Real L;
  equation
    L*der(i) = v;
  end Inductor;

  model Capacitor
  extends TwoPin;
    parameter Real C ;
  equation
    i=C*der(v);
  end Capacitor;

  model Source
  extends TwoPin;
    parameter Real A,w;
  equation
    v = A*sin(w*time);
  end Source;

  model Ground
    Pin p;
  equation
    p.v = 0;
  end Ground;

end ElectricalComponents;
```

B. Tables

B.1 Simulation time measurements

Table B.1: Time spent conducting numerical simulation for *OpenModelica.jl*. $\hat{x}$ is the sample median, $\hat{\mu}$ is the sample mean and $\hat{\sigma}$ is the sample standard deviation. The parameter N corresponds to the total amount of equations and variables in the system under simulation. During the experiment the last model, where $N = 25600$ the benchmark program ran into issues, hence three samples where collected manually for this last test. This model and all other models involved are available upon request.

N	$\hat{x}$	$\hat{\mu}$	$\hat{\sigma}$
10	47.854 μs	48.927 μs	8.084 μs
100	474.505 μs	477.517 μs	37.169 μs
200	1.688 ms	1.688 ms	43.439 μs
400	5.592 ms	5.602 ms	80.194 μs
800	23.104 ms	23.121 ms	209.631 μs
1600	0.223 s	0.224 s	410.541 μs
3200	1.818 s	1.818 s	2.448 ms
6400	3.812 s	3.792 s	150.810 ms
12800	12.878 s	12.795 s	394.981ms
25600	41.018 s	42.679 s	5.276 s

Table B.2: Time spent conducting numerical simulation for the OMC. $\hat{x}$ is the sample median, $\hat{\mu}$ is the sample mean and $\hat{\sigma}$ is the sample standard deviation. The parameter N corresponds to the total amount of equations and variables in the system under simulation.

N	$\hat{x}$	$\hat{\mu}$	$\hat{\sigma}$
10	0.083 s	0.084 s	0.003 s
100	0.092 s	0.093 s	0.003 s
200	0.104 s	0.105 s	0.003 s
400	0.135 s	0.136 s	0.007 s
800	0.211 s	0.211 s	0.005 s
1600	0.446 s	0.447 s	0.011 s
3200	1.046 s	1.049 s	0.021 s
6400	2.938 s	2.946 s	0.058 s
12800	10.006 s	10.004 s	0.096 s
25600	46.342 s	46.301 s	0.208 s

B.2 Compilation time measurements

This section contains the compilation time measurements from Chapter 5.

Table B.3: Time spent compiling when generating flat Modelica for the transmission line model in Listing 5.3.1 using OMFrontend.jl. $\hat{x}$ is the sample median, $\hat{\mu}$ is the sample mean and $\hat{\sigma}$ is the sample standard deviation.

N	Equations and Variables	$\hat{x}$	$\hat{\mu}$	$\hat{\sigma}$
10	245	0.723 s	0.727 s	0.01 s
20	455	1.039 s	1.045 s	0.012 s
40	875	1.631 s	1.638	0.015 s
80	1715	2.836 s	2.835 s	0.019 s
160	3395	5.219 s	5.219 s	0.015 s
320	6755	10.039 s	10.035 s	0.029 s
640	13475	19.776 s	19.763 s	0.072 s
1280	26915	39.572 s	39.592 s	0.322 s

Table B.4: Required memory when generating flat Modelica for the transmission line model in 5.3.1.

N	Equations and Variables	Memory (MiB)
10	245	24.84 MiB
20	455	35.12 MiB
40	875	55.52 MiB
80	1715	97.55 MiB
160	3395	182.21 MiB
320	6755	362.87 MiB
640	13475	746.70 MiB
1280	26915	1.62 GiB

Table B.5: Compilation time when generating flat Modelica for the transmission line model in Listing 5.3.1 using the OMC. $\hat{x}$ is the sample median, $\hat{\mu}$ is the sample mean and $\hat{\sigma}$ is the sample standard deviation.

N	Equations and Variables	$\hat{x}$	$\hat{\mu}$	$\hat{\sigma}$
10	245	243.479 ms	242.933 ms	71.733 ms
20	455	269.227 ms	269.371 ms	86.438 ms
40	875	288.616 ms	287.360 ms	76.614 ms
80	1715	425.284 ms	391.838 ms	89.497 ms
160	3395	640.857 ms	639.815 ms	25.950 ms
320	6755	1.441 s	1.440 s	7.798 ms
640	13475	3.341 s	3.369 s	127.039 ms
1280	26915	10.617 s	10.758 s	296.437 ms

Department of Computer and Information Science
Linköpings universitet

Licentiate Theses

Linköpings Studies in Science and Technology
Faculty of Arts and Sciences

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

No 28 **Arne Jönsson, Mikael Patel:** An Interactive Flowcharting Technique for Communicating and Realizing Algorithms, 1984.

No 29 **Johnny Eckerland:** Retargeting of an Incremental Code Generator, 1984.

No 48 **Henrik Nordin:** On the Use of Typical Cases for Knowledge-Based Consultation and Teaching, 1985.

No 52 **Zebo Peng:** Steps Towards the Formalization of Designing VLSI Systems, 1985.

No 60 **Johan Fagerström:** Simulation and Evaluation of Architecture based on Asynchronous Processes, 1985.

No 71 **Jalal Maleki:** ICONStraint, A Dependency Directed Constraint Maintenance System, 1987.

No 72 **Tony Larsson:** On the Specification and Verification of VLSI Systems, 1986.

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

No 74 **Christos Levcopoulos:** New Results about the Approximation Behavior of the Greedy Triangulation, 1986.

No 104 **Shamsul I. Chowdhury: S**tatistical Expert Systems - a Special Application Area for Knowledge-Based Computer Methodology, 1987.

No 108 **Rober Bilos:** Incremental Scanning and Token-Based Editing, 1987.

No 111 **Hans Block:** SPORT-SORT Sorting Algorithms and Sport Tournaments, 1987.

No 113 **Ralph Rönnquist:** Network and Lattice Based Approaches to the Representation of Knowledge, 1987.

No 118 **Mariam Kamkar, Nahid Shahmehri:** Affect-Chaining in Program Flow Analysis Applied to Queries of Programs, 1987.

No 126 **Dan Strömberg:** Transfer and Distribution of Application Programs, 1987.

No 127 **Kristian Sandahl:** Case Studies in Knowledge Acquisition, Migration and User Acceptance of Expert Systems, 1987.

No 139 **Christer Bäckström:** Reasoning about Interdependent Actions, 1988.

No 140 **Mats Wirén:** On Control Strategies and Incrementality in Unification-Based Chart Parsing, 1988.

No 146 **Johan Hultman:** A Software System for Defining and Controlling Actions in a Mechanical System, 1988.

No 150 **Tim Hansen:** Diagnosing Faults using Knowledge about Malfunctioning Behavior, 1988.

No 165 **Jonas Löwgren:** Supporting Design and Management of Expert System User Interfaces, 1989.

No 166 **Ola Petersson:** On Adaptive Sorting in Sequential and Parallel Models, 1989.

No 174 **Yngve Larsson:** Dynamic Configuration in a Distributed Environment, 1989.

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

No 181 **Henrik Eriksson:** A Study in Domain-Oriented Tool Support for Knowledge Acquisition, 1989.

No 184 **Ivan Rankin:** The Deep Generation of Text in Expert Critiquing Systems, 1989.

No 187 **Simin Nadjm-Tehrani:** Contributions to the Declarative Approach to Debugging Prolog Programs, 1989.

No 189 **Magnus Merkel:** Temporal Information in Natural Language, 1989.

No 196 **Ulf Nilsson:** A Systematic Approach to Abstract Interpretation of Logic Programs, 1989.

No 197 **Staffan Bonnier:** Horn Clause Logic with External Procedures: Towards a Theoretical Framework, 1989.

No 203 **Christer Hansson:** A Prototype System for Logical Reasoning about Time and Action, 1990.

No 212 **Björn Fjellborg:** An Approach to Extraction of Pipeline Structures for VLSI High-Level Synthesis, 1990.

No 230 **Patrick Doherty:** A Three-Valued Approach to Non-Monotonic Reasoning, 1990.

No 237 **Tomas Sokolnicki:** Coaching Partial Plans: An Approach to Knowledge-Based Tutoring, 1990.

No 250 **Lars Strömberg:** Postmortem Debugging of Distributed Systems, 1990.

No 253 **Torbjörn Näslund:** SLDFA-Resolution - Computing Answers for Negative Queries, 1990.

No 260 **Peter D. Holmes:** Using Connectivity Graphs to Support Map-Related Reasoning, 1991.

No 283 **Olof Johansson:** Improving Implementation of Graphical User Interfaces for Object-Oriented Knowledge- Bases, 1991.

No 298 **Rolf G Larsson:** Aktivitetsbaserad kalkylering i ett nytt ekonomisystem, 1991.

No 318 **Lena Strömbäck:** Studies in Extended Unification-Based Formalism for Linguistic Description: An Algorithm for Feature Structures with Disjunction and a Proposal for Flexible Systems, 1992.

No 319 **Mikael Pettersson:** DML-A Language and System for the Generation of Efficient Compilers from Denotational Specification, 1992.

No 326 **Andreas Kågedal:** Logic Programming with External Procedures: an Implementation, 1992.

No 328 **Patrick Lambrix:** Aspects of Version Management of Composite Objects, 1992.

No 333 **Xinli Gu:** Testability Analysis and Improvement in High-Level Synthesis Systems, 1992.

No 335 **Torbjörn Näslund:** On the Role of Evaluations in Iterative Development of Managerial Support Systems, 1992.

No 348 **Ulf Cederling:** Industrial Software Development - a Case Study, 1992.

No 352 **Magnus Morin:** Predictable Cyclic Computations in Autonomous Systems: A Computational Model and Implementation, 1992.

No 371 **Mehran Noghabai:** Evaluation of Strategic Investments in Information Technology, 1993.

No 378 **Mats Larsson:** A Transformational Approach to Formal Digital System Design, 1993.

No 380 **Johan Ringström:** Compiler Generation for Parallel Languages from Denotational Specifications, 1993.
No 381 **Michael Jansson:** Propagation of Change in an Intelligent Information System, 1993.
No 383 **Jonni Harrius:** An Architecture and a Knowledge Representation Model for Expert Critiquing Systems, 1993.
No 386 **Per Österling:** Symbolic Modelling of the Dynamic Environments of Autonomous Agents, 1993.
No 398 **Johan Boye:** Dependency-based Groudness Analysis of Functional Logic Programs, 1993.
No 402 **Lars Degerstedt:** Tabulated Resolution for Well Founded Semantics, 1993.
No 406 **Anna Moberg:** Satellitkontor - en studie av kommunikationsmönster vid arbete på distans, 1993.
No 414 **Peter Carlsson:** Separation av företagsledning och finansiering - fallstudier av företagsledarutköp ur ett agent-teoretiskt perspektiv, 1994.
No 417 **Camilla Sjöström:** Revision och lagreglering - ett historiskt perspektiv, 1994.
No 436 **Cecilia Sjöberg:** Voices in Design: Argumentation in Participatory Development, 1994.
No 437 **Lars Viklund:** Contributions to a High-level Programming Environment for a Scientific Computing, 1994.
No 440 **Peter Loborg:** Error Recovery Support in Manufacturing Control Systems, 1994.
FHS 3/94 **Owen Eriksson:** Informationssystem med verksamhetskvalitet - utvärdering baserat på ett verksamhetsinriktat och samskapande perspektiv, 1994.
FHS 4/94 **Karin Pettersson:** Informationssystemstrukturering, ansvarsfördelning och användarinflytande - En komparativ studie med utgångspunkt i två informationssystemstrategier, 1994.
No 441 **Lars Poignant:** Informationsteknologi och företagsetablering - Effekter på produktivitet och region, 1994.
No 446 **Gustav Fahl:** Object Views of Relational Data in Multidatabase Systems, 1994.
No 450 **Henrik Nilsson:** A Declarative Approach to Debugging for Lazy Functional Languages, 1994.
No 451 **Jonas Lind:** Creditor - Firm Relations: an Interdisciplinary Analysis, 1994.
No 452 **Martin Sköld:** Active Rules based on Object Relational Queries - Efficient Change Monitoring Techniques, 1994.
No 455 **Pär Carlshamre:** A Collaborative Approach to Usability Engineering: Technical Communicators and System Developers in Usability-Oriented Systems Development, 1994.
FHS 5/94 **Stefan Cronholm:** Varför CASE-verktyg i systemutveckling? - En motiv- och konsekvensstudie avseende arbetssätt och arbetsformer, 1994.
No 462 **Mikael Lindvall:** A Study of Traceability in Object-Oriented Systems Development, 1994.
No 463 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av Sandviks förvärv av Bahco Verktyg, 1994.
No 464 **Hans Olsén:** Collage Induction: Proving Properties of Logic Programs by Program Synthesis, 1994.
No 469 **Lars Karlsson:** Specification and Synthesis of Plans Using the Features and Fluents Framework, 1995.
No 473 **Ulf Söderman:** On Conceptual Modelling of Mode Switching Systems, 1995.
No 475 **Choong-ho Yi:** Reasoning about Concurrent Actions in the Trajectory Semantics, 1995.
No 476 **Bo Lagerström:** Successiv resultatavräkning av pågående arbeten. - Fallstudier i tre byggföretag, 1995.
No 478 **Peter Jonsson:** Complexity of State-Variable Planning under Structural Restrictions, 1995.
FHS 7/95 **Anders Avdic:** Arbetsintegrerad systemutveckling med kalkylprogram, 1995.
No 482 **Eva L Ragnemalm:** Towards Student Modelling through Collaborative Dialogue with a Learning Companion, 1995.
No 488 **Eva Toller:** Contributions to Parallel Multiparadigm Languages: Combining Object-Oriented and Rule-Based Programming, 1995.
No 489 **Erik Stoy:** A Petri Net Based Unified Representation for Hardware/Software Co-Design, 1995.
No 497 **Johan Herber:** Environment Support for Building Structured Mathematical Models, 1995.
No 498 **Stefan Svenberg:** Structure-Driven Derivation of Inter-Lingual Functor-Argument Trees for Multi-Lingual Generation, 1995.
No 503 **Hee-Cheol Kim:** Prediction and Postdiction under Uncertainty, 1995.
FHS 8/95 **Dan Fristedt:** Metoder i användning - mot förbättring av systemutveckling genom situationell metodkunskap och metodanalys, 1995.
FHS 9/95 **Malin Bergvall:** Systemförvaltning i praktiken - en kvalitativ studie avseende centrala begrepp, aktiviteter och ansvarsroller, 1995.
No 513 **Joachim Karlsson:** Towards a Strategy for Software Requirements Selection, 1995.
No 517 **Jakob Axelsson:** Schedulability-Driven Partitioning of Heterogeneous Real-Time Systems, 1995.
No 518 **Göran Forslund:** Toward Cooperative Advice-Giving Systems: The Expert Systems Experience, 1995.
No 522 **Jörgen Andersson:** Bilder av småföretagares ekonomistyrning, 1995.
No 538 **Staffan Flodin:** Efficient Management of Object-Oriented Queries with Late Binding, 1996.
No 545 **Vadim Engelson:** An Approach to Automatic Construction of Graphical User Interfaces for Applications in Scientific Computing, 1996.
No 546 **Magnus Werner :** Multidatabase Integration using Polymorphic Queries and Views, 1996.
FiF-a 1/96 **Mikael Lind:** Affärsprocessinriktad förändringsanalys - utveckling och tillämpning av synsätt och metod, 1996.
No 549 **Jonas Hallberg:** High-Level Synthesis under Local Timing Constraints, 1996.
No 550 **Kristina Larsen:** Förutsättningar och begränsningar för arbete på distans - erfarenheter från fyra svenska företag. 1996.
No 557 **Mikael Johansson:** Quality Functions for Requirements Engineering Methods, 1996.
No 558 **Patrik Nordling:** The Simulation of Rolling Bearing Dynamics on Parallel Computers, 1996.
No 561 **Anders Ekman:** Exploration of Polygonal Environments, 1996.
No 563 **Niclas Andersson:** Compilation of Mathematical Models to Parallel Code, 1996.

No 567 **Johan Jenvald:** Simulation and Data Collection in Battle Training, 1996.
No 575 **Niclas Ohlsson:** Software Quality Engineering by Early Identification of Fault-Prone Modules, 1996.
No 576 **Mikael Ericsson:** Commenting Systems as Design Support—A Wizard-of-Oz Study, 1996.
No 587 **Jörgen Lindström:** Chefers användning av kommunikationsteknik, 1996.
No 589 **Esa Falkenroth:** Data Management in Control Applications - A Proposal Based on Active Database Systems, 1996.
No 591 **Niclas Wahllöf:** A Default Extension to Description Logics and its Applications, 1996.
No 595 **Annika Larsson:** Ekonomisk Styrning och Organisatorisk Passion - ett interaktivt perspektiv, 1997.
No 597 **Ling Lin:** A Value-based Indexing Technique for Time Sequences, 1997.
No 598 **Rego Granlund:** C^{3}Fire - A Microworld Supporting Emergency Management Training, 1997.
No 599 **Peter Ingels:** A Robust Text Processing Technique Applied to Lexical Error Recovery, 1997.
No 607 **Per-Arne Persson:** Toward a Grounded Theory for Support of Command and Control in Military Coalitions, 1997.
No 609 **Jonas S Karlsson:** A Scalable Data Structure for a Parallel Data Server, 1997.
FiF-a 4 **Carita Åbom:** Videomötesteknik i olika affärssituationer - möjligheter och hinder, 1997.
FiF-a 6 **Tommy Wedlund**: Att skapa en företagsanpassad systemutvecklingsmodell - genom rekonstruktion, värdering och vidareutveckling i T50-bolag inom ABB, 1997.
No 615 **Silvia Coradeschi**: A Decision-Mechanism for Reactive and Coordinated Agents, 1997.
No 623 **Jan Ollinen:** Det flexibla kontorets utveckling på Digital - Ett stöd för multiflex? 1997.
No 626 **David Byers:** Towards Estimating Software Testability Using Static Analysis, 1997.
No 627 **Fredrik Eklund:** Declarative Error Diagnosis of GAPLog Programs, 1997.
No 629 **Gunilla Ivefors:** Krigsspel och Informationsteknik inför en oförutsägbar framtid, 1997**.**
No 631 **Jens-Olof Lindh:** Analysing Traffic Safety from a Case-Based Reasoning Perspective, 1997
No 639 **Jukka Mäki-Turja:**. Smalltalk - a suitable Real-Time Language, 1997.
No 640 **Juha Takkinen**: CAFE: Towards a Conceptual Model for Information Management in Electronic Mail, 1997.
No 643 **Man Lin**: Formal Analysis of Reactive Rule-based Programs, 1997.
No 653 **Mats Gustafsson**: Bringing Role-Based Access Control to Distributed Systems, 1997.
FiF-a 13 **Boris Karlsson:** Metodanalys för förståelse och utveckling av systemutvecklingsverksamhet. Analys och värdering av systemutvecklingsmodeller och dess användning, 1997.
No 674 **Marcus Bjäreland:** Two Aspects of Automating Logics of Action and Change - Regression and Tractability, 1998.
No 676 **Jan Håkegård**: Hierarchical Test Architecture and Board-Level Test Controller Synthesis, 1998.
No 668 **Per-Ove Zetterlund**: Normering av svensk redovisning - En studie av tillkomsten av Redovisningsrådets rekommendation om koncernredovisning (RR01:91), 1998.
No 675 **Jimmy Tjäder**: Projektledaren & planen - en studie av projektledning i tre installations- och systemutvecklingsprojekt, 1998.
FiF-a 14 **Ulf Melin**: Informationssystem vid ökad affärs- och processorientering - egenskaper, strategier och utveckling, 1998.
No 695 **Tim Heyer**: COMPASS: Introduction of Formal Methods in Code Development and Inspection, 1998.
No 700 **Patrik Hägglund:** Programming Languages for Computer Algebra, 1998.
FiF-a 16 **Marie-Therese Christiansson:** Inter-organisatorisk verksamhetsutveckling - metoder som stöd vid utveckling av partnerskap och informationssystem, 1998.
No 712 **Christina Wennestam:** Information om immateriella resurser. Investeringar i forskning och utveckling samt i personal inom skogsindustrin, 1998.
No 719 **Joakim Gustafsson:** Extending Temporal Action Logic for Ramification and Concurrency, 1998.
No 723 **Henrik André-Jönsson:** Indexing time-series data using text indexing methods, 1999.
No 725 **Erik Larsson:** High-Level Testability Analysis and Enhancement Techniques, 1998.
No 730 **Carl-Johan Westin:** Informationsförsörjning: en fråga om ansvar - aktiviteter och uppdrag i fem stora svenska organisationers operativa informationsförsörjning, 1998.
No 731 **Åse Jansson:** Miljöhänsyn - en del i företags styrning, 1998.
No 733 **Thomas Padron-McCarthy:** Performance-Polymorphic Declarative Queries, 1998.
No 734 **Anders Bäckström:** Värdeskapande kreditgivning - Kreditriskhantering ur ett agentteoretiskt perspektiv, 1998.
FiF-a 21 **Ulf Seigerroth:** Integration av förändringsmetoder - en modell för välgrundad metodintegration, 1999.
FiF-a 22 **Fredrik Öberg:** Object-Oriented Frameworks - A New Strategy for Case Tool Development, 1998.
No 737 **Jonas Mellin:** Predictable Event Monitoring, 1998.
No 738 **Joakim Eriksson:** Specifying and Managing Rules in an Active Real-Time Database System, 1998.
FiF-a 25 **Bengt E W Andersson:** Samverkande informationssystem mellan aktörer i offentliga åtaganden - En teori om aktörsarenor i samverkan om utbyte av information, 1998.
No 742 **Pawel Pietrzak:** Static Incorrectness Diagnosis of CLP (FD), 1999.
No 748 **Tobias Ritzau:** Real-Time Reference Counting in RT-Java, 1999.
No 751 **Anders Ferntoft:** Elektronisk affärskommunikation - kontaktkostnader och kontaktprocesser mellan kunder och leverantörer på producentmarknader, 1999.
No 752 **Jo Skåmedal:** Arbete på distans och arbetsformens påverkan på resor och resmönster, 1999.
No 753 **Johan Alvehus:** Mötets metaforer. En studie av berättelser om möten, 1999.

No 754 **Magnus Lindahl:** Bankens villkor i låneavtal vid kreditgivning till högt belånade företagsförvärv: En studie ur ett agentteoretiskt perspektiv, 2000.
No 766 **Martin V. Howard:** Designing dynamic visualizations of temporal data, 1999.
No 769 **Jesper Andersson:** Towards Reactive Software Architectures, 1999.
No 775 **Anders Henriksson:** Unique kernel diagnosis, 1999.
FiF-a 30 **Pär J. Ågerfalk:** Pragmatization of Information Systems - A Theoretical and Methodological Outline, 1999.
No 787 **Charlotte Björkegren:** Learning for the next project - Bearers and barriers in knowledge transfer within an organisation, 1999.
No 788 **Håkan Nilsson:** Informationsteknik som drivkraft i granskningsprocessen - En studie av fyra revisionsbyråer, 2000.
No 790 **Erik Berglund:** Use-Oriented Documentation in Software Development, 1999.
No 791 **Klas Gäre:** Verksamhetsförändringar i samband med IS-införande, 1999.
No 800 **Anders Subotic:** Software Quality Inspection, 1999.
No 807 **Svein Bergum**: Managerial communication in telework, 2000.
No 809 **Flavius Gruian:** Energy-Aware Design of Digital Systems, 2000.
FiF-a 32 **Karin Hedström:** Kunskapsanvändning och kunskapsutveckling hos verksamhetskonsulter - Erfarenheter från ett FOU-samarbete, 2000.
No 808 **Linda Askenäs:** Affärssystemet - En studie om teknikens aktiva och passiva roll i en organisation, 2000.
No 820 **Jean Paul Meynard:** Control of industrial robots through high-level task programming, 2000.
No 823 **Lars Hult:** Publika Gränsytor - ett designexempel, 2000.
No 832 **Paul Pop:** Scheduling and Communication Synthesis for Distributed Real-Time Systems, 2000.
FiF-a 34 **Göran Hultgren:** Nätverksinriktad Förändringsanalys - perspektiv och metoder som stöd för förståelse och utveckling av affärsrelationer och informationssystem, 2000.
No 842 **Magnus Kald:** The role of management control systems in strategic business units, 2000.
No 844 **Mikael Cäker:** Vad kostar kunden? Modeller för intern redovisning, 2000.
FiF-a 37 **Ewa Braf**: Organisationers kunskapsverksamheter - en kritisk studie av ”knowledge management”, 2000.
FiF-a 40 **Henrik Lindberg:** Webbaserade affärsprocesser - Möjligheter och begränsningar, 2000.
FiF-a 41 **Benneth Christiansson:** Att komponentbasera informationssystem - Vad säger teori och praktik?, 2000.
No. 854 **Ola Pettersson:** Deliberation in a Mobile Robot, 2000.
No 863 **Dan Lawesson**: Towards Behavioral Model Fault Isolation for Object Oriented Control Systems, 2000.
No 881 **Johan Moe:** Execution Tracing of Large Distributed Systems, 2001.
No 882 **Yuxiao Zhao:** XML-based Frameworks for Internet Commerce and an Implementation of B2B e-procurement, 2001.
No 890 **Annika Flycht-Eriksson:** Domain Knowledge Management in Information-providing Dialogue systems, 2001.
FiF-a 47 **Per-Arne Segerkvis**t: Webbaserade imaginära organisationers samverkansformer: Informationssystemarkitektur och aktörssamverkan som förutsättningar för affärsprocesser, 2001.
No 894 **Stefan Svarén:** Styrning av investeringar i divisionaliserade företag - Ett koncernperspektiv, 2001.
No 906 **Lin Han**: Secure and Scalable E-Service Software Delivery, 2001.
No 917 **Emma Hansson:** Optionsprogram för anställda - en studie av svenska börsföretag, 2001.
No 916 **Susanne Odar:** IT som stöd för strategiska beslut, en studie av datorimplementerade modeller av verksamhet som stöd för beslut om anskaffning av JAS 1982, 2002.
FiF-a-49 **Stefan Holgersson:** IT-system och filtrering av verksamhetskunskap - kvalitetsproblem vid analyser och beslutsfattande som bygger på uppgifter hämtade från polisens IT-system, 2001.
FiF-a-51 **Per Oscarsson:** Informationssäkerhet i verksamheter - begrepp och modeller som stöd för förståelse av informationssäkerhet och dess hantering, 2001.
No 919 **Luis Alejandro Cortes:** A Petri Net Based Modeling and Verification Technique for Real-Time Embedded Systems, 2001.
No 915 **Niklas Sandell:** Redovisning i skuggan av en bankkris - Värdering av fastigheter. 2001.
No 931 **Fredrik Elg:** Ett dynamiskt perspektiv på individuella skillnader av heuristisk kompetens, intelligens, mentala modeller, mål och konfidens i kontroll av mikrovärlden Moro, 2002.
No 933 **Peter Aronsson:** Automatic Parallelization of Simulation Code from Equation Based Simulation Languages, 2002.
No 938 **Bourhane Kadmiry**: Fuzzy Control of Unmanned Helicopter, 2002.
No 942 **Patrik Haslum**: Prediction as a Knowledge Representation Problem: A Case Study in Model Design, 2002.
No 956 **Robert Sevenius:** On the instruments of governance - A law & economics study of capital instruments in limited liability companies, 2002.
FiF-a 58 **Johan Petersson:** Lokala elektroniska marknadsplatser - informationssystem för platsbundna affärer, 2002.
No 964 **Peter Bunus:** Debugging and Structural Analysis of Declarative Equation-Based Languages, 2002.
No 973 **Gert Jervan:** High-Level Test Generation and Built-In Self-Test Techniques for Digital Systems, 2002.
No 958 **Fredrika Berglund:** Management Control and Strategy - a Case Study of Pharmaceutical Drug Development, 2002.
FiF-a 61 **Fredrik Karlsson:** Meta-Method for Method Configuration - A Rational Unified Process Case, 2002.
No 985 **Sorin Manolache:** Schedulability Analysis of Real-Time Systems with Stochastic Task Execution Times, 2002.
No 982 **Diana Szentiványi:** Performance and Availability Trade-offs in Fault-Tolerant Middleware, 2002.
No 989 **Iakov Nakhimovski:** Modeling and Simulation of Contacting Flexible Bodies in Multibody Systems, 2002.
No 990 **Levon Saldamli:** PDEModelica - Towards a High-Level Language for Modeling with Partial Differential Equations, 2002.
No 991 **Almut Herzog:** Secure Execution Environment for Java Electronic Services, 2002.

No 999 **Jon Edvardsson:** Contributions to Program- and Specification-based Test Data Generation, 2002.
No 1000 **Anders Arpteg:** Adaptive Semi-structured Information Extraction, 2002.
No 1001 **Andrzej Bednarski:** A Dynamic Programming Approach to Optimal Retargetable Code Generation for Irregular Architectures, 2002.
No 988 **Mattias Arvola:** Good to use! : Use quality of multi-user applications in the home, 2003.
FiF-a 62 **Lennart Ljung:** Utveckling av en projektivitetsmodell - om organisationers förmåga att tillämpa projektarbetsformen, 2003.
No 1003 **Pernilla Qvarfordt:** User experience of spoken feedback in multimodal interaction, 2003.
No 1005 **Alexander Siemers:** Visualization of Dynamic Multibody Simulation With Special Reference to Contacts, 2003.
No 1008 **Jens Gustavsson:** Towards Unanticipated Runtime Software Evolution, 2003.
No 1010 **Calin Curescu:** Adaptive QoS-aware Resource Allocation for Wireless Networks, 2003.
No 1015 **Anna Andersson:** Management Information Systems in Process-oriented Healthcare Organisations, 2003.
No 1018 **Björn Johansson:** Feedforward Control in Dynamic Situations, 2003.
No 1022 **Traian Pop:** Scheduling and Optimisation of Heterogeneous Time/Event-Triggered Distributed Embedded Systems, 2003.
FiF-a 65 **Britt-Marie Johansson:** Kundkommunikation på distans - en studie om kommunikationsmediets betydelse i affärstransaktioner, 2003.
No 1024 **Aleksandra Tešanovic:** Towards Aspectual Component-Based Real-Time System Development, 2003.
No 1034 **Arja Vainio-Larsson**: Designing for Use in a Future Context - Five Case Studies in Retrospect, 2003.
No 1033 **Peter Nilsson:** Svenska bankers redovisningsval vid reservering för befarade kreditförluster - En studie vid införandet av nya redovisningsregler, 2003.
FiF-a 69 **Fredrik Ericsson:** Information Technology for Learning and Acquiring of Work Knowledge, 2003.
No 1049 **Marcus Comstedt:** Towards Fine-Grained Binary Composition through Link Time Weaving, 2003.
No 1052 **Åsa Hedenskog:** Increasing the Automation of Radio Network Control, 2003.
No 1054 **Claudiu Duma:** Security and Efficiency Tradeoffs in Multicast Group Key Management, 2003.
FiF-a 71 **Emma Eliason:** Effektanalys av IT-systems handlingsutrymme, 2003.
No 1055 **Carl Cederberg:** Experiments in Indirect Fault Injection with Open Source and Industrial Software, 2003.
No 1058 **Daniel Karlsson:** Towards Formal Verification in a Component-based Reuse Methodology, 2003.
FiF-a 73 **Anders Hjalmarsson:** Att etablera och vidmakthålla förbättringsverksamhet - behovet av koordination och interaktion vid förändring av systemutvecklingsverksamheter, 2004.
No 1079 **Pontus Johansson:** Design and Development of Recommender Dialogue Systems, 2004.
No 1084 **Charlotte Stoltz:** Calling for Call Centres - A Study of Call Centre Locations in a Swedish Rural Region, 2004.
FiF-a 74 **Björn Johansson:** Deciding on Using Application Service Provision in SMEs, 2004.
No 1094 **Genevieve Gorrell:** Language Modelling and Error Handling in Spoken Dialogue Systems, 2004.
No 1095 **Ulf Johansson:** Rule Extraction - the Key to Accurate and Comprehensible Data Mining Models, 2004.
No 1099 **Sonia Sangari:** Computational Models of Some Communicative Head Movements, 2004.
No 1110 **Hans Nässla:** Intra-Family Information Flow and Prospects for Communication Systems, 2004.
No 1116 **Henrik Sällberg:** On the value of customer loyalty programs - A study of point programs and switching costs, 2004.
FiF-a 77 **Ulf Larsson:** Designarbete i dialog - karaktärisering av interaktionen mellan användare och utvecklare i en systemutvecklingsprocess, 2004.
No 1126 **Andreas Borg:** Contribution to Management and Validation of Non-Functional Requirements, 2004.
No 1127 **Per-Ola Kristensson:** Large Vocabulary Shorthand Writing on Stylus Keyboard, 2004.
No 1132 **Pär-Anders Albinsson:** Interacting with Command and Control Systems: Tools for Operators and Designers, 2004.
No 1130 **Ioan Chisalita:** Safety-Oriented Communication in Mobile Networks for Vehicles, 2004.
No 1138 **Thomas Gustafsson:** Maintaining Data Consistency in Embedded Databases for Vehicular Systems, 2004.
No 1149 **Vaida Jakoniené:** A Study in Integrating Multiple Biological Data Sources, 2005.
No 1156 **Abdil Rashid Mohamed**: High-Level Techniques for Built-In Self-Test Resources Optimization, 2005.
No 1162 **Adrian Pop:** Contributions to Meta-Modeling Tools and Methods, 2005.
No 1165 **Fidel Vascós Palacios:** On the information exchange between physicians and social insurance officers in the sick leave process: an Activity Theoretical perspective, 2005.
FiF-a 84 **Jenny Lagsten:** Verksamhetsutvecklande utvärdering i informationssystemprojekt, 2005.
No 1166 **Emma Larsdotter Nilsson:** Modeling, Simulation, and Visualization of Metabolic Pathways Using Modelica, 2005.
No 1167 **Christina Keller:** Virtual Learning Environments in higher education. A study of students' acceptance of educational technology, 2005.
No 1168 **Cécile Åberg:** Integration of organizational workflows and the Semantic Web, 2005.
FiF-a 85 **Anders Forsman:** Standardisering som grund för informationssamverkan och IT-tjänster - En fallstudie baserad på trafikinformationstjänsten RDS-TMC, 2005.
No 1171 **Yu-Hsing Huang:** A systemic traffic accident model, 2005.
FiF-a 86 **Jan Olausson:** Att modellera uppdrag - grunder för förståelse av processinriktade informationssystem i transaktionsintensiva verksamheter, 2005.
No 1172 **Petter Ahlström**: Affärsstrategier för seniorbostadsmarknaden, 2005.
No 1183 **Mathias Cöster**: Beyond IT and Productivity - How Digitization Transformed the Graphic Industry, 2005.
No 1184 **Åsa Horzella**: Beyond IT and Productivity - Effects of Digitized Information Flows in Grocery Distribution, 2005.
No 1185 **Maria Kollberg**: Beyond IT and Productivity - Effects of Digitized Information Flows in the Logging Industry, 2005.
No 1190 **David Dinka**: Role and Identity - Experience of technology in professional settings, 2005.

No 1191 **Andreas Hansson**: Increasing the Storage Capacity of Recursive Auto-associative Memory by Segmenting Data, 2005.
No 1192 **Nicklas Bergfeldt:** Towards Detached Communication for Robot Cooperation, 2005.
No 1194 **Dennis Maciuszek:** Towards Dependable Virtual Companions for Later Life, 2005.
No 1204 **Beatrice Alenljung:** Decision-making in the Requirements Engineering Process: A Human-centered Approach, 2005.
No 1206 **Anders Larsson:** System-on-Chip Test Scheduling and Test Infrastructure Design, 2005.
No 1207 **John Wilander:** Policy and Implementation Assurance for Software Security, 2005.
No 1209 **Andreas Käll:** Översättningar av en managementmodell - En studie av införandet av Balanced Scorecard i ett landsting, 2005.
No 1225 **He Tan:** Aligning and Merging Biomedical Ontologies, 2006.
No 1228 **Artur Wilk:** Descriptive Types for XML Query Language Xcerpt, 2006.
No 1229 **Per Olof Pettersson:** Sampling-based Path Planning for an Autonomous Helicopter, 2006.
No 1231 **Kalle Burbeck:** Adaptive Real-time Anomaly Detection for Safeguarding Critical Networks, 2006.
No 1233 **Daniela Mihailescu:** Implementation Methodology in Action: A Study of an Enterprise Systems Implementation Methodology, 2006.
No 1244 **Jörgen Skågeby:** Public and Non-public gifting on the Internet, 2006.
No 1248 **Karolina Eliasson:** The Use of Case-Based Reasoning in a Human-Robot Dialog System, 2006.
No 1263 **Misook Park-Westman:** Managing Competence Development Programs in a Cross-Cultural Organisation - What are the Barriers and Enablers, 2006.
FiF-a 90 **Amra Halilovic:** Ett praktikperspektiv på hantering av mjukvarukomponenter, 2006.
No 1272 **Raquel Flodström**: A Framework for the Strategic Management of Information Technology, 2006.
No 1277 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Embedded Systems, 2006.
No 1283 **Håkan Hasewinkel:** A Blueprint for Using Commercial Games off the Shelf in Defence Training, Education and Research Simulations, 2006.
FiF-a 91 **Hanna Broberg:** Verksamhetsanpassade IT-stöd - Designteori och metod, 2006.
No 1286 **Robert Kaminski:** Towards an XML Document Restructuring Framework, 2006.
No 1293 **Jiri Trnka:** Prerequisites for data sharing in emergency management, 2007.
No 1302 **Björn Hägglund:** A Framework for Designing Constraint Stores, 2007.
No 1303 **Daniel Andreasson**: Slack-Time Aware Dynamic Routing Schemes for On-Chip Networks, 2007.
No 1305 **Magnus Ingmarsson:** Modelling User Tasks and Intentions for Service Discovery in Ubiquitous Computing, 2007.
No 1306 **Gustaf Svedjemo**: Ontology as Conceptual Schema when Modelling Historical Maps for Database Storage, 2007.
No 1307 **Gianpaolo Conte**: Navigation Functionalities for an Autonomous UAV Helicopter, 2007.
No 1309 **Ola Leifler**: User-Centric Critiquing in Command and Control: The DKExpert and ComPlan Approaches, 2007.
No 1312 **Henrik Svensson**: Embodied simulation as off-line representation, 2007.
No 1313 **Zhiyuan He:** System-on-Chip Test Scheduling with Defect-Probability and Temperature Considerations, 2007.
No 1317 **Jonas Elmqvist:** Components, Safety Interfaces and Compositional Analysis, 2007.
No 1320 **Håkan Sundblad:** Question Classification in Question Answering Systems, 2007.
No 1323 **Magnus Lundqvist**: Information Demand and Use: Improving Information Flow within Small-scale Business Contexts, 2007.
No 1329 **Martin Magnusson:** Deductive Planning and Composite Actions in Temporal Action Logic, 2007.
No 1331 **Mikael Asplund:** Restoring Consistency after Network Partitions, 2007.
No 1332 **Martin Fransson:** Towards Individualized Drug Dosage - General Methods and Case Studies, 2007.
No 1333 **Karin Camara:** A Visual Query Language Served by a Multi-sensor Environment, 2007.
No 1337 **David Broman:** Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments, 2007.
No 1339 **Mikhail Chalabine:** Invasive Interactive Parallelization, 2007.
No 1351 **Susanna Nilsson:** A Holistic Approach to Usability Evaluations of Mixed Reality Systems, 2008.
No 1353 **Shanai Ardi:** A Model and Implementation of a Security Plug-in for the Software Life Cycle, 2008.
No 1356 **Erik Kuiper:** Mobility and Routing in a Delay-tolerant Network of Unmanned Aerial Vehicles, 2008.
No 1359 **Jana Rambusch**: Situated Play, 2008.
No 1361 **Martin Karresand:** Completing the Picture - Fragments and Back Again, 2008.
No 1363 **Per Nyblom:** Dynamic Abstraction for Interleaved Task Planning and Execution, 2008.
No 1371 **Fredrik Lantz:** Terrain Object Recognition and Context Fusion for Decision Support, 2008.
No 1373 **Martin Östlund:** Assistance Plus: 3D-mediated Advice-giving on Pharmaceutical Products, 2008.
No 1381 **Håkan Lundvall:** Automatic Parallelization using Pipelining for Equation-Based Simulation Languages, 2008.
No 1386 **Mirko Thorstensson:** Using Observers for Model Based Data Collection in Distributed Tactical Operations, 2008.
No 1387 **Bahlol Rahimi:** Implementation of Health Information Systems, 2008.
No 1392 **Maria Holmqvist:** Word Alignment by Re-using Parallel Phrases, 2008.
No 1393 **Mattias Eriksson**: Integrated Software Pipelining, 2009.
No 1401 **Annika Öhgren:** Towards an Ontology Development Methodology for Small and Medium-sized Enterprises, 2009.
No 1410 **Rickard Holsmark:** Deadlock Free Routing in Mesh Networks on Chip with Regions, 2009.
No 1421 **Sara Stymne:** Compound Processing for Phrase-Based Statistical Machine Translation, 2009.
No 1427 **Tommy Ellqvist**: Supporting Scientific Collaboration through Workflows and Provenance, 2009.
No 1450 **Fabian Segelström:** Visualisations in Service Design, 2010.
No 1459 **Min Bao**: System Level Techniques for Temperature-Aware Energy Optimization, 2010.
No 1466 **Mohammad Saifullah**: Exploring Biologically Inspired Interactive Networks for Object Recognition, 2011

No 1468 **Qiang Liu**: Dealing with Missing Mappings and Structure in a Network of Ontologies, 2011.
No 1469 **Ruxandra Pop**: Mapping Concurrent Applications to Multiprocessor Systems with Multithreaded Processors and Network on Chip-Based Interconnections, 2011.
No 1476 **Per-Magnus Olsson**: Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles, 2011.
No 1481 **Anna Vapen**: Contributions to Web Authentication for Untrusted Computers, 2011.
No 1485 **Loove Broms:** Sustainable Interactions: Studies in the Design of Energy Awareness Artefacts, 2011.
FiF-a 101 **Johan Blomkvist:** Conceptualising Prototypes in Service Design, 2011.
No 1490 **Håkan Warnquist:** Computer-Assisted Troubleshooting for Efficient Off-board Diagnosis, 2011.
No 1503 **Jakob Rosén:** Predictable Real-Time Applications on Multiprocessor Systems-on-Chip, 2011.
No 1504 **Usman Dastgeer:** Skeleton Programming for Heterogeneous GPU-based Systems, 2011.
No 1506 **David Landén:** Complex Task Allocation for Delegation: From Theory to Practice, 2011.
No 1507 **Kristian Stavåker**: Contributions to Parallel Simulation of Equation-Based Models on Graphics Processing Units, 2011.
No 1509 **Mariusz Wzorek:** Selected Aspects of Navigation and Path Planning in Unmanned Aircraft Systems, 2011.
No 1510 **Piotr Rudol:** Increasing Autonomy of Unmanned Aircraft Systems Through the Use of Imaging Sensors, 2011.
No 1513 **Anders Carstensen:** The Evolution of the Connector View Concept: Enterprise Models for Interoperability Solutions in the Extended Enterprise, 2011.
No 1523 **Jody Foo:** Computational Terminology: Exploring Bilingual and Monolingual Term Extraction, 2012.
No 1550 **Anders Fröberg:** Models and Tools for Distributed User Interface Development, 2012.
No 1558 **Dimitar Nikolov:** Optimizing Fault Tolerance for Real-Time Systems, 2012.
No 1582 **Dennis Andersson:** Mission Experience: How to Model and Capture it to Enable Vicarious Learning, 2013.
No 1586 **Massimiliano Raciti:** Anomaly Detection and its Adaptation: Studies on Cyber-physical Systems, 2013.
No 1588 **Banafsheh Khademhosseinieh:** Towards an Approach for Efficiency Evaluation of Enterprise Modeling Methods, 2013.
No 1589 **Amy Rankin:** Resilience in High Risk Work: Analysing Adaptive Performance, 2013.
No 1592 **Martin Sjölund:** Tools for Understanding, Debugging, and Simulation Performance Improvement of Equation-Based Models, 2013.
No 1606 **Karl Hammar:** Towards an Ontology Design Pattern Quality Model, 2013.
No 1624 **Maria Vasilevskaya:** Designing Security-enhanced Embedded Systems: Bridging Two Islands of Expertise, 2013.
No 1627 **Ekhiotz Vergara:** Exploiting Energy Awareness in Mobile Communication, 2013.
No 1644 **Valentina Ivanova:** Integration of Ontology Alignment and Ontology Debugging for Taxonomy Networks, 2014.
No 1647 **Dag Sonntag:** A Study of Chain Graph Interpretations, 2014.
No 1657 **Kiril Kiryazov:** Grounding Emotion Appraisal in Autonomous Humanoids, 2014.
No 1683 **Zlatan Dragisic:** Completing the Is-a Structure in Description Logics Ontologies, 2014.
No 1688 **Erik Hansson:** Code Generation and Global Optimization Techniques for a Reconfigurable PRAM-NUMA Multicore Architecture, 2014.
No 1715 **Nicolas Melot:** Energy-Efficient Computing over Streams with Massively Parallel Architectures, 2015.
No 1716 **Mahder Gebremedhin:** Automatic and Explicit Parallelization Approaches for Mathematical Simulation Models, 2015.
No 1722 **Mikael Nilsson:** Efficient Temporal Reasoning with Uncertainty, 2015.
No 1732 **Vladislavs Jahundovics:** Automatic Verification of Parameterized Sytems by Over-Approximation, 2015.
FiF 118 **Camilla Kirkegaard:** Adding Challenge to a Teachable Agent in a Virtual Learning Environment, 2016.
No 1758 **Vengatanathan Krishnamoorthi:** Efficient and Scalable Content Delivery of Linear and Interactive Branched Videos, 2016.
No 1771 **Andreas Löfwenmark:** Timing Predictability in Future Multi-Core Avionics Systems, 2017.
No 1777 **Anders Andersson:** Extensions for Distributed Moving Base Driving Simulators, 2017.
No 1780 **Olov Andersson:** Methods for Scalable and Safe Robot Learning, 2017.
No 1782 **Robin Keskisärkkä**: Towards Semantically Enabled Complex Event Processing, 2017.
No 1783 **Daniel de Leng**: Spatio-Temporal Stream Reasoning with Adaptive State Stream Generation, 2017.
No 1827 **Johan Falkenjack**: Towards a Model of General Text Complexity for Swedish, 2018.
No 1836 **Magdalena Granåsen**: Exploring C2 Capability and Effectiveness in Challenging Environments: Interorganizational Crisis Management, Military Operations and Cyber Defence, 2019.
No 1848 **Alachew Mengist**: Methods and Tools for Efficient Model-Based Development of Cyber-Physical Systems with Emphasis on Model and Tool Integration, 2019.
No 1871 **Klervie Toczé**: Latency-aware Resource Management at the Edge, 2020.
No 1881 **Chih-Yuan Lin**: A Timing Approach to Network-based Anomaly Detection for SCADA Systems, 2020.
No 1886 **August Ernstsson**: Designing a Modern Skeleton Programming Framework for Parallel and Heterogeneous Systems, 2020.
No 1892 **John Törnblom**: Formal Verification of Tree Ensembles in Safety-Critical Applications, 2020.
FiF 128 **Amanda Olmin**: On Uncertainty Quantification in Neural Networks: Ensemble Distillation and Weak Supervision, 2022.
No 1937 **John Tinnerholm**: A Composable and Extensible Environment for Equation-based Modeling and Simulation of Variable Structured Systems in Modelica, 2022.

FACULTY OF SCIENCE AND ENGINEERING

Linköping Studies in Science and Technology, Licentiate Thesis No. 1937, 2022
Department of Computer and Information Science

Linköping University
SE-581 83 Linköping, Sweden

www.liu.se