Linköping Studies in Science and Technology
Thesis No. 1333

Meta-Languages and Semantics for Equation-Based Modeling and Simulation

by

David Broman

Linköping University
INSTITUTE OF TECHNOLOGY

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

Linköping 2010

Copyright notice for Chapter 2 and 6:
© ACM, 2006. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 5th international conference on Generative programming and component engineering, Portland, Oregon, USA , 2006, http://doi.acm.org/10.1145/1173706.1173729
ACM COPYRIGHT NOTICE. Copyright © 2006 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or permissions@acm.org.

Cover page: The first equation with equality sign, by Robert Recorde, 1557 [60].
Watercolor interpretation of the equation by David Broman, 2010.

ISBN 978-91-7393-335-3
ISSN 0345-7524
Thesis No. 1333
October 1, 2010

Electronic version available at:
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-58743
Printed by LiU-Tryck, Linköping 2010

Abstract

Performing computational experiments on mathematical models instead of building and testing physical prototypes can drastically reduce the develop cost for complex systems such as automobiles, aircraft, and powerplants. In the past three decades, a new category of equation-based modeling languages has appeared that is based on acausal and object-oriented modeling principles, enabling good reuse of models. However, the modeling languages within this category have grown to be large and complex, where the specifications of the language's semantics are informally defined, typically described in natural languages. The lack of a formal semantics makes these languages hard to interpret unambiguously and to reason about. This thesis concerns the problem of designing the semantics of such equation-based modeling languages in a way that allows formal reasoning and increased correctness. The work is presented in two parts.

In the first part we study the state-of-the-art modeling language Modelica. We analyze the concepts of types in Modelica and conclude that there are two kinds of type concepts: class types and object types. Moreover, a concept called structural constraint delta is proposed, which is used for isolating the faults of an over- or under-determined model.

In the second part, we introduce a new research language called the *Modeling Kernel Language (MKL)*. By introducing the concept of higher-order acausal models (HOAMs), we show that it is possible to create expressive modeling libraries in a manner analogous to Modelica, but using a small and simple language concept. In contrast to the current state-of-the-art modeling languages, the semantics of *how to use* the models, including meta operations on models, are also specified in MKL libraries. This enables extensible formal executable specifications where important language features are expressed through libraries rather than by adding completely new language constructs. MKL is a statically typed language based on a typed lambda calculus. We define the core of the language formally using operational semantics and prove type safety. An MKL interpreter is implemented and verified in comparison with a Modelica environment.

This research work has been funded by CUGS (the National Graduate School in Computer Science, Sweden), by SSF under the VISIMOD II project, by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project, by the ITEA2 OPENPROD project, by Linköping University under the ELLIIT project, and by the Swedish Research Council (VR).

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

To my lovely wife Åsa and wonderful childen Tove and Hampus

Acknowledgments

First, I would like to express my gratitude to my supervisor Peter Fritzson, who made this thesis possible in the first place by believing in me and enrolling me into the PhD program. You have supported and helped me in many situations during this thesis work.

My greatest thanks to Jeremy Siek who was my host when I was visiting University of Colorado at Boulder as a guest scholar in 2008 and then became my co-supervisor. Your help, energy, and friendliness have been invaluable for me while struggling with semantics, lemmas, and proofs.

I would especially like to thank Thomas Schön, my old friend and research "mentor", for giving me inspiration and advice during this thesis work.

Thanks to all the members of Modelica Association who have taken part in the Modelica design meetings that I have been attending. These discussions have given many ideas to this research. I would also like to thank all colleagues at PELAB for interesting, fun, and sometimes devastating long coffee break discussions. A special thanks goes to Bodil Mattson Kihlström for all your help during these years.

These five years have not just included time for research, but also a large amount of teaching. I would especially like to thank Kristian Sandahl who has been my colleague regarding common teaching efforts in software engineering. Our interesting and sometimes too long discussions have been giving a lot of energy during these years.

During most of the time of this thesis work, I have been living in Stockholm but working in Linköping. Several people have helped me to make this life easier. Thanks to Thomas Sjöland and Björn Lisper for arranging a room at KTH in Kista, Mikael Zayenz Lagerkvist for interesting discussions, and my grandmother Ingrid Broman who has been an almost too friendly host in Linköping.

I would also like to thank the following people for many interesting discussions during the last years: Johan Åkesson, Sébastien Furic, Dirk Zimmer, Hans Olsson, Thomas Doumenc, and Michael Tiller. I would especially like to thank Henrik Nilsson for all rewarding discussions and for inviting me over to Nottingham.

Parts of the final draft of this thesis have been proofread by Jeremy Siek, Thomas Schön, Henrik Nilsson, Walid Taha, Sibylle Schupp, and Hilding Elmqvist. I'm grateful for all comments and suggestions which substantially have improved this work. I would especially like to thank Peter Fritzson for his painstaking effort of reading the whole draft.

Thanks to all my friends and family for all the support you have given me. Thanks to my mother Eva, father Olof, mother-in-law Britt-Marie, and father-in-law Rune for all invaluable help you have given me and my family during the last months. Especially, I would like to express my deepest gratitude to my lovely wife Åsa, who has encouraged me during this time. Without your help, love, and energy, this thesis would never have been completed. I also want to thank my wonderful children Tove and Hampus. During my parental leaves in the fall 2008 and the fall 2010 you have really reminded me that there are other more important things in life than modeling languages.

Finally, I would like to thank the Ethiopian goatherd who according to a legend first discovered coffee. Without this vital beverage, this thesis would never have been finished.

Linköping, August 30, 2010

David Broman

Contents

1 Introduction **1**

1.1 Modeling and Simulation 2

1.1.1 Example of a Mechanical System 3

1.1.2 Importance of Modeling and Simulation 6

1.2 Equation-Based Object-Oriented Languages 6

1.2.1 Domain-Specific Language 7

1.2.2 Objects 8

1.2.3 Mathematical Equations and Acausality 9

1.3 Problem Area 12

1.3.1 Safety Aspects 12

1.3.2 Expressiveness and Extensibility Aspects 14

1.4 Research Questions 14

1.4.1 Understanding the Semantics of the Modelica Language 15

1.4.2 Early Detection of Modeling Errors 15

1.4.3 Expressive and Extensible Formal Semantics 15

1.4.4 Scope 16

1.5 Thesis Outline and Contributions 16

1.5.1 Part I - The Modelica Language 16

1.5.2 Part II - The Modeling Kernel Language 17

1.5.3 Part III - Related Work and Concluding Remarks 18

1.5.4 Published Papers 18

1.5.5 Origin of Contributions 20

1.5.6 Reading Guidelines 21

1.6 Research Method 21

I The Modelica Language **23**

2 Introduction to Modelica **25**
2.1 Equation-Based Modeling in Modelica 26
2.2 The Modelica Compilation and Simulation Process 29
2.3 Chapter Summary and Conclusions 31

3 Specifying the Modelica Language **33**
3.1 Introduction and Motivation 34
3.1.1 Unambiguous and Understandable Language Specification 34
3.1.2 Previous Specification Attempts 34
3.1.3 Abstract Syntax as a Middle-Way Strategy 35
3.2 Specifying the Modelica Language 36
3.2.1 Transformation Aspects - *What* is Actually the Result of an Execution? 36
3.2.2 Checking Aspects - *What* is actually a Valid Modelica Model? . . 37
3.2.3 Specification Approaches - *How* can we State What it's all About? 39
3.3 An Abstract Syntax Specification Approach 40
3.3.1 Specifying the Elaboration Process 40
3.3.2 Specifying the Abstract Syntax 41
3.3.3 The Structure of an Abstract Syntax 42
3.3.4 A Connector S-AST Example with Meta-Variables 42
3.4 Chapter Summary and Conclusions 44

4 Growing the Modelica Language **45**
4.1 Different Ways of Growing a Language 46
4.1.1 The Ways of Growth Matrix 46
4.1.2 Growth by Adding New Language Features 46
4.1.3 Growth by Adding Syntactic Sugar 47
4.1.4 Growth by New Meanings of Annotations or Built-in Functions . 49
4.1.5 Growth by New User Defined Abstractions 50
4.1.6 Restricting the Language 50
4.2 The Right Way to Grow 51
4.2.1 Stakeholders of an Object-Oriented Equation-Based Modeling Language 51
4.2.2 Language Designers' Perspective 51
4.2.3 End Users' Perspective 52
4.2.4 Library Users' Perspective 52
4.2.5 Tool Vendors' Perspective 53
4.3 Chapter Summary and Conclusions 53

5 Types in the Modelica Language **55**
5.1 Types, Subtyping and Inheritance 56
5.1.1 Language Safety and Type Systems 56
5.1.2 Subtyping 58
5.1.3 Inheritance 59
5.1.4 Structural and Nominal Type Systems 61

5.2 Polymorphism . . . 63
5.2.1 Subtype Polymorphism . . . 64
5.2.2 Parametric Polymorphism . . . 65
5.2.3 Ad-hoc Polymorphism . . . 65
5.3 Modelica Types . . . 67
5.3.1 Class Types and Object Types . . . 67
5.3.2 Prefixes in Types . . . 71
5.3.3 Completeness of the Type Syntax . . . 74
5.4 Chapter Summary and Conclusions . . . 74

6 Over- and Under-Constrained Systems of Equations 75
6.1 Problem and Motivation . . . 76
6.2 Featherweight Modelica . . . 77
6.2.1 Syntax and Semantics . . . 78
6.2.2 Type-Equivalence and Subtyping . . . 78
6.3 The Approach of Structural Constraint Delta . . . 80
6.3.1 Algorithms for Computing C_Δ and E_Δ . . . 81
6.3.2 Extending the Type System with C_Δ . . . 87
6.4 Prototype Implementation . . . 87
6.4.1 Constraint Checking of Separately Compiled Components . . . 88
6.4.2 Error Detection and Debugging . . . 90
6.5 Chapter Summary and Conclusions . . . 91

II The Modeling Kernel Language 93

7 Introduction to Functional Programming in MKL 95
7.1 Functional Programming in MKL . . . 96
7.1.1 Higher-Order Functions and Currying . . . 97
7.1.2 Tuples, Lists, and Pattern Matching . . . 101
7.1.3 Equality, Abstract Data Types, and Modules . . . 103
7.2 Lambda Calculus and Operational Semantics . . . 105
7.2.1 Untyped Lambda Calculus . . . 105
7.3 Chapter Summary and Conclusions . . . 108

8 Modeling in MKL 109
8.1 Basic Physical Modeling in MKL . . . 110
8.1.1 A Simple Electrical Circuit . . . 111
8.1.2 Models and Equation Systems . . . 113
8.2 Higher-Order Acausal Modeling . . . 115
8.2.1 Parameterization of Models with Models . . . 116
8.2.2 Recursively Defined Models . . . 117
8.2.3 Higher-Order Functions for Generic Model Composition . . . 119
8.3 Dynamic Data Structures and Polymorphism . . . 121
8.3.1 Model Composition over Lists of Models . . . 121
8.3.2 Parametric Polymorphism . . . 122
8.4 Chapter Summary and Conclusions . . . 123

9 Intensional Analysis of Models **125**
9.1 Models and Unknowns . . . 126
9.1.1 Unknowns . . . 126
9.1.2 Model Type . . . 126
9.1.3 Models as Data Structures . . . 128
9.2 Intensional Analysis of Models . . . 129
9.2.1 Pattern Matching on Models . . . 129
9.2.2 Analyzing Systems of Equations . . . 131
9.3 Chapter Summary and Conclusions . . . 135

10 Semantics of MKL **137**
10.1 Syntax . . . 137
10.2 Type System and Model Lifting . . . 139
10.2.1 Type Consistency . . . 140
10.2.2 Type System . . . 141
10.2.3 Model Lifting . . . 143
10.3 Cast Insertion and Dynamic Semantics . . . 144
10.3.1 Cast Insertion . . . 145
10.3.2 Dynamic Semantics . . . 146
10.3.3 Casts . . . 148
10.4 Type Safety . . . 148
10.5 Extending the Core . . . 151
10.5.1 Other Expressions and the Bot Type . . . 151
10.5.2 Pattern Matching . . . 152
10.5.3 Lifting and Binary Operators . . . 152
10.5.4 Equality . . . 152
10.6 Chapter Summary and Conclusions . . . 153

11 Elaboration Semantics **159**
11.1 Overview of Elaboration . . . 159
11.1.1 Type Checking of Models . . . 160
11.1.2 Collapsing the Instance Hierarchy . . . 161
11.2 Connection Semantics . . . 161
11.2.1 A Minimal Circuit in Modelica . . . 161
11.2.2 A Minimal Circuit in MKL . . . 164
11.2.3 Formalization of the Connection Semantics . . . 168
11.2.4 Composition, and Multiple States . . . 170
11.2.5 Executable Specification . . . 173
11.3 Extracting Model Information . . . 177
11.3.1 Hierarchy Naming vs. Probing . . . 177
11.3.2 Modeling with Probes in MKL . . . 178
11.3.3 Elaboration Semantics of Probes . . . 180
11.4 Chapter Summary and Conclusions . . . 181

12 Implementation, Verification, and Evaluation **183**
12.1 Implementation . . . 183
12.1.1 File Includer and Symbol Table . . . 184
12.1.2 Desugaring . . . 184
12.1.3 Type Checking and Model Translation . . . 185
12.1.4 Program Evaluation after Translation . . . 185
12.2 Uses of Models . . . 186
12.2.1 Exporting the DAE to Flat Modelica . . . 186
12.2.2 Simulating the DAE . . . 187
12.3 Verification . . . 189
12.4 Discussion and Evaluation . . . 191
12.4.1 Safety Aspects . . . 191
12.4.2 Expressiveness and Extensibility Aspects . . . 193
12.4.3 Performance Aspects . . . 195
12.5 Chapter Summary and Conclusions . . . 196

III Related Work and Concluding Remarks **199**

13 Related Work **201**
13.1 Equation-Based Modeling Languages . . . 201
13.1.1 Modelica and Predecessors . . . 201
13.1.2 Extensions to Modelica . . . 202
13.1.3 VHDL-AMS . . . 202
13.1.4 Verilog-AMS . . . 203
13.1.5 gPROMS . . . 203
13.1.6 Hybrid Chi . . . 203
13.1.7 Functional Hybrid Modeling and Hydra . . . 204
13.1.8 Sol . . . 204
13.1.9 Acumen . . . 205
13.1.10 Comparison to MKL . . . 205
13.2 Modelica Semantics . . . 205
13.2.1 Natural Semantics . . . 205
13.2.2 Instance Creation . . . 206
13.2.3 Modelica Types . . . 206
13.2.4 Balanced Models . . . 206
13.2.5 Structural Checking of Models . . . 207
13.3 MKL Semantics . . . 207
13.3.1 Formal Semantics . . . 207
13.3.2 Metaprogramming in EOO Context . . . 208
13.3.3 Metaprogramming in General Purpose Languages . . . 209

14 Concluding Remarks **211**
14.1 Conclusions . . . 211
14.1.1 Understanding the Semantics of the Modelica Language . . . 212
14.1.2 Early Detection of Modeling Errors . . . 212

14.1.3 Expressive and Extensible Formal Semantics . . . 213
14.2 Future Work . . . 214
14.2.1 Extensional Metaprogramming . . . 214
14.2.2 Hybrid and Structural Dynamic Systems . . . 214
14.2.3 Code Generation and Time Aspects . . . 215
14.2.4 Structural Constraint Delta . . . 215
14.2.5 Polymorphism, Type Classes, and Algebraic Data Types . . . 216
14.2.6 Efficient Compilation . . . 216
14.2.7 More Complex Modeling . . . 216
14.2.8 Uses Beyond Simulation . . . 216

A Syntax of MKL **217**
A.1 Concrete Syntax . . . 217
A.1.1 Notational Conventions . . . 217
A.1.2 Comments . . . 217
A.1.3 Lexical Structure . . . 217
A.1.4 Reserved Words . . . 218
A.1.5 Top-Level . . . 218
A.1.6 Types . . . 218
A.1.7 Expressions . . . 219
A.1.8 Pattern Matching . . . 220
A.2 Abstract Syntax . . . 221
A.2.1 Types . . . 222
A.2.2 Expressions . . . 223
A.2.3 Values . . . 223

B Built-in Abstract Data Types **225**
B.1 Array . . . 225
B.2 Set . . . 226
B.3 Map . . . 226
B.4 DAESolver . . . 228

C Big-step Semantics of MKL Core **229**

D MKL Library **231**
D.1 Base . . . 232
D.2 Modeling . . . 233
D.3 Electrical . . . 233
D.4 AnalogElectrical . . . 233
D.5 Mechanical . . . 234
D.6 RotationalMechanical . . . 235
D.7 Elaboration . . . 236
D.8 MechatronicElaboration . . . 239
D.9 Simulation . . . 239
D.10 Export Modelica . . . 241
D.11 Performance Test Source Code . . . 243
D.11.1 MechSys . . . 243

D.11.2 CircuitHierarchy 244

Bibliography **247**

Index **260**

1 Introduction

THIS thesis concerns the problem of designing and defining the semantics of equation-based modeling languages. Such languages, used for mathematical modeling of the dynamics of complex physical systems (e.g., automobiles, aircraft, and powerplants), have in the previous decade gained considerable attention from both industry and academia. This language category is based on the concepts of object-orientation and acausal modeling using equations. This enables good reuse of model components resulting in considerably reduced modeling effort [48]. One such language is Modelica [104], which is an attempt to unify concepts and notation from several earlier languages originating from research projects and industrial initiatives, as well as developing a new language design to address modeling problems. Other examples of languages in this category are gPROMS [13, 115] for chemical engineering and VHDL-AMS [40, 72] a hardware description language (HDL) with analog and mixed-signal extensions.

However, these languages are large and very complex, where the concrete syntax is formally defined using grammars, but the semantics informally described using natural language. The lack of formal semantics makes these languages hard to interpret unambiguously and precisely reason about. A major challenge regarding designing such a complex modeling language is to find a good trade-off between *language safety* (i.e., protect model abstractions by detecting and isolating errors and faults), *performance* (e.g., fast model simulation), *expressiveness* (i.e., ease of expressing complex models and/or tasks), and *extensibility* (i.e., mechanisms to add new language features). The topic of this thesis is the problem of designing and defining language semantics with respect to some of the trade-offs mentioned above.

The rest of the introduction chapter is organized as follows:

- We first give the background of mathematical modeling and simulation together with an overview of how equation-based object-oriented (EOO) languages fit into the picture of domain-specific languages (DSLs) (Section 1.1 and 1.2).

- We discuss the problem area (Section 1.3) and state the research questions. (Section 1.4).
- We present an outline of this thesis together with a summary of the main contributions of the work. We list publications that are part of this thesis and describe the origin of the contributions (Section 1.5).
- Finally, we discuss our scientific viewpoint of the work and the research method used (Section 1.6).

1.1 Modeling and Simulation

Modeling is today a very active area of research in computer science as well as in most disciplines of engineering. The term *model* is used in various settings meaning completely different things, which may unfortunately lead to confusion and misunderstanding regarding the subject. During the recent decades, modeling of software has become very popular; especially in industry. One of the main driving forces is the Model Driven Architecture (MDA) [96] initiative and the popular graphical modeling framework of the Unified Modeling Language (UML) [113, 114].

This thesis does *not* concern modeling or languages used for modeling of software or software systems. Instead, we are primarily interested in languages in which *physical systems* can be described using models. In particular, we are concerned with languages that can support modeling within in a combination of different physical domains, e.g., electrical, mechanical, and hydraulic domains.

To be able to reason about the process of modeling and simulation, some definitions of terms have to be clarified. The following definition was first coined by Marvin Minsky in 1965 [35, p. 5]:

> "A model (M) for a system (S) and an experiment (E) is anything to which E can be applied in order to answer a question about S"

According to this definition, a model can be seen as an abstraction of the system, where some details of the real system is left out. The definition does not imply that the model has to be of a certain kind (e.g., a mathematical formula or computer program), only that experiments should be possible to apply to it to answer questions about the system. However, in this thesis the term model means a *mathematical model* describing dynamic and static properties of a continuous-time system, i.e., a system evolving continuously as a function of time. Several modeling languages also address discrete-time modeling, which however is not covered by this thesis and left as future work.

Many physical systems can be described by *ordinary differential equations (ODEs)* of the form

$$F\big(t, x, \dot{x}, u\big) = 0, \tag{1.1}$$

or in explicit *state-space form*

$$\dot{x} = f\big(t, x, u\big), \tag{1.2}$$

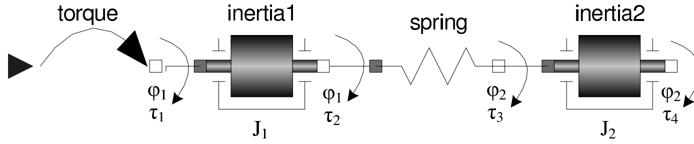


Figure 1.1: *A simple model of a rotational mechanical system representing a drive shaft with a torque.*

where $x \in \mathbb{R}^n$ is the unknown state vector to be solved for, $u \in \mathbb{R}^m$ the vector of input signals, and t the independent variable representing time.

An ODE has a *general solution*, but when studying a model for a specific application it is desirable to find a *unique solution* by also giving the *initial conditions*. The ODE together with the initial conditions is an *initial value problem*:

$$\dot{x} = f\big(t, x, u\big) \tag{1.3}$$

$$x(t_0) = x_0 \tag{1.4}$$

where $x_0 \in \mathbb{R}^n$ is the initial conditions. Note that the dimensions of the vectors x_0 and x are equal.

1.1.1 Example of a Mechanical System

Let us consider a simplified example of a drive shaft for a truck, i.e., the part of a power-train used for transmitting the rotational torque between axles. A graphical model of the shaft is outlined in Figure 1.1 and an example where such a shaft could be used in reality is illustrated in Figure 1.2. The model represents two inertias connected in series, with a spring in between. To the left, a torque is driving the shaft.

Because the inertial bodies are rigid, the angle φ (rad) is the same on each side of the body, here defined as φ_1 and φ_2. However, the torque τ (N m) is different between each component. For example τ_2 is affected both by the driving torque to the left and the conserved energy in the spring.

We define the angular velocities ω_1 (rad/s) and ω_2 together with the equations $\omega_1 = \dot{\varphi_1}$ and $\omega_2 = \dot{\varphi_2}$. By using Newton's law of motion in the rotational domain, we know that the angular acceleration $\dot{\omega}$ (rad/s^2) is proportional to the torque of the shaft, where the proportionality constant is the inertia J (N m/s^2). Hence, we have the equations $J_1 \cdot \dot{\omega}_1 = \tau_1 + \tau_2$ and $J_2 \cdot \dot{\omega}_2 = \tau_3 + \tau_4$ respectively. Because the right hand side of the shaft is not connected, we have $\tau_4 = 0$. The torque affected by the spring is proportional to the angular difference $\varphi_2 - \varphi_1$, where the proportional constant c (N m/rad) is called the spring constant. This adds the equation $\tau_2 = c \cdot (\varphi_2 - \varphi_1)$ to the system of equations. We also know that the spring torque is the same on each side of the spring, but with different sign, i.e., $\tau_2 = -\tau_3$. Finally, we also have the input torque u giving $u = \tau_1$.

We now have a system of equations with 8 equations and 8 unknowns (φ_1, φ_2, ω_1, ω_2, τ_1, τ_2, τ_3, τ_4), where four unknowns appears differentiated ($\dot{\varphi_1}$, $\dot{\varphi_2}$, $\dot{\omega}_1$, $\dot{\omega}_2$).

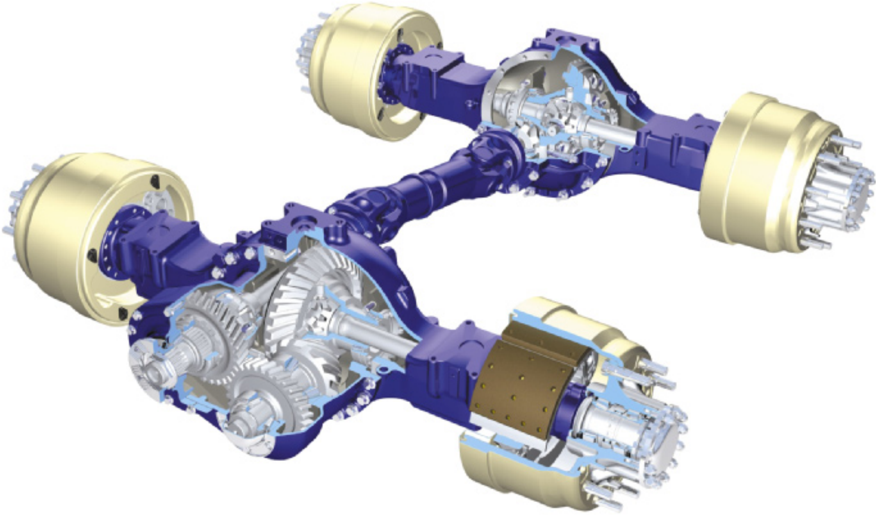


Figure 1.2: *The figure shows Tandem axles RST 2370 A B-Ride Bogie (Volvo Trucks). The shaft between the axles is an example of a rotating shaft that is part of a powertrain for transmission of the torque. Used with permission.*

We can rewrite our example as follows:

$$\dot{\varphi}_1 = \omega_1 \tag{1.5}$$
$$\dot{\varphi}_2 = \omega_2 \tag{1.6}$$
$$\dot{\omega}_1 = \frac{\tau_1 + \tau_2}{J_1} \tag{1.7}$$
$$\dot{\omega}_2 = \frac{\tau_3 + \tau_4}{J_2} \tag{1.8}$$
$$\tau_1 = u \tag{1.9}$$
$$\tau_2 = c \cdot (\varphi_2 - \varphi_1) \tag{1.10}$$
$$\tau_3 = -c \cdot (\varphi_2 - \varphi_1) \tag{1.11}$$
$$\tau_4 = 0 \tag{1.12}$$

Here the last four equations (1.9-1.12) are called *algebraic equations.*

Recall the definition of an ODE (1.1) where all variables except the independent variable appears differentiated. In the mechanical example above, variables τ_1, τ_2, τ_3, τ_4 do not appear differentiated. These variables are called *algebraic* meaning that they are free from derivatives. Hence, our system of equations is not an ODE, but a system of

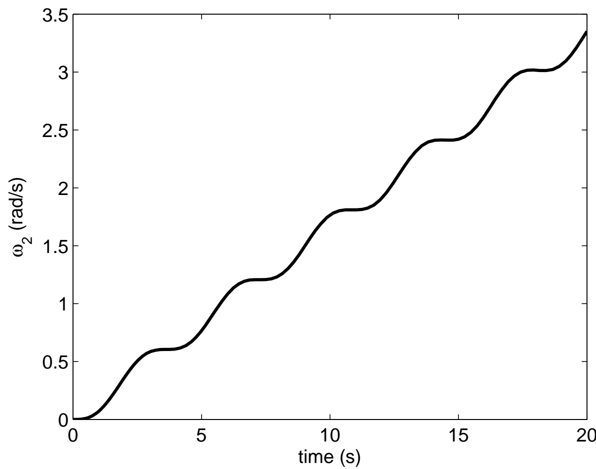


Figure 1.3: *Plot of the angular velocity ω_2 of the shaft example.*

differential algebraic equations (DAE). The general form of a DAE is

$$F(t, x, \dot{x}, y, u) = 0, \tag{1.13}$$

where t is the independent variable of time, x a vector of variables that appear differentiated, y a vector of algebraic variables, and u a vector of input signals.

In this simple example, the algebraic equations can be directly eliminated by substitution into equations (1.7) and (1.8), thus forming an ODE. However, this is not possible in the general case and there are sophisticated methods described in the literature for solving DAEs numerically and symbolically [83, 118]. Differential-algebraic equations is the kind of equation system used in equation-based languages discussed in this thesis, such as Modelica, for describing continuous-time behavior.

We have in this example shown how we describe a mathematical model of a mechanical system. We can now use our model to answer questions about the system, using experiments. This can be performed using *simulation*, or as stated by Granino Korn and John Wait according to Cellier [35, p. 6],

> "A simulation is an experiment performed on a model"

Hence, we can simulate our example model to study the behavior of the physical system.

Assuming that we know the parameters of the system (J_1, J_2, and c) and that we have a known input signal u, we can simulate the system using a numerical integration algorithm to solve the system of differential equations[1].

Figure 1.3 shows an example where the angular velocity ω_2 has been be plotted for the interval 0 to 20 s. The plot shows how the shaft starts to oscillate due to flexibility introduced by the spring.

[1] In the example, we assign $J_1 = 10$ kg m^2, $J_2 = 2$ kg m^2 and $c = 5$ N m/rad. We also let the input signal u be a constant value of 2 N m. These values do not represent a shaft for a powertrain in reality; it is used for the purpose of showing a clear oscillation.

1.1.2 Importance of Modeling and Simulation

Why is modeling and simulation of physical systems important? Before we discuss this question, let us define *the process of modeling* by quoting Cellier and Kofman [37, p. 8]:

> "The process of *modeling* concerns itself with the extraction of knowledge from the physical plant to be simulated, organizing that knowledge appropriately, and representing it in some unambiguous fashion."

Modeling and simulation is perhaps one of the most common techniques for answering questions by scientists and engineers. While scientists are focused on understanding and observing the world, engineers primarily want to design new artifacts. In both cases *modeling* is a central process for abstracting, extracting, and organizing the knowledge for further analysis.

There are many reasons why modeling and simulation is beneficial. For example:

- It is typically much *cheaper* to perform experiments on the model compared to performing them directly on the real system. For example, when developing a control system for a landing gear of an aircraft, several engineers can test their control system simultaneously by simulating a model of the landing gear, instead of using direct access of a physical prototype.
- It might be *too dangerous* to do the experiments in reality. When testing "what-if" scenarios on a nuclear power-plant, it is safer to do these experiments on a mathematical model compared to a real plant.
- The system *may not exist*, i.e., the model is a prototype that is evaluated and tested during development. Most product development cycles still need physical prototypes for evaluation, but by using a combination of virtual prototypes of mathematical models the development time can potentially be dramatically shortened.
- Some unknown variables *are not accessible* in the real system, but can be observed in a simulation. For example, measuring the temperature inside certain areas of an engine can be physically impossible without affecting the engine's behavior.
- It is *easy to use* and modify models, to change parameters and perform new experiments (simulations). For example, it is much easier to experiment with and change the size of wind turbines on a model than on physical prototypes.

However, as pointed out both by Cellier [35] and Fritzson [51], the ease of use is also the main danger and drawback with modeling and simulation. There is a risk to ignore the fact that the model is only valid under certain conditions, and that the model is in fact an abstraction of the reality and not the reality itself. Consequently, care must be taken regarding which simulations are suitable to apply on a model, so that the results reach the desired level of accuracy.

1.2 Equation-Based Object-Oriented Languages

In the previous section we gave an introduction to continuous-time system modeling and simulation. Designing languages for continuous-time systems is not new and one of the

earliest initiatives was the *Continuous System Simulation Language (CSSL)* specified in 1967 [11]. Derivations of CSSL are all based on state-space descriptions where the underlying mathematical description is an ODE [35]. General-purpose simulation tools, e.g., Simulink [92], using block diagrams and causal connections, have now dominated the area for many years. Block diagrams make it possible to graphically model ODEs and the software tool is then used for performing the numerical simulation.

In the 1960's, the first object-oriented language was designed with the initial purpose of *discrete event-based modeling* and simulation. This language, Simula [44], founded the fundamental concepts of object-orientation and object-oriented languages. The fundamental principles of *object-oriented modeling languages* for continuous-time modeling and simulation have been around for about 30 years. According to Cellier [36], this started with the pioneering work explored in two separate PhD theses by Hilding Elmqvist [47] and Tom Runge.

Several languages have been developed during the years with the common properties of physical modeling using equation systems. Today the state of the art within multi-domain physical modeling (e.g., containing mechanical, electrical, hydraulic, thermal, fluid, and control components) is Modelica [104]. Other examples of languages with similar modeling and simulation capabilities are gPROMS [13, 115] for chemical engineering and VHDL-AMS [40, 72] a hardware description language (HDL) with analog and mixed-signal extensions.

However, not until recently has a common name for this category of languages appeared. We call this language category *equation-based object-oriented (EOO) languages*[2]. The exact meaning of this name can be a subject for discussion, but we propose the following definition:

Definition 1.2.1 (Equation-based object-oriented (EOO) language).
An equation-based object-oriented (EOO) language is a domain-specific language used for modeling the interaction between objects, by utilizing mathematical equations to provide an acausal description of behavior.

This informal definition includes the following vague terms:

- Domain-specific language
- Objects
- Mathematical equations and acausality

In the rest of this section, we will discuss and clarify these terms.

1.2.1 Domain-Specific Language

A domain-specific language (DSL) can, according to van Deursen *et. al.* [148, p. 26], be defined as follows:

> "A *domain-specific language* (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

[2] The term was coined by the author of this thesis and first publicly used at a poster session at the conference on programing language design and implementation (PLDI) in 2006 [22].

The vague defining term of this definition is the term *problem domain*. In [148], van Deursen *et. al.* implicitly defined the term by giving examples of existing DSLs within areas such as software engineering, systems software, multi-media, telecommunication, and miscellaneous domains such as simulation, robot control and solving partial differential equations. Other authors, such as Czarnecki and Eisenecker [43, p. 34], define this as follows:

> "Domain: An area of knowledge
>
> - Scoped to maximize the satisfaction of the requirements of its stakeholders
> - Includes a set of concepts and terminology understood by practioners in the area
> - Includes the knowledge of how to build software systems (or parts of software systems) in that area"

From this point of view, we can for example regard both Modelica, gPROMS, and VHDL-AMS as domain-specific languages, with their expressive power focused on modeling of physical systems. However, according to van Deursen *et. al.* [148], DSLs are usually *small* languages with a restricted set of notations and abstractions. In fact, DSLs are sometimes referred to as *little languages* [147], compared to larger *general-purpose languages (GPLs)*. Can we regard Modelica with an informal language specification of 250 pages [104] as a small language? Moreover, Modelica is commonly referred to as a multi-domain modeling language. How can this be regarded as a domain-specific language?

As with all informal definitions, it depends on the interpretation of the terms - in this case the word domain. We regard for example Modelica and VHDL-AMS as *large* domain-specific languages with regards to the domain of modeling physical systems. This holds especially when comparing to a GPL, in which arbitrary computational tasks can be described. Moreover, we may also regard particular libraries defined in these languages as specialized sub-domain-specific languages, e.g., the Modelica Bond Graph library [38] or the Fluid library [105]. With this view, the Modelica design group that is designing the language are the *domain experts* of physical modeling and the designers of a particular library the *sub-domain* experts for a particular physical domain.

1.2.2 Objects

The term *object-oriented* in EOO is not used with exactly the same meaning as for the common *object-oriented programming (OOP)* languages. In for example Smalltalk [64] an object is an instance of class that can send or receive messages. In C++ [75, 139], behavior is described by invoking methods associated with an object. Somewhat simplified, we might say that an object in OOP can be described be the following equation

$$\text{object}_{\text{OOP}} = \text{data} + \text{methods}$$

Similar to OOP languages, objects in EOO languages are used for describing the combination of data and behavior. In contrast to OOP, the behavior of the objects in EOO languages is described by mathematical equations instead of methods or message passing. Hence, one view of an object in EOO would be

$\text{object}_{\text{EOO}}$ = data + equations

Many EOO languages, such as Modelica and Omola [6] use language mechanisms from OOP languages (for example inheritance and subtyping-polymorphism), but we do not regard this as a necessary condition for being an EOO language. There are many concepts related to OOP languages and as shown by Armstrong [7] there is no clear consensus of what actually defines the core concepts of OO languages.

Recall Figure 1.1 showing a graphical Modelica model of a rotational mechanical system. Objects are in Modelica referred to as *components*. This mechanical model has four components (objects): a torque, two inertias, and a spring. We say that objects are instances of *EOO models* (or *Modelica models*). Moreover, an EOO model can compose and encapsulate one or more model instances. For example, the objects `inertia1` and `inertia2` are instances of a common EOO model representing the general behavior of an inertia. When the objects are created, they are given different inertia values, J_1 and J_2 respectively.

Objects are connected via *ports* (called *connectors* in Modelica). In Figure 1.1, the object `intertia1` is connected to `torque` and `spring`. The object `intertia2` is connected to the `spring` on its left hand side and is *unconnected* on the right hand side.

In state-of-the-art EOO languages, objects are used only for hierarchically compose EOO models, i.e., objects are not created dynamically during simulation. However, this is an active area of research called *structurally dynamic systems* [62, 155].

We shall note one thing regarding terminology. As stated in the beginning of Section 1.1, we use the term *model* with the general meaning of a mathematical model, i.e., a system of equations. When it is clear from the context, the term *model* may either refer to an EOO model or the underlying equation system represented by the EOO model.

1.2.3 Mathematical Equations and Acausality

The foundation of EOO languages is that behavior is described declaratively using mathematical equations. Even though most EOO languages describe behavior using DAEs (e.g., Modelica and VHDL-AMS), the behavior could also be described by *partial differential equations (PDEs)* or by equational constraints for model-based diagnosis [30]. The main point is that the equations are acausal (also called non-causal), meaning that the causality of how to solve the equations is not decided at modeling time. Acausality should be present at two levels of abstraction:

- at the equation-level
- at the object connection level

We say that a system of equations is acausal, if the order in which the unknowns are solved is not decided at modeling time. Consider for example the equation of Ohm's law

$$v = R \cdot i,$$

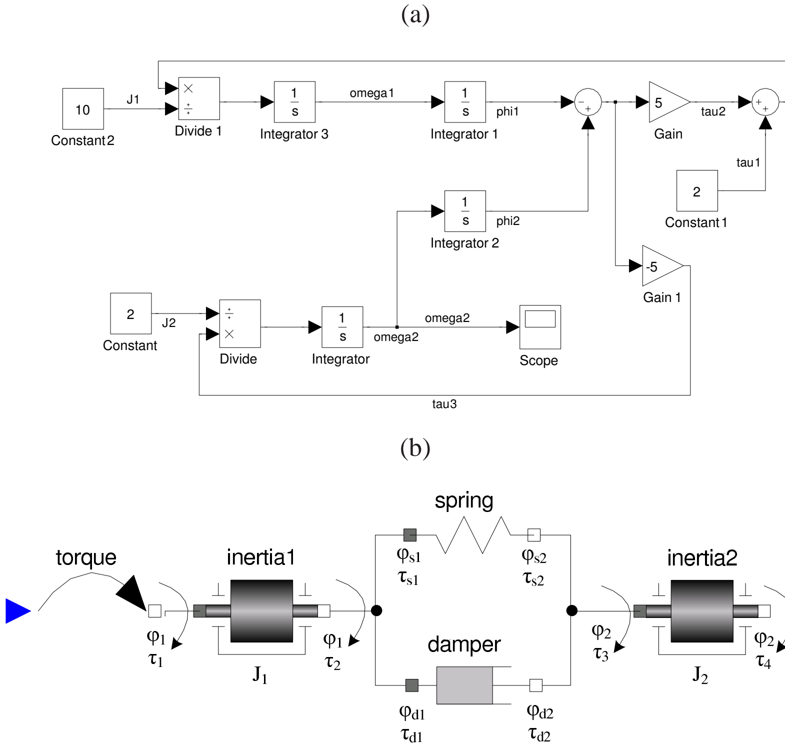


Figure 1.4: *Model (a) is a causal block diagram model of the mechanical system in Figure 1.1. Model (b) shows how the model in Figure 1.1 has been reused and the original spring replaced with a parallel spring-damper.*

where v is the voltage, R the resistance, and i the current. Depending on which variable is unknown, it can be translated into three different assignment statements

$$v := R \cdot i,$$
$$i := v/R,$$
$$R := v/i,$$

Acausality at the object connection level is the second central part of acausality of EOO languages. Recall the mechanical system in Section 1.1.1, where we used the graphical model as illustration for the rotational system when performing the equational modeling by hand. However, Figure 1.1 is actually the graphical representation of an executable Modelica model. One of the main benefits with an acausal model such as this one is that the topology corresponds to how objects in the physical world would be connected. This is referred to as *physical modeling* [51]. A causal model of the same mechanical system is given in Figure 1.4a. The causal model consists of *blocks* with defined input and output. Compared to the acausal model, the physical topology is lost.

Now, assume that we reuse the model in Figure 1.1 and replaces the spring object with a spring-damper object. The resulting model is shown in Figure 1.4b. Because the model

is acausal, a simulation tool can automatically generate an new equation system for the updated model. However, if the block model is changed to include a spring-damper, the large parts of the diagram needs to be rearranged, because the block diagram is dependent on the causality of the underlying equation system.

The key to acausal physical models is in the basic physical principle of *conservation of energy*, stating that the total amount of energy in a closed system is constant over time; it can neither be created nor destroyed. To support this principle, acausal ports have both a *potential variable* (also called *across*) and a *flow variable* (also called *through*). This principle of potential/flow variables is applicable in several different physical domains, e.g., [104]:

Domain	**Potential Variables**	**Flow Variables**
Electrical Analog	Electrical Potential	Electrical Current
Translational Mechanics 1D	Distance	Force
Rotational Mechanics 1D	Angle	Torque
Heat Transfer	Temperature	Heat Flow Rate
Magnetic	Magnetic Potential	Magnetic Flux

If two or more ports are connected, the potential variables are set equal, whereas the flow variables are summed-to-zero. In the electrical domain the sum-to-zero principle corresponds to Kirchhoff's current law. For example, consider the connections between the `spring`, the `damper`, and the `inertia2` object in Figure 1.4b. The following equality equations are generated for the potential variables:

$$\varphi_{s2} = \varphi_2$$
$$\varphi_{d2} = \varphi_2$$

Note that no redundant equation is generated between φ_{s2} and φ_{d2}. Finally, the sum-to-zero equation is generated:

$$\tau_{s2} + \tau_{d2} + \tau_3 = 0$$

The phase of generating equations from the description of connections between ports is referred to as *connection semantics*. This phase is in turn part of the *elaboration process*[3], where an EOO model is translated into a set of equations. The other two essential phases in this elaboration process are *type checking* (deciding which models that are considered correct according to a defined type system) and *collapsing the instance hierarchy* (translating components into a system of equations).

The generated equations from the elaboration process forms the resulting DAE of the EOO model. The process of a general translation from a DAE to an ODE (or to a so called index-1 DAE) is the result of extensive research and involves symbolic manipulation of

[3]In this thesis, we call the process *elaboration*. In the Modelica specification 3.2, this process is called *flattening* because it creates a flat system of equations. Sometimes, this translation is also referred to as *instantiation*. However, we think that both these terms are misleading. The former, because the final equation system does not need to be flat - it can still be represented in a hierarchical structure. The latter, because it is typically performed at compile time and is not allocating memory analogous to instance creating in standard programming languages.

the equation system. Key aspects of this process are the use of Pantelides algorithm [118], block lower triangular form (BLT) [70, 71], dummy-derivatives [94], and tearing [49]. The details of this translation falls outside the scope of this thesis, but are central to the performance and accuracy of implementations of Modelica software tools.

1.3 Problem Area

This thesis concerns the problem of designing and defining the semantics of equation-based modeling languages. By *semantics* we mean the meaning of a model. This includes both the static semantics (translating an EOO model into an equation system and rejecting illegal models) and dynamic semantics (to use the equation system, i.e., for numerical simulation). A major challenge regarding the design of such complex languages is to find a good trade-off between different quality and design aspects of the language. In this section we give an overview of the problem area with regards to the following aspects:

- *Safety aspects* - to protect model abstractions by detecting and isolating errors and faults.
- *Expressiveness and extensibility aspects* - makeing it easy to express complex modeling tasks and to provide mechanisms for extending the language with new features.

1.3.1 Safety Aspects

It is not always possible to simulate an EOO model because the model might have been incorrectly specified. Furthermore, even if a simulation result is generated, this does not imply that the result is correct, i.e., that the simulation result corresponds to the real system. We will in the first section outline the overall problems and challenges regarding safety aspects of EOO languages and their environments,

By following the terminology defined in the IEEE Standard 100 [107], we define an *error* to be something that is made by human beings. As the consequence of an error, a *fault* exists in an artifact, such as an EOO model, source code, or a language specification. Another word for fault would be bug or defect. If a fault is executed, this results in a *failure*, i.e., it is possible to detect that something went wrong.

People make mistakes, i.e., make errors when modeling systems. This can result in either incorrect simulation results, or no results at all. To produce products (e.g., aircraft, cars, and factory machines) based on incorrect simulation results, can be very expensive or even result in devastating consequences. Hence, it is of great importance to efficiently handle errors in a safe manner.

There are many different sources of errors in a modeling and simulation environment. Consider Figure 1.5, which outlines relations between sources of errors and faults. The center box illustrates the simulation tool, which takes an EOO model as input (left side) and produces a *simulation result* if the simulation was successful, or a *simulation failure report* if an error occurs during simulation. In the model, there are three actors that can produce errors that affect the tool's output.

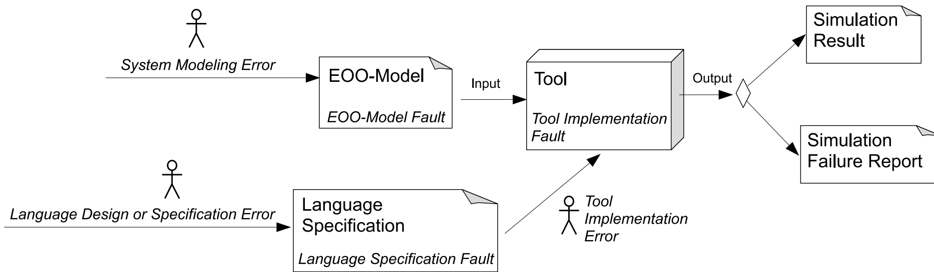


Figure 1.5: *Relations between possible errors and faults in a modeling and simulation environment.*

System Modeling Errors. A *system modeling error* can result in an EOO model containing an *EOO model fault*, which obviously affects the simulation result. Some modeling errors can result in failures already in the elaboration phase (e.g., illegal access of elements in objects or wrong number of equations), while others result in simulation failures during simulation (e.g., numerical singularities). Moreover, an engineer can make mistakes while modeling a system, which still gives simulation results, but perhaps incorrect values. In this thesis, we are mainly concerned with modeling errors that can result in a failure during the elaboration phase, i.e., before simulation.

Language Design and Specification Errors. Almost all commonly used languages evolve over time, resulting in high demands on the language design effort and the work to produce precise, consistent, and error free language specifications. The Modelica language is no exception, which has resulted in a large and complex language with an informal specification using plain text. This fact can lead to *language design errors* because it is hard to grasp the complete semantics of the language. For example, it could be a fault in the specification of the type system if models containing faults can be executed, although they should have been rejected by the type checker. Moreover, if the language design effort intends to give guarantees that a certain kind of modeling error should be detected, it is obviously necessary that the specification is precise and easy to reason about. Hence, one of the main challenges is to be able to define this kind of languages in a precise way, using formal semantics.

Tool Implementation Errors. In addition, language specification faults and unclear semantics may lead to *tool implementation errors*. If only one tool exists for the language, the importance of implementation errors compared to the specification might be ignorable. However, if there exist several tools, tool implementation errors may lead to incompatible models or even non-deterministic simulation results. For example, Modelica has a complex semantics for name lookup that has in our experience lead to considerable effort to make the OpenModelica [53, 117] compiler compatible with other commercial tools of Modelica, primarily Dymola [45].

To mitigate the fact that people make errors, we see three major challenges regarding error handling:

1. ***Detecting* the existence of an error early.** If a simulation fails, it is trivial to detect that an error must exist. However, if a simulation takes 48 hours to complete, it is not desirable to wait 46 hours before the error is detected.
2. ***Isolating* the fault implied by the error.** If we have detected that an error must exist, how do we know where the actual fault is located? Is it located in the main model, in some model library, or even in the simulation tool itself? For example, if an engine is modeled and then translated into a system of equations containing 20000 equations and 20001 unknowns, it is trivial to detect that this is a fault. However, it is a non-trivial task to isolate the fault so that the error can be resolved.
3. ***Guaranteeing* that faults do not exist.** If we can detect an error by using e.g., *testing* and then isolate the fault using some kind of *debugging* technique, how do we know that there do not exist any other errors? Consequently, would it be possible to give guarantees that some kind of fault cannot exist in a model, e.g., that a specific type of errors will always be detected?

1.3.2 Expressiveness and Extensibility Aspects

State of the art EOO languages, such as Modelica, are expressive regarding the ability to define new models within existing libraries. However, new libraries of the Modelica standard library (MSL) are often released together with a new version of the Modelica language that supports new language constructs required for the new library. Examples include the handling of over-determined connectors used in the multi-body library and stream connectors for the Fluid library. This leads to a continuously growing the size and complexity of the language. The challenge is to make the EOO language *expressive* enough so that the language does not need to be extended when a new libraries are released.

Software tools and compilers can *use* the EOO models for different purposes. One such purpose, which today is the most common use, is to analyze the equation system, transform it, generate executable code, and simulate the system. However, there are several other potential uses of mathematical models, such as optimization problems [79], model reduction, generation of real-time simulation code, and exporting data to standardized model formats [34]. The problem is that an EOO language has to be extended with additional language constructs to support the new use, leading to either a number of new languages with specific extensions, or to an EOO language that is even larger and more complex. Hence, the challenge is to make the EOO language *extensible* so that the language does not have to be updated if a model is used in a new manner.

1.4 Research Questions

From the description of the problem area in Section 1.3, a number of research questions are formulated below. We categorize the questions within three areas:

- Understanding the Semantics of the Modelica Language
- Early Detection of Modeling Errors
- Expressive and Extensible Formal Semantics

1.4.1 Understanding the Semantics of the Modelica Language

Both the dynamic and static semantics of the Modelica language are informally described using natural language and source code examples. Because the language has grown to be very large and complex, it is hard in the short term to define a formal semantics for the complete language; leading to the following question:

Research Question 1. How can an informal language specification be restructured to be less ambiguous and still understandable?

A common way of statically detecting and isolating errors in a language is to use type checking. However, in Modelica, the concept of types is only implicitly described using informal natural language. Hence, our second question in the study concerns Modelica types.

Research Question 2. What is the actual meaning of types in Modelica and how does this compare to the class concept in the language?

1.4.2 Early Detection of Modeling Errors

If a model is incorrectly described and contains more equations than unknowns (over-determined) or fewer equations than unknown (under-determined), it is easy to detect the error after elaboration by just counting the number of equations and variables. However, it is much harder to isolate the error to a specific model instance. Earlier approaches have tried to analyze the flat system of equations after elaboration, and then tracing back the faults to the original models [29], leading to the following question:

Research Question 3. Is it possible to define an approach to detect under- and over-constrained errors in the model *before* elaboration, enabling the user to *isolate* the fault to a certain model instance?

1.4.3 Expressive and Extensible Formal Semantics

State of the art EOO languages are large and complex with the semantics informally described. Programming language researchers have for decades formalized languages based on small and expressive calculi, where the lambda calculus [12] is the prominent one.

Research Question 4. Is it possible to formally define an EOO language as an extension of a typed lambda calculus that gives the expressive power of state of the art EOO languages?

As described in the problem area description in Section 1.3, a major problem for language extensibility is that new modeling demands often require changes in the language specification of the EOO language. This requires both new language revisions and that different vendors need to update their modeling and simulation products. This process is both time consuming and error prone and limits the possibility for the domain experts to express their modeling needs.

Research Question 5. Can we design a modeling kernel language where domain experts can extend both modeling constructs for an EOO language, as well as the semantics for using the models?

1.4.4 Scope

The research questions stated in the previous sections are broad and therefore the following scope is given for the thesis:

- For domain experts of EOO languages who are developing libraries, the concrete syntax is of essential importance. However, in this thesis we do not make any attempt of analyzing how or which syntax is most convenient for domain experts.
- EOO languages typically have both a textual and a graphical representation. Even though we acknowledge the need and benefits of graphical syntax, we are only analyzing the textual representation in this thesis.
- Performance aspects of the proposed solution are only analyzed at a high level of abstraction because our prototype is implemented as an interpreter and not as a compiler.
- We investigate the formal semantics of expressing models, for elaborating structured models down to equation systems, and for analyzing the equation system. The semantics for the model compiler backend concerning symbolic manipulation and solving the DAEs are outside the scope of the thesis.
- We are primarily concerned with the continuous-time aspects of EOO languages and we will not discuss hybrid models (the mixture of continuous-time and discrete-time models) in this thesis.
- We will not attempt to formally define an existing EOO language, e.g., Modelica.

1.5 Thesis Outline and Contributions

The thesis is divided into two main parts. Part I relates to the Modelica language and in Part II we propose a new research language called the *Modeling Kernel Language (MKL)*.

In the following, we give an overview of the contents of the thesis as well as a statement of the main contributions. For each of the contributions, a note is given for where the contribution is discussed and detailed, as well as a pointer to which research question the contribution relates to.

1.5.1 Part I - The Modelica Language

Part I is dedicated to studying the Modelica language - both with regards to interpreting and understanding the current semantics of the language, as well as proposing extensions and improvements to the language. In Part I we make the following main contributions:

- We discuss different strategies for specifying the Modelica language as well as suggesting approaches that make it easier to extend and grow the language in the future. This work does not present any specific technical contribution, but instead presents a categorization and discussion about the Modelica specification (Chapters 3 and 4, Research Question 1).

- We give to the best of our knowledge, the first description and interpretation of the type concept in Modelica to the level of precision that makes it clear that Modelica has two categories of types: class types and object types. The corresponding paper was published in 2006 [27] and has influenced the improved description of the Modelica specification (Chapter 5, Research Question 2).

- We propose an approach for determining if a model is under- or over-constrained without elaborating its subcomponents. The main insight is the idea of annotating the constraint value on types. We call the approach *structural constraint delta*, denoted C_{Δ}, and define an algorithm for a subset of the Modelica language. However, this approach is not limited to Modelica and should be useful in other EOO languages as well. The work was published in 2006 [28] and somewhat influenced the design of balanced models, part of the Modelica 3.0 standard from 2007 (Chapter 6, Research Question 3).

1.5.2 Part II - The Modeling Kernel Language

Part II concerns the problem of creating a formally defined language that is both expressive and extensible (Research Question 5 and 4).

The proposed research language MKL is not intended as a new end-user EOO language. It is a kernel language in which domain experts and language designers can define EOO language constructs within MKL libraries. Also, the main objective is not to translate available EOO languages to MKL. Instead, we see MKL as a research language for exploring new language constructs and principles, which can then later perhaps find their way into available EOO languages.

In Part II we make the following specific contributions:

- We explore the concept that we call *higher-order acausal models (HOAMs)*, which is the combination of higher-order functions and acausal models. We show that HOAMs enable great expressive power with few required language concepts (Chapter 8, Research Question 4).

- We show how the MKL language is capable of performing intensional analysis on models, i.e., inspect and traverse the equation system (Chapter 9, Research Question 5).

- We define a formal operational semantics and related static type system for the core of MKL. The language is an extension of a typed lambda calculus and forms the foundation of the MKL language. We prove type safety of the core language. (Chapter 10, Research Question 5).

- We formally define the elaboration semantics of an EOO language, i.e., the translation process from an EOO model to an equation system. We describe both an

approach for the connection semantics as well as a solution for the problem of extraction simulation results. (Chapter 11, Research Question 4).

- We explain our prototype implementation of MKL, define the semantics of simulating a model using an external DAE solver, as well as exporting the DAE to a *flat Modelica model*, i.e., a model with only equations and no components. Finally, we verify, discuss, and evaluate our solution. (Chapter 12, Research Question 5).

1.5.3 Part III - Related Work and Concluding Remarks

In the final part, we do not make any new contributions. Instead our work is compared to related work (Chapter 13). Finally, we state the conclusions of the thesis and outline future work (Chapter 14).

1.5.4 Published Papers

The research results given in this thesis are partially based on the following published papers and reports:

Journal Paper

- David Broman and Peter Fritzson. Higher-Order Acausal Models. Simulation News Europe 19(1):5-16, ARGESIM, 2009

Peer Reviewed Conference and Workshop Papers

- David Broman and Peter Fritzson. Higher-Order Acausal Models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 59-69, Paphos, Cyprus, LIU Electronic Press, 2008 (superseded by the journal version)

- David Broman and Peter Fritzson. Abstract Syntax Can Make the Definition of Modelica Less Abstract. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools.* pages 111-126. Berlin, Germany. Linköping University Electronic Press. 2007

- David Broman, Kaj Nyström, and Peter Fritzson. Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06).* pages 151-160. Portland, Oregon, USA. ACM Press. 2006

- David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference.* pages 303-315. Vienna, Austria. 2006

Invited Paper

- David Broman. Growing an Equation-Based Object-Oriented Modeling Language. In *Proceedings of MATHMOD 09*, Vienna, pages 1316-1324, Vienna, Austria, 2009

Technical Reports and Thesis

- David Broman. Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments. *Licentiate thesis.* Thesis No 1337. Department of Computer and Information Science, Linköping University, December, 2007

- David Broman. Flow Lambda Calculus for Declarative Physical Connection Semantics. *Technical Reports in Computer and Information Science No. 1*, Linköping University Electronic Press. 2007

Papers and Reports not Included in the Thesis

The following papers and reports are of related interest, but not directly included in this thesis. The papers were authored or co-authored during the period of this thesis work.

- Peter Aronsson and David Broman. Extendable Physical Unit Checking with Understandable Error Reporting. In *Proceedings of the 7th International Modelica Conference*, Como, Italy, 2009

- David Broman. Should Software Engineering Projects be the Backbone or the Tail of Computing Curricula?. In *Proceedings of the 23th IEEE Conference on Software Engineering Education and Training*, Pages 153-156, Pittsburgh, USA, 2010

- Peter Fritzson, Adrian Pop, David Broman, and Peter Aronsson. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Proceedings of the 20th Australian Software Engineering Conference ASWEC 2009*, pages 256-266, Gold Coast, Queensland, Australia, IEEE Computer Society, 2009

- David Broman, Peter Aronsson, and Peter Fritzson. Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica. In *Proceedings of the 6th International Modelica Conference*, pages 3-12, Bielefeld, Germany, 2008

- Peter Fritzson, David Broman, and François Cellier. Equation-Based Object-Oriented Languages and Tools. Report on the 2nd Workshop EOOLT at ECOOP 2008. *Object-Oriented Technology. ECOOP 2008 Workshop Reader*, Volume 5475 of LNCS, pages 18-29, Springer-Verlag, 2009 (Invited)

- Peter Fritzson, David Broman, François Cellier, and Christoph Nytsch-Geusen. Equation-Based Object-Oriented Languages and Tools. Report on the Workshop EOOLT 2007 at ECOOP 2007. *Object-Oriented Technology. ECOOP 2007 Workshop Reader*, Volume 4906 of LNCS, pages 27-39, Springer-Verlag, 2008 (Invited)

- Peter Fritzson, François Cellier, and David Broman (Eds.). Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools. Cyprus, July 2008. ISSN 1650-3686, LIU Electronic Press

- Kristoffer Norling, David Broman, Peter Fritzson, Alexander Siemers, and Dag Fritzson. Secure Distributed Co-Simulation over Wide Area Networks. In *Proceedings of the 48th Conference on Simulation and Modelling (SIMS'07)*. Göteborg, Sweden, Linköping University Electronic Press, 2007

- Peter Fritzson, Peter Aronsson, Adrian Pop, Håkan Lundvall, Kaj Nyström, Levon Saldamli, David Broman, Anders Sandholm. OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, Munich, Germany, 2006 (Invited)

- David Broman and Peter Fritzson. Ideas for Security Assurance in Security Critical Software using Modelica. In *Proceedings of the Conference on Modeling and Simulation for Public Safety*, pages 45-54, Linköping, Sweden, 2005

- Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. In *Proceedings of the 46th Conference on Simulation and Modeling*, pages 83-90, Trondheim, Norway, 2005

1.5.5 Origin of Contributions

The most significant part of the research work and contributions in this thesis originates entirely from the author of the thesis. However, because several of the published papers included in this thesis have co-authors, we detail the exact origin of the contributions below.

Part I - The Modelica Language

The work on how to specify the Modelica language [24] and strategies for growing equation-based languages [19] are done solely by the author with Peter Fritzson as supporting supervisor.

The work on *Modelica types* [27] was carried out by the author, where both Peter Fritzson and Sébastien Furic co-authored the publication. They contributed as discussion partners and through proofreading the manuscript.

The idea and formalization of the work on the *structural constraint delta* [28] was carried out solely by the author. Co-authors Kaj Nyström and Peter Fritzson contributed as discussion partners, as proofreaders, and with shorter sections in the original paper.

Part II - The Modeling Kernel Language

The published work in this part concerns *higher-order acausal models (HOAM)* [25, 26]. The co-author Peter Fritzson has been supervisor of the work, contributing with feedback and proofreading.

The semantics of the MKL language has been developed solely by the author of the thesis, where Jeremy Siek has been supporting co-supervisor.

The implementation, validation, and written manuscript are performed entirely by the author of the thesis.

1.5.6 Reading Guidelines

We will now give some brief guidelines for reading the thesis. The aim of this thesis is to have a broad audience, where readers might come from either different engineering fields or from the field of computer science. The two main parts of the thesis (Part I about Modelica and Part II about MKL) are self contained and can be read independently of each other. For each of the parts, there is an introduction chapter which is recommended reading before proceeding with the other chapters.

Readers from different engineering fields with an interest in modeling and a background of e.g., Modelica might be especially interested in Chapter 5 about types in Modelica, as well as the ideas of higher-order acausal models, presented in Chapter 8.

Readers with a background of designing modeling languages in general and Modelica in particular might be interested in Chapter 4 about growing an EOO language. Part II presents language concepts that are not directly related to Modelica, but can be of interest for further extensions of such a language. Chapters that can be of particular interest are: introduction to functional programming in MKL (Chapters 7), modeling with higher-order acausal models (Chapter 8), using and inspecting the content of models (Chapter 9), and description of elaboration semantics (Chapter 11).

Readers with a computer science background would perhaps be most interested in Part II. If the reader has a more theoretical programming language background, Chapter 10 with formal operational semantics and type safety proofs could be of interest.

1.6 Research Method

There are several different paradigms on how to perform research within engineering and computer science. The ACM Task Force on the *core of computer science* suggests three different paradigms for performing research within the discipline of computing [42]:

1. *Theory.* In this paradigm, the discipline is rooted in mathematics, where the objects of study are defined, hypotheses (the theorems) are stated, and proofs of the theorems are given. Finally, the result is interpreted.

2. *Abstraction (modeling).* The second paradigm is rooted in experimental scientific methods. First, a hypothesis is formulated, followed by the construction of a model and/or an experiment from which data is collected. Finally the result is analyzed.

3. *Design.* The third paradigm is rooted in engineering and consists of stating requirements, defining the specification, designing and implementing the system, and finally testing the system. The purpose of constructing the system is to solve a given problem.

Theory is the fundamental paradigm in mathematical science, the abstraction paradigm in natural science, and design in the discipline of engineering. We agree with the statement that is pointed out by Denning *et. al.* [42], that all three paradigms are equally important and that computer science and engineering consist of a mixture of all three paradigms. In this work, we have used different paradigms for the different parts of the work.

In the work on *types in the Modelica language* (Chapter 5), the type concept of Modelica is studied and interpreted and a concrete syntax of types for Modelica is described. The closest paradigm used in this work is design, where the designed artifact is the grammar for types and the interpreted prefix definitions. The correctness of the grammar is verified using the parser generator tool ANTLR [119]. In this case, the Modelica specification itself can be seen as the requirements specification and the produced artifact is an interpretation of this specification.

In the work on *structural constraint delta* (Chapter 6) we define a new approach and an algorithm for determining over- and under-constrained systems of equations. This research can be assigned to both the theory and the design paradigms. From the theory point of view, if a theorem was formulated for the correctness of the algorithm, a proof would justify the correctness of the algorithm. On the other hand, from a design point of view, the requirement of detecting and isolating the error before elaboration can be seen as a specification, and an implementation of the algorithm as the system. Because Modelica's semantics is not formally defined, it is not possible to conduct any proof of the correctness of the algorithm in relation to the elaboration semantics. Hence, we use a test procedure where the correctness of the algorithm is tested using different test models, where the model is executed in the commercial Modelica tool Dymola version 6 [45], and compared to an implementation of the algorithm given in Chapter 6. We should note that this test only checks the correctness of the algorithm, and does not verify that the approach of the structural constraint delta actually helps the user to detect the error and isolate the fault.

Finally, our work of being able to design EOO language constructs in the *modeling kernel language (MKL)* (Part II) has been verified using basic engineering principles of testing as well as conducting proofs of the language's properties. For the testing and verification, a set of models have been implemented in both Modelica and using a standard library in MKL. The MKL models have then been translated into flat Modelica code. The results of simulating both the translated and the native Modelica model have then been compared. Also, the models have been simulated using MKL, where the simulation result has been compared with Modelica simulations. For the theory part, we have proved type safety for a core language of MKL. This gives us higher confidence of the correctness of our approach, but can of course not guarantee the correctness of the correspondence between the formal semantics and the implementation.

Part I

The Modelica Language

2

Introduction to Modelica

MODELICA is a standardized language aimed primarily for modeling and simulation of complex physical systems. The first language specification 1.0 [101] was released in September 1997. Since then, the current specification 3.2 [104] has evolved to be a specification of a language that has a large number of complex constructs.

During these past 13 years, the Modelica user community has grown to become fairly large. Modelica has been used successfully in industry and the *Modelica standard library (MSL)* has evolved to include domains such as electrical, mechanical, hydraulic, fluid, thermal and control. The dominating Modelica tool has for a long time been the commercial tool Dymola [45]. However, during recent years, alternative tools have emerged; both open source (e.g., OpenModelica [53, 117], Scicos [73, 106], and JModelica.org [5, 78]) and commercial environments (e.g., MathModelica System Designer [91], MOSILAB [112], SimulationX [76], LMS Imagine.Lab AMESim [89], and MapleSim [90]).

Modelica Association [100] is responsible for both the language specification as well as the Modelica standard library. The author of this study has been a member of the Modelica language design group since 2005. The work presented in this part has been developed 2005 to 2010 contains both discussions and analysis of the current language specification (Chapters 3-5) as well as proposed extensions (Chapter 6). This part of the thesis consists of the following chapters:

- **Chapter 2 - Introduction to Modelica**. In this introductory chapter we first give a brief informal overview of the Modelica language from a modeling point of view. This is followed by a description of Modelica's compilation and simulation process.

- **Chapter 3 - Specifying the Modelica Language**. This chapter concerns the problem of having a large and informal language specification. We discuss different aspects of formulating the Modelica language.

- **Chapter 4 - Growing the Modelica Language**. We discuss how the Modelica language can be planned for growth, i.e., how the language can be extended over time.
- **Chapter 5 - Types in the Modelica Language**. We investigate and analyze the concept of types in the Modelica language as well as proposing a concrete syntax for describing Modelica types.
- **Chapter 6 - Over- and Under-Constrained Systems of Equations**. In this chapter we propose a technique for detecting and isolating over- and under-constrained systems of equations in EOO languages. We test the described approach, called *structural constraint delta*, on a subset of the Modelica language. However, the approach is not limited to Modelica and is applicable for EOO languages in general.

2.1 Equation-Based Modeling in Modelica

In this section we illustrate some important and fundamental concepts in modeling with Modelica. A comprehensive description of the language is given by Fritzson [51].

The basic structuring element in Modelica is the *class*. There are several restricted class categories with specific keywords, such as `model`, `record` (a class without equations), and `connector` (a record that can be used in connections). A class contains *elements*, which can be other class definitions, extends elements (for inheritance of other classes), or *components* (instances of classes).

The main difference compared to traditional OO languages is that instead of methods, Modelica primarily uses *equations* to specify behavior.

As a brief introduction to Modelica, we present a model of a simple electrical circuit (Figure 2.1). On the left hand side the textual representation of the circuit is given

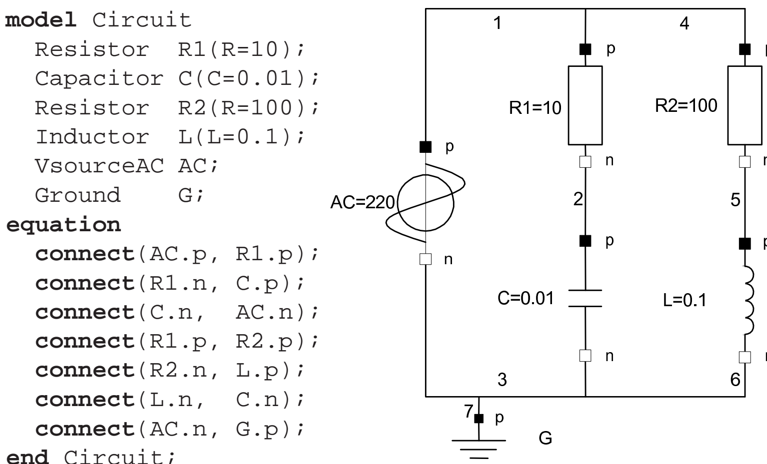


Figure 2.1: *Modelica model of an electrical circuit.*

and on the right hand side the graphical representation. Typically, a Modelica modeling environment lets the user model the system in either the graphical or the textual view.

Now, let us consider the textual representation. The first six lines of code represent the *components* of the model. These have a direct correspondence to the components in the graphical view. In the first component declaration

```
Resistor  R1(R=10);
```

the indentifier `Resistor` is a class reference, `R1` is a component name, and `R=10` is a modification that sets the resistance `R` equal to `10`.

The last seven lines that are part of the `equation` section are called `connect`-equations. These equations are used for connecting *connectors* (also called ports) together. For example, the equations `connect(AC.p, R1.p)` and `connect(R1.p, R2.p)` state that connectors `AC.p`, `R1.p`, and `R2.p` are connected together. Such a set of connected connectors is called a *connection set*. Note that because models are hierarchically defined, names to particular components are specified using a dot-notation, e.g., `R1.p` is the positive connector of component `R1`.

Now consider the following example of a connector used for acausal connections in the electrical domain:

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

where `v` is the potential variable representing voltage and `i` the flow variable for electrical current. The connection set {`AC.p`, `R1.p`, `R2.p`} is then during the elaboration phase translated into two equality equations for the potential variables:

```
AC.p.v = R1.p.v;
R1.p.v = R2.p.v;
```

and one sum-to-zero equation for Kirchoff's current law:

```
AC.p.i + R1.p.i + R2.p.i = 0;
```

Variables in connectors can also have the prefixes `input` or `output` stating that these variables are used for *causal connections*.

Now, let us consider how the models of the components in Figure 2.1 are defined. A first observation can be made that all these components have two ports (except for the ground component that has one port). Thus it is useful to define a "base" `TwoPin` model as follows:

```
model TwoPin "Superclass of model components with two pins"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

This component has two connectors `p` and `n` defined as an instance of the connector class `Pin`. The variable `v` defines the voltage drop over the component using the equation `v = p.v - n.v`. The variable `i` is the current into the pin `p`.

To define a model for an electrical capacitor we can now extend our base class `TwoPin` and add a declaration of a variable for the capacitance and one equation expressing the capacitor's behavior:

```
model Capacitor "Ideal electrical capacitor"
  extends TwoPin;
  Real C "Capacitance";
equation
  C*der(v) = i;
end Capacitor;
```

The keyword `extends` denotes inheritance from one or more base classes. Elements and equations are inherited from the parent. The equation `C*der(v) = i` contains the expression `der(v)` meaning the derivative of `v`.

When extending or declaring an element, we can also add *modification equations*. The simplest form of modification is binding a value to a variable:

```
  Resistor R1(R=100);
```

It is also possible to alter the internal structure of a component when declaring or extending it, using redeclarations. The `redeclare` construct changes the class of the component being replaced. There are two restrictions on this operation:

1. The component we are replacing must be declared as `replaceable`.
2. The replacing class's type must be a subtype of the type of the component being replaced.

Consider now the following example

```
model A
  replaceable Resistor R1(R=100);
end model A;

model B
  extends A(redeclare TempResistor R1);
end B;
```

where we define a model `B` by extending from model `A` and at the same time change resistor `R1` to be a `TempResistor`.

This short introduction to Modelica only describes a very small part of the Modelica language. However, there should be enough details to understand the rest of Part I of the thesis. The language contains many more constructs that are outside the scope of this thesis. Some of these constructs are:

- Arrays, vectors, and matrices.
- Side-effect free functions and algorithm sections.
- When- and if-equations for expressing hybrid models.
- Conditional components for selecting components during elaboration.
- Packages for structuring source code into modules.
- Overloading of operators.
- Stream connectors.
- Mapping of models to execution environments (used in for example hardware-in-the-loop simulations.).

See Fritzson [51] and the Modelica specification 3.2 [104] for descriptions of these and other advanced constructs in Modelica.

2.2 The Modelica Compilation and Simulation Process

Figure 2.2 outlines a typical compilation and simulation process for a Modelica software tool. The input (left hand side of the figure) to the process is a Modelica model, which can compose and reference a large set of additonal Modelica models. The first phase that is carried out is standard lexical analysis and parsing. Output from this step is an *abstract syntax tree (AST)*. This phase can, depending on the implementation, be performed in several stages, where each stage simplifies and normalizes the form of the AST.

The second phase of the process is the *elaboration*, where the AST is transformed into a *hybrid DAE*. A hybrid DAE consists of variable declarations, the differential-algebraic equation system (for continuous-time behavior), algorithm sections, and `when`-clauses for triggering discrete-time behavior. During this phase the model is also checked for errors, such as conformance of types.

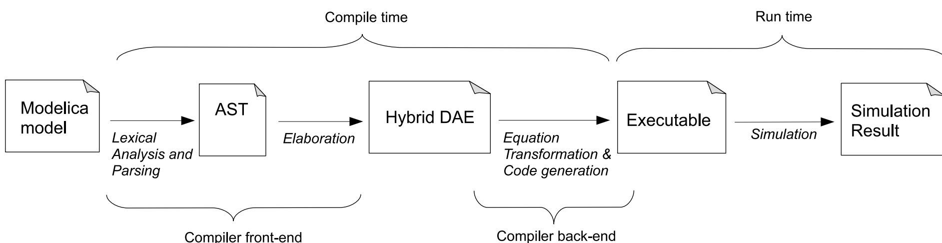


Figure 2.2: *Outline of a typical compilation and simulation process for a Modelica language tool.*

The two first phases, lexical analysis and parsing followed by elaboration, is often referred to as the compiler *front-end*. Semantics discussed in this thesis is focused on this part.

The next phase of the compiler, the *back-end*, is first transforming and manipulating the equation system to make it solvable. Key aspects of this process are the use of Pantelides algorithm [118], block lower triangular form (BLT) [70, 71], dummy-derivatives [94], and tearing [49]. Typically the DAE is reduced to an index one problem and then solved with a DAE solver such as DASSL [121] or the IDA solver within the SUNDIALS solver suite [68]. The equation system could also be translated and sorted to form an ODE, to be solved with a numerical integration method, such as Runge-Kutta. Typically, the right-hand side of the equation system (for an ODE) or the residual function (for an DAE) is translated to executable code, where the typical target language is C. Finally, these generated functions together with a main program is linked together with a numerical solver and then compiled into an executable file.

The process of phases two and three is typically performed at compile time. The last step, when the model is simulated (executed) is often referred to as the run time semantics of the process. Output from this process is typically a file containing simulation data for the state variables. The data is then later visualized using a *graphical user interface (GUI)*.

The process of elaboration, where a EOO model is translated into an equation system, can informally be described to perform at least the following three main activities:

- *Type checking of models.* Check that parameterized models conform to the type rules of the language and that basic operations and function calls are type correct. For example, a function having one argument cannot be applied to two arguments and a plus operator cannot have a string as its left operand and an integer as its right operand, etc.

- *Collapsing the instance hierarchy.* During this activity, new unknowns and equations are created for sub-components of a model. For example, if a model contains two resistors `R1` and `R2`, where `R1` is parameterized with 10 ohm, and `R2` with 50 ohm, two equations are created `u1 = 10 * i1` and `u2 = 50 * i2`. Moreover, unknowns, such as the voltage drop over the components must be different for the components. Hence, `u1` and `u2` must be different unknowns.

- *Connection semantics.* Acausal ports contain flow and potential variables, where the former must sum-to-zero at connection points and the latter must have the same potential at the connection point. This activity generates equations and unknowns to enable acausal modeling.

We postpone the discussion about type checking of Modelica models to Chapter 5 and 6 and the discussion of connection semantics to Chapter 11.

We will now present a very simple example to give an intuition of how the instance hierarchy of a model is collapsed. The example only illustrates the basic principles for composed models, together with Modelica's inheritance mechanism. In a compiler that handles the full Modelica language this is one of the most substantial parts due to number of constructs in the language and its complicated look-up mechanism.

Consider the following Modelica models:

```
model B
  Real y;
  Real x;
equation
  y=der(x);
end B;
```

```
model C
  Real z=10;
  Real t;
equation
  t=z*2;
end C;
```

```
model A
  extends C(z=5);
  B b;
end A;
```

The task is to elaborate model `A`. This means that from the code in model `A`, we should extract the corresponding system of equations. Examining model `A`, we find that it extends (inherits from) `C`. Our action is then to simply copy the contents of model `C` into our working copy of model `A`. The modification equation to variable `z` in the extends clause replaces the declaration equation to variable `z` in `C`. All modifications are resolved as equations so the overriding modification `z=5` is put in the equation section. The result so far is:

```
model A
  Real z;
  Real t;
  B b;
equation
  z=5;
  t=z*2;
end A;
```

We do not have to do anything about declarations of variables with *predefined types* (e.g., `Real`, `Integer`, and `Boolean`). However, the component `b` must be elaborated because `B` is not of a predefined type. We investigate the model `B` and find that it contains the declarations `Real y` and `Real x`. These declarations and all equations in the model `B` will now be inserted in our working model `A` with the prefix `b.` as we have now entered the namespace of the component `b`. The elaboration is now complete because there are only predefined types left in our working model. The final result of this basic elaboration (excluding type checking and connection semantics) is

```
model A
  Real z;
  Real t;
  Real b.x;
  Real b.y;
equation
  b.y=der(b.x);
  z=5;
  t=z*2;
end A;
```

2.3 Chapter Summary and Conclusions

We have in this chapter given a brief overview of the Modelica language. We have presented modeling examples within the electrical domain and explained the overall structure of the Modelica compilation and simulation process. Finally, we have given a short overview of the elaboration phase in Modelica.

3

Specifying the Modelica Language

THE Modelica language specification defines a formal concrete syntax, but the semantics is informally described using natural language and with examples. This makes the language's semantics hard to interpret unambiguously and to reason about, which affects both tool development and language evolution. Even if a completely formal semantics of the Modelica language can be seen as a natural goal, it is a well-known fact that defining understandable and concise formal semantics specifications for large and complex languages is a very hard problem. In this chapter, we will discuss different aspects of formulating a Modelica specification; both in terms of *what* should be specified and *how* it can be done. Moreover, we will further argue that a "middle-way" strategy can make the specification both clearer and easier to reason about. An idea for a proposal is outlined where the current informally specified semantics is complemented with several grammars, specifying intermediate representations of abstract syntax. We believe that this kind of evolutionary strategy is easier to gain acceptance for, and is more realistic in the short-term, than a revolutionary approach of using a fully formal semantics definition of the language. This chapter is organized as follow:

- We introduce and motivate the need for a middle-way strategy for specifying the Modelica language (Section 3.1).

- Different ways of specifying a Modelica specification are discussed and analyzed (Section 3.2).

- We explain the idea of specifying abstract syntax for improving the specification (Section 3.3).

3.1 Introduction and Motivation

In this section we first discuss different alternatives of specifying a language specification and then briefly describe previous attempts of specifying subsets of Modelica. Finally, we introduce the idea of using several intermediate languages as middle-way strategy.

3.1.1 Unambiguous and Understandable Language Specification

The natural goal of a language specification is to be *unambiguous*, so that tool implementors interpret the specification in exactly the same way. At the same time, it is important that the specification is *easy to understand* for the intended audience. Unfortunately, it is not that easy to meet both of these goals when describing a large and complex modeling language such as Modelica. There are several specification approaches with different pros and cons. Hence, the overall problem is to find an approach that satisfies the specification goals in the best possible way.

If the language is described using *formal semantics*, e.g., structured operational semantics [126], the language semantics can in some cases be proved to have certain properties, such as type safety [124, 151]. However, to understand and interpret a formal language specification requires a rigorous theoretical computer science knowledge. Furthermore, even if great effort has been spent during the recent decades in formalizing mainstream programing languages, only a few, e.g., Standard ML [98], are actually fully formally specified. Accordingly, it turns out to be a very hard task to specify an understandable and concise formal specification of an existing complex language.

Alternatively, if the language semantics is described using *natural languages*, e.g., plain English text describing the semantics, it might be easy for software engineers to understand the specification. Many languages are described in this way, for example Java [65], C++ [75], and Modelica [102]. However, ease of understanding does not imply that different individuals interpret the specification in the same way. It is a well known fact that it is very hard to write unambiguous natural language specifications, and perhaps even harder to verify their consistency.

3.1.2 Previous Specification Attempts

Several previous attempts have been made to formalize and improve the specification of the Modelica language. The most obvious one is the further development of the official language specification itself, conducted by the Modelica Association. The work that resulted in version 3.0 of the language specification contained substantial restructuring and a more detailed description of the semantics of the language. However, it is not planned to include any formal descriptions.

Three earlier attempts for improving the specification of the Modelica language have previously been conducted. The first one, started already in 1998 by Kågedal and Fritzson [84, 85] tried to describe the language using *Natural Semantics* [81]. As the specification grew over time it has evolved into the OpenModelica compiler [117]. The second attempt by Mauss [95] described the instance creation of the elaboration, but not the type checking. Finally, our work on Modelica types [27] (described in the next Chapter), tries to clarify the type concept in Modelica, but does not involve the actual type checking

algorithm. A more in depth discussion of related work is given at the end of the thesis in Section 13.2.

A common denominator for all these isolated formal specification attempts is that they have been conducted in parallel with the official language specification. Even if a proposed alternative specification covers large portions of the language, it will not be used as a specification by the community if it is not replacing the official specification. If there are two specifications of the same concept, how do we then know which one is valid if they are not consistent? Nevertheless, these formal specification attempts are still very important to promote understanding and discussion about the informal semantics. It is of great importance that these works gradually find their way into the official specification. The problem is how to make this possible in practice because all attempts so far only model subsets of the real language.

3.1.3 Abstract Syntax as a Middle-Way Strategy

Improving the natural language description of the Modelica specification is an obvious way of increasing the understandability and removing ambiguity. However, previous work on formalization of the complete semantics of subsets of the language has shown to be complex and resulting in very large specifications. Hence, there is a concrete and practical need to find a "middle-way" strategy to improve the clarity of the complete language, not just subsets. This strategy must be simple enough to not require in depth theoretical computer science knowledge of the reader, but still precise enough to avoid ambiguities.

When a compiler parses a Modelica model, the result is transformed into an *abstract syntax tree (AST)* [4, 124]. The abstract syntax can be specified using a *context-free grammar*.

The internal representation of an AST is often seen as a tool implementation issue, and not as something that is defined in a language specification. Nevertheless, in this chapter we sketch the idea that an intermediate representations between the transformation steps (recall Figure 2.2) should be described by specifying its abstract syntax. Note that this abstract syntax is *only* intended as an abstract concept specified in a language specification and read by humans. It is not intended for implementation.

However, specifying different forms of abstract syntax *cannot* replace the semantic specification needed in the transformation process because the syntax only describes the *structure* of a Modelica model, while the semantics states the *meaning* of it. Hence, in the short term, this specification *complements* the current informal specification, by clarifying exactly what both the input and the output structure of a transformation are.

By following this *evolutionary* strategy, the semantic description may then be gradually specified more formally. However, this is not straight forward when considering the whole Modelica language. The main purposes of including abstract syntax definitions in the specification can be summarized to be:

1. **Specifying Valid Input.** Increase the clarity of what valid Modelica actually is, i.e, to make sure that different tools reject and accept the same models.

2. **Specifying Expected Output.** Remove confusion of what the actual outcome of executing a Modelica model is.

3. Promoting Language Simplification. The Modelica language has been identified to be sometimes more complicated than necessary (e.g., relations between the general class and restricted classes). An abstract syntax formulation can be used as a guidance tool for identifying the most useful reformulations needed.

Part of the first item is already specified using the concrete grammar. To increase the level of detail that can be specified of the abstract syntax, we will later on suggest an informal approach to include context-sensitive information in the abstract grammar specification. This rules out parts of the informal semantics used for rejecting invalid models. However, large parts of the rejecting semantics must still be described using another semantic specification form.

3.2 Specifying the Modelica Language

Defining a new language from scratch with an unambiguous and understandable language specification is a difficult and time consuming task. Developing and enhancing a language over many years and still being able to keep the language backwards compatible and the specification clear, is perhaps an even more challenging mission. In the previous section, we described this problem with the current specification, motivated the need for improvement, and briefly introduced a proposed strategy. In the beginning of this section, we will focus on the question *what* should actually be specified in the Modelica specification. At the end of the section, we will discuss *how* this specification can be achieved by surveying some different specification approaches and compare how they relate to the abstract syntax approach.

At a high level, the syntax and semantics of Modelica can be divided into two main aspects:

- *Transformation*, i.e., the process of transforming a Modelica source code model into a well defined result. Depending on the purpose, the result can either be an intermediate form of a hybrid DAE, or the final simulation result.

- *Checking*, i.e., rules describing what a valid Modelica model actually is. These rules should unambiguously describe when a tool should reject the input model as invalid.

Both these aspects are important for a clear-cut result, so that tool vendors can create compatible tools.

3.2.1 Transformation Aspects - *What* is Actually the Result of an Execution?

In the introduction section of the Modelica specification 3.2 [104], it is stated that the scope of the specification is to define the semantics of the translation to a flat Hybrid

DAE and that it does not define the result of a simulation. A mathematical notation of the hybrid DAE is given, but no precise and complete output is defined.

However, many constructs given in the specification are not transformed to more primitive constructs during this translation to a Hybrid DAE. Hence, the semantics of these constructs (e.g., when-equations, algorithm sections), are implicitly defined, even if the specification states that this should not be the case.

Therefore questions arise: what is actually the transformation process? What is the expected result of the execution? We would argue that the answer to these questions would differ depending on who you ask because the current specification is open for interpretation.

Static vs. Dynamic

In the previous description of the compilation and simulation process, it was assumed that the process was *compiled* and not *interpreted*. This is not a specification requirement, even if it is common that tools are implemented as compilers. The definitions of static and dynamic semantics are often confusing in relation to compile-time and simulation-time. Some people will argue that the dynamic semantics is only the simulation sub-process and that the elaboration and equation transformation as well as the code generation phases are the static semantics. However, in Modelica it is possible to define a recursive model that refers to itself, thus resulting in an infinite loop during the elaboration process. In such a case, it questionable if the elaboration process can be called static.

From the above discussion, it is clear that we need to have a precise definition of the input and the output of the elaboration process. Whether the two last sub-processes should be part of the specification is an open design issue, but it is obviously important that the decision is made if it should be completely included or removed.

3.2.2 Checking Aspects - *What* is actually a Valid Modelica Model?

In the current specification, it is hard to interpret what valid Modelica input is, i.e., it is difficult for a tool implementor to know which models that should be rejected as invalid Modelica. A restrictive abstract syntax definition can help clarifying several issues.

Besides specifying the translation semantics of a model, a language specification typically describes which models that should be treated as valid, and which should not. By an *invalid model* we mean an transformation that should result in an error report by the tool. In order for different tool vendors to be able to state that exactly the same models are invalid, *when* and *how* to detect model faults must be clearly and precisely described in the language specification. Unfortunately, this is not as easy as it might seem.

Basically, rules in a specification for stating a valid model can be specified by using one of the following strategies, or a combination of both:

- Specify rules that indicate valid models. All models that do not fit to these rules are assumed to be invalid.
- Assume that all models are valid. Explicitly state exceptions where models are *not* valid.

The current Modelica specification mostly follows the latter approach. Here the concrete syntax constrains the set of legal models at a syntactic level. Then, informal rules given in natural language together with concrete examples state when a model can be legal or illegal.

The problem with this approach is that it is very hard for a tool vendor to be sure that a tool is compliant with the specification.

Time of checking

Detecting that a model is invalid can take place at different points in time during the compilation and simulation phase. Even if this can be regarded as a tool issue and not a language specification detail, the checking time has great implications on a tool's ability to guarantee detection of invalid models.

Figure 3.1 outlines a simplified view of the earlier described compilation and simulation process, where the three sub-processes of equation-transformation, code generation and simulation are combined into one transformation step. Also, the lexical analysis and parsing steps are omitted from the figure.

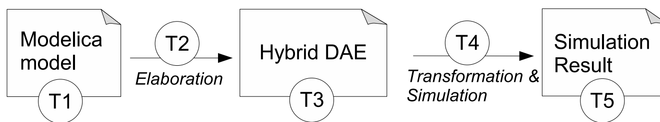


Figure 3.1: *Possible points in time when the checking can occur.*

The figure shows five (`T1 - T5`) conceptual points in time where the checking and rejection of models can take place. Starting from the end, `T5` illustrates the final step of checking that the simulation result data is correct according to some requirements. This checking can normally not be conducted by a tool, but only by humans who have the domain knowledge.

The checking at point `T4` takes place during simulation of the model. This is what many would refer to as *dynamic checking* because it is performed during runtime. Errors which can occur here are for example numerical singularities after events or array out-of-bound errors. Because Modelica does not have an exception handling mechanism, it is implicitly assumed that the tool exits with an error statement. Checking point `T3` is performed after the elaboration phase. This can for example concern controlling that the number of equations equals the number of unknowns.

Even if it is not stated in the Modelica specification, `T2` is our interpretation of the specification where the type checking takes place. Here, the naming of this kind of checking is often a source of confusion. If the elaboration phase is regarded as the *static semantics*, some people call this *static type checking*. However, because the elaboration phase is the major part of the semantics described in the specification, and it involves complex transformation semantics, this can be viewed as something dynamic from an interpretive semantics point of view, or as something static from a translational semantics point of view. Using an interpretive semantics style, `T2` would involve *dynamic type checking*.

Following this argumentation, then `T1` would represent *static type checking*, i.e., the types in the language are checked *before* elaboration. This reasoning is analogous to

dynamic checking in languages such as PHP and Common LISP, compared to static type checking in Haskell, Standard ML, or Java. Even if the Modelica specification does not currently support this kind of static checking, it has a major impact on the ability to detect and isolate for example over- and under-constrained systems of equations or to enable separate compilation.

3.2.3 Specification Approaches - *How* can we State What it's all About?

When it is clear *what* to specify, the next obvious question is *how* to specify it. There are several specification approaches, and we have briefly mentioned some of them earlier in this chapter.

As evaluation criteria, it is natural to use the specification goals of *understandability*[1] and *unambiguity*. Furthermore, it is also of interest to estimate the *expressiveness* of the approach, i.e., how much of the intended specification task can be covered by the approach.

In the following table, a number of possible specification approaches are listed, with our judgements of the evaluation criteria.

Approach	**Understandability**	**Expressiveness**	**Unambiguous**
Natural language description	High-Medium	High	Low
Formal semantics	Low	Medium	High
Abstract Syntax Grammar	Medium	Medium	High
Concrete Syntax Grammar	Medium	Low	High
Test suite	High	High	Medium
Reference Implementation	Medium	High	High

Table 3.1: *Possible specification approaches with our judgements of the evaluation criteria.*

A natural language specification can be understandable and expressive, depending on the size and quality of the text, but easily leads, as we have discussed earlier, to ambiguous specifications. Using a formal type system together with formal semantics [124] is here seen as having low understandability because it requires high technical training. It is however very precise and fairly expressive.

The expressiveness of the abstract syntax is stated as higher than the concrete syntax because we can introduce context dependent information in the grammar using meta-variables. An example of this will be given in the next section.

We have also included approaches such as the use of a test suite and reference implementation. The approach to use a test suite as a specification can be an interesting complement to abstract syntax and informal semantics. However, it is very important to

[1]Understandability is of course a very subjective measurement. In this context, we have chosen to also include the level of needed knowledge to understand the concept, i.e., a concept requiring an extensive computer science or mathematical background results in lower understandability rating.

state which description that has precedence if ambiguities are discovered. Finally, a reference implementation can also be seen as a specification, even if it could be hard to get a good overview over it if the language is large and complex.

3.3 An Abstract Syntax Specification Approach

In the following section we will briefly discuss the idea to use abstract syntax as part of the Modelica specification. Initially, the different abstract syntax representations are outlined in relation to the transformation process described in Section 3.2.1, followed by a discussion about the specification and representation of the syntax.

3.3.1 Specifying the Elaboration Process

An *Abstract Syntax Tree* (AST) can be seen as a specific instance of an abstract syntax. Transformation processes inside a compiler can be defined as transformations from one intermediate representation to another. ASTs are a natural form of intermediate representation.

Consider Figure 3.2, where the elaboration process is shown with surrounding ASTs. The first step in the process is the ordinary scanning and parsing step, which is formally defined in the specification using lexical definitions and concrete syntax definitions using Extended BNF.

Complete AST (C-AST)

This step transforms into the first tree called *Complete AST (C-AST)*, which is a direct mapping of the concrete syntax. Although this is a natural step in a compiler implementation, it is of minor interest from a specification perspective.

Simplified AST (S-AST)

From the C-AST, a simplification transformation translates the C-AST into a simplified form called *Simplified AST (S-AST)*. This transformation's goals are:

- *Desugaring* : The process of removing so called *syntactic sugar*, which is a convenient syntactic extension for the modeling engineer, but with no direct implication on the semantics. Example of such desugaring of a model is to collect all equation

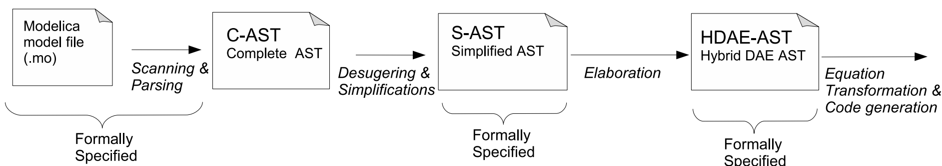


Figure 3.2: *Modelica's compilation process divided into intermediate representations in the form of abstract syntax trees (ASTs).*

sections into one list because the Modelica syntax allows several algorithm and equation sections to be defined in a model.

- *Normalizing Transformations* : Minor transformations and operations that help the S-AST to be a canonical form which is more suitable as input to the elaboration process. For example assigning correct prefixes to subelements.

- *Checking model validity* : One of the purposes with S-AST is that it is more restrictive than the C-AST. Hence, some C-AST constructs are not valid S-AST. This restriction gives the possibility to ensure certain model properties, which in the current Modelica specification are described using informal natural languages. For example, which kind of restricted classes is the record class allowed to contain as its elements?

The S-AST can be seen as a simplified internal language analogously to the *bare* language of Standard ML [98]. However, initially, we do not see a similar short and precise way of specifying the transformation from C-AST to S-AST because of the size and complexity of the language.

Hybrid DAE AST (HDAE-AST)

Besides S-AST, the output of the elaboration phase called Hybrid DAE AST (HDAE-AST) is proposed to be specified formally in the specification. The HDAE-AST must not just be a high-level mathematical description of an Hybrid DAE, but an explicit syntax description describing a complete specification of what the actual output of the elaboration phase is. This does not only include equations and variables, but function definitions, algorithm sections, when-equations and when-statements. Even if this information is possible to derive from the current specification, it would be a great help for the reader to actually know what the output is, not just assume it.

Note that our approach suggests that the language specification should initially include a precise description of the possible *structures* of the ASTs; specifying input and output to the transformation process. The semantics of the transformation must still be described using another approach that initially could still be an informal description.

3.3.2 Specifying the Abstract Syntax

The specification of the syntax must be described using some kind of *grammar*. The syntax can be specified using a *context-free* grammar, e.g. in Backus-Naur Form (BNF). However, we propose a more abstract definition of a grammar, where certain *meta-variables* range over names and identifiers.

For example, by stating that a meta-variable R_r ranges over *names* (identifiers with possible dot-notation) referencing a `record`, we have introduced a contextual dependency in the grammar. The grammar declaratively states the requirement that this name must after lookup be a record, without stating *how* the name lookup should be performed.

The latter must of course also be described in the specification, but in this way the different issues are separated. Consequently, this grammar is not intended to be used directly by a parser generator tool such as Yacc, but as a high-level specification which is less open for interpretation.

3.3.3 The Structure of an Abstract Syntax

Depending on the purpose and language for an abstract syntax, the structure of the syntax itself can be very different.

When specifying a simple functional languages, it is common that the grammar of the abstract syntax only has one non-terminal, namely a *term* [124]. Hence, all evaluation semantics is performed on this node type only, and all terms can be nested into each other. This gives a very expressive language, but the constraining rules ensuring the validity of an input program must be given in another form. This form is normally a formal *type system*, describing allowed terms.

Another method is to describe the abstract syntax with many non-terminals; more than needed for a production compiler. In for example the Modelica case, the different restricted classes: `model`, `block`, `connector`, `package`, and `record` would not be represented as one non-terminal *class*, but as different non-terminals. This structure would be more verbose, but also give the possibility of more precisely describing relations between restricted classes.

Somewhere inbetween those two extremes is for example the `SCODE` representation used in the earlier RML specification [84] and the current OpenModelica implementation. For the specification purpose, we suggest to use the most verbose alternative, i.e. the second alternative using many non-terminals. The rational for this choice is basically that this more restrictive form gives more information about what the actual input and output of the elaboration processes are.

3.3.4 A Connector S-AST Example with Meta-Variables

To give a concrete example where a grammar for S-AST can improve the clarity compared to the current informal specification, we take the restricted class `connector` as an example. In the Modelica specification it is stated that for a connector *"No equations are allowed in the definition or in any of its components"*. What does this mean? That no equations are allowed at all? Are declaration equations allowed, for example `Real x = 4`? Obviously, it is not allowed to have instances of models that contain equations, but is it allowed to have models that do not contain equations? Is it only allowed to have connectors inside connectors, or can we also have records in connectors, since these are not allowed to have equations either? These questions are not easy to answer with the current specification, because it is open for interpretation.

Consider Figure 3.3, where an example of the non-terminal for a `connector` is listed using a variant of Extended BNF[2]. As usual, alternatives are seprated using the '|' symbol, and curly brackets ($\{\ldots\}$) denote that the enclosing elements can be repeated zero or more times.

[2]The following example grammar is not intended to exactly describe the current Modelica specification. The aim is only to outline the principle of such grammar in order to describe the abstract syntax approach.

$$
\begin{array}{rcl}
\mathit{connector} & ::= & \texttt{\textbf{Connector(}} \\
& & \{\texttt{\textbf{Extends(}} C_r\ conModification\texttt{\textbf{)}}\} \\
& & \{\texttt{\textbf{DeclCon(}} modifiability\ outinner\ C_d\ connector\texttt{\textbf{)}}\} \\
& & \{\texttt{\textbf{DeclRec(}} modifiability\ outinner\ R_d\ record\texttt{\textbf{)}}\} \\
& & \{\texttt{\textbf{CompCon(}} conconstraint\ C_r\ c_d\ conModification\texttt{\textbf{)}}\} \\
& & \{\texttt{\textbf{CompRec(}} conconstraint\ R_r\ r_d\ recModification\texttt{\textbf{)}}\} \\
& & \{\texttt{\textbf{CompInt(}} conconstraint\ x_d\texttt{\textbf{)}}\} \\
& & \{\texttt{\textbf{CompReal(}} conconstraint\ flowprefix\ y_d\texttt{\textbf{)}}\} \\
& & \texttt{\textbf{)}} \\
access & ::= & \texttt{\textbf{Public}} \mid \texttt{\textbf{Protected}} \\
modifiability & ::= & \texttt{\textbf{Replaceable}} \mid \texttt{\textbf{Final}} \\
outinner & ::= & \texttt{\textbf{Outer}} \mid \texttt{\textbf{Inner}} \mid \texttt{\textbf{OuterInner}} \mid \texttt{\textbf{NotOuterInner}} \\
conconstraint & ::= & \texttt{\textbf{Input}} \mid \texttt{\textbf{Output}} \mid \texttt{\textbf{InputOutput}} \\
flowprefix & ::= & \texttt{\textbf{Flow}} \mid \texttt{\textbf{NonFlow}}
\end{array}
$$

Figure 3.3: *Example of a grammar for the connector non-terminal. Non-terminals conModification and recModification is not defined in the figure and can be assumed to define the connector and record modifications respectively.*

The grammar is extended with a more abstract notation of *metavariables*, which range over names or identifiers. Metavariables C_d and R_d range over identifiers declaring a new connector respectively record; C_r and R_r range over connector and record names referencing an already declared connector or record. Metavariables c_d, r_d, x_d, and y_d range over *component* identifiers having the type of connector, record, Integer, and Real. All bold strings denote a node in the AST. If the AST is given in a concrete textual representation, these keywords are used when performing a pre-order traversal of the tree.

In the example, *connector* can hold zero or many `extends` nodes, referencing the meta-variable C_r, denoting all names that reference a declared connector. Hence, using this meta-variable notation, this rule states that a connector is only allowed to inherit from another connector.

Furthermore, the example shows that a connector is allowed to have two kinds of local classes: Connector and Record (nodes `DeclCon` and `DeclRec`). `CompCon` and `CompRec` state that a connector can have both connector and record components.

For each of the different kinds of elements, it is stated exactly which prefixes that are allowed. This description is more restrictive than the concrete syntax, which basically allows any prefix. In the current specification these restrictions are stated in natural languages, spread out over the specification. For example, on one page it is stated *"Variables declared with the flow type prefix shall be a subtype of Real"*. Such a text is superfluous when the grammar for S-AST is specified (note that *flowprefix* is only available in the `CompReal` node).

3.4 Chapter Summary and Conclusions

In this chapter we have briefly discussed the idea of finding a middle way between an informal language specification described in natural language and a formal definition. The main idea is to describe several intermediate languages for the Modelica simulation process, and to give an abstract syntax for each intermediate language.

We presented the ideas in this chapter in 2007 [24] as an attempt to seek new ways of improving the Modelica specification. The target audience was the Modelica design group and others interested in the design of the Modelica language. However, until this date no further work has been conducted to formalize the specification in this way.

4

Growing the Modelica Language

EQUATION-BASED OBJECT-ORIENTED (EOO) modeling languages are typically rather complex. In the hypothetical ideal case, a language can be defined once and subsequently for all future fulfill all demands a user might require regarding expressiveness, performance, and safety. Unfortunately, this is never the case. Language theory is one of the core areas within computer science, and history has shown that language design is a very difficult task and that there is no simple solution to design a language that covers all problem domains at once. In a famous talk by Guy L. Steele, he discusses the essence of designing a language for the future [136, 138]:

> "If I want to help other persons to write all sorts of programs, should I design a small programming language or a large one? I stand on this claim: I should not design a small language, and I should not design a large one. I need to design a language that can grow. I need to plan ways in which it might grow - but I need, too, to leave some choices so that other persons can make those choices at a later time."

The design space and problems of growing an equation-based object-oriented (EOO) modeling language have much in common with the design of a general purpose programming language. However, there are also several aspects where EOO languages differ, raising new questions and design problems.

This chapter discusses and analyzes how EOO languages in general can be designed for growth, and in particular how this relates to the evolution of the Modelica language. We do not present any technical contribution, but a systematic categorization of how a language can grow. The rest of the chapter is organized as follows:

- The design space of how an EOO language can grow is outlined using a new matrix model that categorizes different ways of growth (Section 4.1).

- The trade-offs for different ways of growth are discussed and analyzed from various stakeholder' perspective (Section 4.2).

4.1 Different Ways of Growing a Language

A language can grow in many different ways and directions. However, in the end, it is all about changing the language's syntax and/or semantics. In this section, we categorize and exemplify different ways of growing a language.

4.1.1 The Ways of Growth Matrix

The relationship between syntax and semantics regarding language growth is illustrated in Figure 4.1.

		Extending the Semantics	
		yes	no
Extending the Syntax	yes	*"growth by adding new language features"*	*"growth by adding syntactic sugar"*
	no	*"growth by new meanings of annotations or built-in functions"*	*"growth by new user defined abstractions"*

Figure 4.1: *Categorization of different ways of growth depending on whether the language is extended by syntax and/or semantics.*

This matrix shows the different ways of growth, whether a language is extended with its syntax or semantics, both, or none of them. The following sub-sections describe these ways of growth by giving examples from the Modelica language.

4.1.2 Growth by Adding New Language Features

The most obvious one is given in the upper left corner of Figure 4.1, i.e., extending both the syntax and the semantics. This is the ordinary way of adding a new language feature, where the new language construct is added to the syntax grammar and the new semantics for this construct is defined.

For example, lookup of variables in Modelica can be according to lexical scope and scope defined by instance hierarchy. The latter was added by defining new syntax where variables could be defined to be `inner` or `outer`. For example, consider Figure 4.2: Inside model `N` two instances of model `M` are created, namely `m1` and `m2`. Besides the syntactic extension needed for this language feature extension, the meaning of a variable declared as `inner` and `outer` must be defined.

```
model M              model N
  outer Real x;        inner Real x;
  ...                  M m1, m2;
end M;                 ...
                     end N;
```

Figure 4.2: *Example of the rules for using inner/outer.*

Both the dynamic and the static semantics must be defined. The dynamic semantics can be seen as the meaning of the actual scoping. From the specification [104], the definition is:

> *"An element declared with the prefix outer references an element instance with the same name but using the prefix inner which is nearest in the enclosing instance hierarchy of the outer element declaration."*

For example, when component `m1` is elaborated, it is discovered that `x` defined in `M` is an `outer` element. Hence, it looks up the variable with same name (in this case `x`) nearest in the instance hierarchy, which is the `x` declared as `inner` in model `N`. Hence, `N.x`, `N.m1.x`, and `N.m2.x` are the same variable. The static semantics define the type system, e.g., if `N.x` is an integer but `M.x` is a real, a conflict exists.

4.1.3 Growth by Adding Syntactic Sugar

Another approach of growing a language is to extend the syntax, but to leave the semantics as it is (the upper right corner of Figure 4.1). This way of extending a language is often referred to as adding *syntactic sugar*. What does this mean?

Basically, the idea is that neither the dynamic nor the static semantics are changed. Instead, only the grammar for the concrete syntax is extended, but not the abstract syntax. A transformation rule from the new syntax to the abstract syntax is then defined. Hence, the core of the semantics is left unchanged, but a new syntactic form is added (the syntactic sugar).

Let us explain the idea with a concrete example. Consider the four Modelica models `M1a`, `M1b`, `M1c`, and `M1d` given in Figure 4.3.
All four models state a simple initial value problem, with a slight difference in their definitions. Are all these models stating exactly the same problem, i.e., are the model's meaning the same?

The simulation result for the first three models are the same, but model `M1d` does not compile. In model `M1a` and `M1c` the `start` attribute states that the initial condition for `x` is 5 at time 0. In Modelica specification version 2.0, the procedure for specifying initial conditions were changed, and the ability to add a `initial equation` section was added [93]. Examples `M1b` and `M1d` both show that the initial equation `x = 5` is used instead of the `start` attribute. Hence, the same meaning for initial conditions can be specified in different syntactic ways. Would it not be possible to just specify the meaning of one form, and then add the other form as syntactic sugar?

```
model M1a
  Real x(start=5);
equation
  der(x) = -x + 2;
end M1a;
```

```
model M1b
  Real x;
equation
  der(x) = -x + 2;
initial equation
  x = 5;
end M1b;
```

```
model M1c
   Real x(start = 5,
          fixed=true);
equation
  der(x) = -x + 2;
end M1c;
```

```
model M1d
  Real x(fixed =
            true);
equation
  der(x) = -x + 2;
initial equation
  x = 5;
end M1d;
```

Figure 4.3: *Four almost identical Modelica models stating a simple initial value problem.*

Yes it would be possible, if it was not for the additional special attribute `fixed`, which was introduced in the language before the initial conditions. The intuitive meaning of `fixed` is that if it is true, then the corresponding `start` attribute must hold during the initialization (`M1c`). This is equivalent to an explicit initial equation (`M1b`). However, if `fixed` is false, the start attribute is treated as a *guess* value, i.e., the solver can use it as an initial guess, but it does not need to be the initial value. This is the case in model `M1a` because variables in Modelica have as default `fixed = false`. Why can we not compile `M1d` then? The reason is that the attribute `fixed` does not concern the initial equation here, but the `start` attribute. In Modelica, all variables of type real have as default `start = 0`. Hence, in the case `M1d`, the initial condition states that `x` must be both equal to 0 and to 5 at the same time, i.e., the initial condition is over-determined and cannot be solved.

Now, consider Figure 4.4, which shows three potential ways of modeling a steady state initialization:

Model `M2c` is using initial equations for modeling the steady state, which is one of the motivations of introducing initial equations. However, as can be seen in the first two models, it can also be modeled by using the start attribute together with a new algebraic variable `dx`. Models `M2b` and `M2c` always give a steady state initialization, but `M2a` depends on if the tool chooses to use the start value of `dx` as initial value (which turned out to not be the case in the tested Dymola [45] environment).

With the current design of Modelica, it is not trivial to define for example the `start` attribute as syntactic sugar of initial equations. However, if it was possible, one can argue that the *meaning* of the start attribute would be easier to grasp, both for an end user and a compiler engineer. As shown in Figure 4.4, it is also possible to define steady state initialization, by using the start attribute together with ordinary equations.

```
model M2a                    model M2b
  Real x;                      Real x;
  Real dx(start=0);            Real dx(start=0, fixed=true);
equation                     equation
  der(x) = -x + 2;             der(x) = -x + 2;
  der(x) = dx;                 der(x) = dx;
end M2a;                     end M2b;


                             model M2c
                               Real x;
                             equation
                               der(x) = -x + 2;
                             initial equation
                               der(x) = 0;
                             end M2c;
```

Figure 4.4: *Steady state initialization modeled in three different ways.*

If initial equations were the basic primitive construct (part of the AST) and the start attribute was added later as syntactic sugar, the language would have grown without changing the semantics. However, as it turned out in the Modelica case, the initial equations were added afterwards, resulting in that both the semantics and the syntax needed to be changed.

4.1.4 Growth by New Meanings of Annotations or Built-in Functions

A somewhat more unusual way to grow a language is to extend the semantics *without* changing the syntax (the lower left corner of Figure 4.1). Hence, this approach changes the meaning of programs without the need to update the grammar for the concrete syntax of the language. How is this possible?

One way of achieving this has been done in the Modelica language using built-in functions, e.g., `sin(x)`, `cos(x)`, `floor(x)`, `delay(expr,delayTime)` etc., that all adhere to the standard function syntax. The semantics of such a functions are informally described in the specification using natural language. Hence, the semantics is extended without changing the syntax.

Modelica uses a sophisticated approach for this called *annotations*. Annotations can be used for storing various extra information about models, such as graphics, version information, or documentation.

In the latest specification, a number of annotations are standardized (i.e., the meaning (semantics) of them are specified). However, vendor tools are free to add their own annotations, as long as the annotations names start with the company's name. For example, Figure 4.5 shows an example of a vendor specific graphical annotation:

```
annotation (
  Icon(coordinateSystem(extent={{-100,-100}, {100,100}}),
       graphics={__NameOfVendor(Circle(center={0,0}, radius=10))})
);
```

Figure 4.5: *A vendor specific annotation for a circle [103].*

4.1.5 Growth by New User Defined Abstractions

The lower right corner of Figure 4.1 assumes that neither the syntax nor the semantics is extended. How is it then possible to grow the language at all?

This is actually a very fundamental and natural way that has been part of programming language history from the beginning. The key point is that the user can grow the language by adding new words and meaning without altering the language definition itself. In a functional language it is done by defining new functions, in object-oriented languages by adding class definitions or method definitions to existing classes. In many languages, these new abstractions can be collected into *libraries*, enabling reuse at a later time.

In Modelica, the user can grow the set of new user defined abstractions by adding definitions of functions, classes, models, blocks etc. and then encapsulate them into packages. Hence, growth by new user abstractions is the natural way of programming/modeling, where library developers develop libraries that can later on be reused by other users. Although this principle is natural and obviously beneficial, it is far from trivial to create a language that enables this growth.

A key point, also emphasized in Steele's speech [136], is that new definitions defined in libraries should look like primitives in the language itself. Hence, in the ideal case, a user of a language should not be able to distinguish if the language has been extended with new functionality via a library definition, or by changes in the language specification. One early programming language that achieved this is LISP [137]. In this language, new definitions defined by users look like primitives and all primitives look like definitions by users. Hence, LISP is a language truly built for growth by its users.

4.1.6 Restricting the Language

In the previous sections, four different categories for extending the semantics and/or the syntax were given. In these scenarios, the language grows by providing more expressiveness, i.e., that new models or programs can be expressed that were not possible before, or that the same models can be defined in a more concise manner. However, how does a language's safety aspect grow, i.e., how can the language be improved for detecting errors, isolating faults in models, and possibly guaranteeing the absence of certain kind of faults in models?

The safety aspect of a language can actually grow by restricting the language, i.e., by defining rules that reject models as illegal. This can be defined by restricting the grammar (the syntax) or by adding semantic rules, e.g., using a static type system to define legal models (the semantics).

Our previous work on determining over- and under-constrained models by extending

the static type system of an EOO language [28] is an example of such an approach. Parts of these ideas have later also been included in the latest Modelica specification [103]. This is one of the major changes in Modelica version 3.0, where Modelica models are required to be balanced, i.e., to have the same number of equations as unknowns. A more detailed overview and rationale of balanced models in Modelica is given by Olsson *et. al.* [116].

One major implication of growth when restricting the language is backwards compatibility. Unavoidably, models that were earlier legal will become illegal in later language versions. As long as the illegal models were in fact useless models, e.g., models that were not possible to simulate, this backwards incompatibility could be acceptable. However, to reject legal working models are of course more controversial. Regarding balanced models, it has been argued in the Modelica design group, that it is now possible to check libraries and detect errors earlier and therefore enable the user to build larger models with less effort.

4.2 The Right Way to Grow

Which is the right way to grow? The right way to grow a modeling or programming language is not always the easiest way. The easy way is not always easy for everyone. We will in this section discuss and analyze the benefits and drawbacks of the various ways of growth from different stakeholders' perspective.

4.2.1 Stakeholders of an Object-Oriented Equation-Based Modeling Language

The design and evolution of a language for modeling and analysis of systems is affected by several different stakeholders:

- *Language Designers.* Person(s) inventing and designing the actual language.
- *End Users.* The users who use the language for modeling and analysis. In the Modelica case, these are usually engineers who create the model mainly using the graphical component-based drag-and-drop user interface.
- *Library Users.* Engineers and scientists who develop reusable model libraries. Libraries are created by editing textual Modelica code. The free Modelica standard library is one example.
- *Tool Vendors.* Computer scientists and computer engineers who develop the compiler and tools for viewing, editing, compiling, and executing models.

Each of these stakeholders have different demands and priorities regarding what is important when growing the language.

4.2.2 Language Designers' Perspective

Unfortunately, language designers tend to want their language to be able to handle everything. One of their main challenges is not what to put in the language, but what to actually

leave out. If a language is designed by one person or a small group, these individual(s) need to judge, test and take all decisions by themselves. This may lead to a concise design, but there is a considerable risk that important input from other stakeholder's, such as end users becomes limited. On the other hand, if the language is designed by a community with a committee, input comes from many sources. However, there is a substantial risk that the different parties involved will lead to many compromises that can make the language large and complex. The latter approach with a design community and committee is the path that the design of Modelica has followed.

When many parties are involved in the language the risk is that new "features" are continuously added to the specification, i.e., the upper left corner of the matrix in Figure 4.1 where both the syntax and semantics are changed. However, if fewer people are involved in the process, the language may be designed with a more well-defined core semantics and large parts are defined by using the approach with syntactic sugar. This is the way that for example Standard ML is defined [98]. This way of defining a language is hard and challenging, but can if done right lead to a less ambiguous specification. See [24] for further discussions regarding this topic.

Finally, one of the language designers incentives, that is often forgotten, is the need for change. If the language is completed, their role is not needed anymore.

4.2.3 End Users' Perspective

From an end user's perspective it is of course very important that the language is easy to use and understand. Moreover, the semantics that the language actually has must be close to what a fairly new user of the language expects. A clear core semantics is of course beneficial when using syntactic sugar that clearly states similar constructs' meaning. Hence, situations as described previously about initial equations in Modelica should if possible be avoided.

If a user makes mistakes, i.e., creates errors, it is of high importance that the errors can be detected and that the faults in the model can be isolated and resolved. However, restricting the language so that working models become unusable (i.e., non backwards compatibility) is generally not acceptable. Hence, from an end user's perspective, language changes that restrict the language should preferably be done very early in the language's history.

End users will of course also be able to solve new problems and use existing models in different ways. Even though the Modelica language is primarily designed for simulation, there are several other kinds of analysis that are important, such as applications for automatic control [34] and optimization problems [79].

4.2.4 Library Users' Perspective

The library user wants expressiveness. In the ideal case, the library user can grow the language by himself/herself, by adding new functionality which is indistinguishable from primitive language constructs.

Library users may have conflicting interests with both language designers and tool vendors because complications and details about the language is not the primary focus for the user. Hence, library users are typically stakeholders who want to continuously expand

and add new complex features into the language, so that it becomes more expressive for their needs (adding both new syntax and semantics).

4.2.5 Tool Vendors' Perspective

Tool vendors create tools based on their interpretation of the language specification. Hence, one of the fundamental needs for a tool vendor is that the specification can be interpreted unambiguously. The specification must be easy to read, which is the case for an informal specification written in a natural language. However, it also needs to be precise and not open for different interpretations.

The approach of using a core semantics and define large parts of the language using syntactic sugar potentially gives a middle way. For example, the built-in `edge(b)` operator is defined to be equal to

```
(b and not pre(b))
```

Hence, parts that are defined as built-in operators can in fact be treated as syntactic sugar.

Finally, a perspective that should not be forgotten is the tool vendor's commercial perspective. i.e., their focus is primarily their sales possibilities, their customers' needs, and making their customers dependent on their tools. This is indicated by the fact that tool vendors often want to be different compared to their competitors. Hence, this can be a conflict of interest with the language designers because tool vendors might not always want to be 100% compatible with competitors.

4.3 Chapter Summary and Conclusions

A programming language in general and an equation-based object-oriented modeling language in particular cannot be designed once and for all. Hence, there is a need to plan for the language to grow.

We have in this chapter categorized ways of growing a language, by either extending the semantics and/or the syntax. Moreover, we have listed how different stakeholders have different perspectives on what is important when growing a language. The importance of the different ways of growing can be summarized as follows:

- *Growth by adding new language features.* Always changing both the syntax and the semantics is the most drastic kind of change of a language and should be minimized or avoided, especially for mature and widely used languages. The stakeholders that are most negatively affected of such changes are language designers and tool vendors, while library users might be the ones that push most for such extensions.

- *Growth by adding syntactic sugar.* Extending only the syntax by using syntactic sugar and at the same time keeping a core semantics is one of the preferable approaches to language growth. It gives both a precise language definition for the tool vendors as well as an understandable language for the user.

- *Growth by new meanings of annotations or built-in functions.* Growth by only changing the semantics and not the syntax might first seem to be a very attractive

approach, especially for language designers because few changes are needed in the specification. However, it can also be dangerous, e.g., in cases where many tool dependent annotations might make different tools incompatible.

- *Growth by new user defined abstractions.* Finally, growth by user defined abstractions, i.e., neither the syntax nor the semantics are changed, is the preferable approach in the long term. However, it is far from obvious how to achieve this, especially in such a young language research area as equation-based object-oriented languages.

5

Types in the Modelica Language

ONE long term goal of modeling and simulation languages is to give engineers the possibility to discover modeling errors at an early stage, i.e., to discover problems in the model during design and not after simulation. This kind of verification is traditionally accomplished by the use of *types* in the language, where the process of checking for such errors by the compiler is called *type checking*. However, the concept of types is often not very well understood outside parts of the computer science community, which may result in misunderstandings when designing new languages. Why are then types important? Types in programming languages serve several important purposes such as naming of concepts, providing the compiler with information to ensure correct data manipulation, and enabling data abstraction. Almost all programming or modeling languages provide some kind of types. However, few language specifications include precise definitions of types and type systems. This may result in incompatible compilers and unexpected behavior when using the language.

The purpose of this chapter is twofold. The first part gives an overview of the concept of types, states concrete definitions, and explains how this relates to the Modelica language. Hence, the first goal is to augment the computer science perspective of language design among the individuals involved in the Modelica language design. The long-term objective of this work is to provide aids for further design considerations when developing, enhancing and simplifying the Modelica language. The intended audience is consequently engineers and computer scientists interested in the foundation of the Modelica language.

The second purpose of this chapter is to study the type concept in Modelica. The main contribution of this work is the insight that Modelica has two categories of types: class types and object types. The rest of this chapter is organized as follow:

- We outline the concept of types, subtypes, type systems, and inheritance, and how these concepts are used in Modelica and other mainstream languages (Section 5.1).

- We give an overview of the three main forms of polymorphism, and how these concepts correlate with each other and the Modelica language (Section 5.2).
- We introduce the type concept of Modelica more precisely, where we give a concrete syntax for expressing Modelica types (Section 5.3).

5.1 Types, Subtyping and Inheritance

There exist several models of representing types, where the *ideal model* [33] is one of the most well-known. In this model, there is a universe V of all values, containing all values of integers, real numbers, strings and data structures such as tuples, records and functions. Here, types are defined as sets of elements of the universe V. There is an infinite number of types, but all types are not legal types in a programming language. All legal types holding some specific property, such as being an unsigned integer. Figure 5.1 gives an example of the universe V and two types: real type and function type, where the latter has the *domain* of integer and *codomain* of boolean.

In most mainstream languages, such as Java and C++, types are *explicitly typed* by stating information in the syntax. In other languages, such as Standard ML and Haskell, a large portion of types can be *inferred* by the compiler, i.e., the compiler deduces the type from the context. This process is referred to as *type inference* and such a language is said to be *implicitly typed.* Modelica is an explicitly typed language.

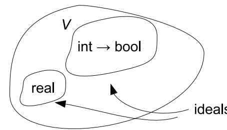


Figure 5.1: *Schematic illustration of Universe V.*

5.1.1 Language Safety and Type Systems

When a program is executed, or in the Modelica case: during simulation, different kinds of execution errors can take place. It is practical to distinguish between the following two types of runtime errors [32].

- *Untrapped errors* are errors that can go unnoticed and later cause arbitrary behavior of the system. For example, writing data out of bound of an array might not result in an immediate error, but the program might crash later during execution.
- *Trapped errors* are errors that force the computation to stop immediately; for example division by zero. The error can then be handled by the runtime system or by a language construct, such as exception handling.

A programming language is said to be *safe* if no untrapped errors are allowed to occur. These checks can be performed as *compile-time checks*, also called *static checks*, where

the compiler finds the potential errors and reports them to the programmer. Some errors, such as array out of bound errors are hard to resolve statically. Therefore, most languages are also using *runtime checks*, also called *dynamic checking*. However, note that the distinction between compile-time and runtime becomes vaguer when the language is intended for interpretation.

Typed languages can enforce language safety by making sure that *well-typed* programs cannot cause type errors. Such a language is often called *type safe* or *strongly typed*. This checking process is called *type checking* and can be carried out both at runtime and compile-time.

The behavior of the types in a language is expressed in a *type system*. A type system can be described informally using plain English text, or formally using *type rules*. The Modelica language specification is using the former informal approach. Formal type rules have much in common with logical inference rules, and might at first glance seem complex, but are fairly straightforward once the basic concepts are understood. Consider the following:

$$\frac{\Gamma \vdash e_1 : \text{bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(t-if)}$$

which illustrates a type rule for the following Modelica `if`-expression:

```
if e1 then e2 else e3
```

A type rule is written using a number of *premises* located above the horizontal line and a *conclusion* below the line. The *typing judgement* $\Gamma \vdash e : T$ means that expression e has type T with respect to a static typing environment Γ. Hence, the rule (t-if) states that *guard* e_1 must have the type of a boolean and that e_2 and e_3 must have the same type, which is also the resulting type of the `if`-expression after evaluation. This resulting type is stated in the last part of the conclusion, i.e., $: T$.

If the language is described formally, we can attempt to prove the *type soundness theorem* [151]. If the theorem holds, the type system is said to be *sound* and the language *type safe* or or just *safe*. The concept of type safety can be illustrated by Robin Milner's famous statement "Well-typed programs cannot go wrong" [97]. Modern type soundness proofs are based on Wright and Felleisen's approach where type systems are proven correct together with the language's operational semantics [151]. Using variant of this technique, informally stated, type safety holds if and only if the following two statements hold:

- *Preservation* - If an expression e has a type T and e evaluates to a value v, then v also has type T.

- *Progress* - If an expression e has a type T then either e evaluates to a new expression e' or e is a value. This means that a well typed program never gets "stuck", i.e., it cannot go into a undefined state where no further evaluations are possible.

Note that the above properties of type safety correspond to our previous description of absence of untrapped errors. For example, if a division by zero error occurs, and the semantics for such event is undefined, the progress property will not hold, i.e., the evaluation

gets "stuck", or enters an undefined state. However, if dynamic semantics are defined for throwing an exception when the division by zero operation occurs, the progress property holds.

For the imperative and functional parts of the Modelica language, the safety concept corresponds to the same methodology as other languages, such as Standard ML. However, for the instantiation process of models, the correspondence to the progress and preservation properties are not obvious.

Table 5.1 lists a number of programming languages and their properties of being type safe [32, 99]. The table indicates if the languages are primarily designed to be checked statically at compile-time or dynamically at runtime. However, the languages stated to be statically type checked typically still perform some checking at runtime. Although many of the languages are commonly believed to be safe, few have been formally proven to be so.

Language	**Type Safe**	**Checking**
Standard ML	yes	static
Java	yes	static
Common LISP	yes	dynamic
Modelica	yes	static
C/C++	no	static
Assembler	no	-

Table 5.1: *Believed type safety of selected languages.*

One can argue whether Modelica is statically or dynamically checked, depending on how the terms compile-time and runtime are defined. Furthermore, because no exception handling is currently part of the language, semantics for handling dynamic errors such as array out of bounds is not defined in the language and is therefore considered a compiler implementation issue. Hence, the Modelica language can *only* be regarded to be safe if the tool unconditionally detects all errors and terminates the computation with an error message.

5.1.2 Subtyping

Subtyping is a fundamental language concept used in most modern programming languages. It means that if a type S has all the properties of another type T, then S can be safely used in all contexts where type T is expected. This view of subtyping is often called *the principle of safe substitution* [124]. In this case, S is said to be a subtype of T, which is written as

$$S <: T \tag{5.1}$$

This relation can be described using the following important type rule called the *rule of subsumption.*

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T} \text{ (t-sub)}$$

The rule states that if $S <: T$, then every *term*[1] t of type S is also a term of type T. This shows a special form of *polymorphism*, which we will further explore in Section 5.2.

5.1.3 Inheritance

Inheritance is a fundamental language concept found in basically all class based *Object-Oriented (OO)* languages. From an existing *base class*, a new *subclass* can be created by *extending* from the base class, resulting in the subclass *inheriting* all properties from the base class. One of the main purposes with inheritance is to save programming and maintenance efforts of duplicating and reading duplicates of code. Inheritance can in principle be seen as an implicit code duplication which in some circumstances implies that the subclass becomes a subtype of the type of the base class.

Figure 5.2 shows an example[2] where inheritance is used in Modelica. A *model* called `Resistor` extends a base class `TwoPin`, which includes two elements `v` for voltage and `i` for current. Furthermore, two instances `p` and `n` of connector `Pin` are public elements of `TwoPin`. Because `Resistor` extends `TwoPin`, all elements `v`, `i`, `p` and `n` are "copied" to class `Resistor`. In this case, the type of `Resistor` will also be a subtype of `TwoPin`'s type.

```
connector Pin
  SI.Voltage v;
  flow SI.Current i;
end Pin;

partial model TwoPin
  SI.Voltage v;
  SI.Current i;
  Pin p, n;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;

model Resistor
  extends TwoPin;
  parameter SI.Resistance R=100;
equation
  R*i = v;
end Resistor;
```

Figure 5.2: *Example of inheritance in Modelica, where a new subclass* `Resistor` *is created by extending the base class* `TwoPin`*.*

[1]The word *term* is commonly used in the literature as an interchangeable name for expression.

[2]These classes are available in the Modelica Standard Library 2.2, but are slightly modified for reasons of readability.

However, a common misunderstanding is that subtyping and inheritance is the same concept [99]. A simple informal distinction is to say that "subtyping is a relation on interfaces", but "inheritance is a relation on implementations". In the resistor example, not only the public elements `v`, `i`, `p` and `n` will be part of class `Resistor`, but also the meaning of this class, i.e, the equations `v = p.v - n.v`, `0 = p.i + n.i` and `i = p.i`.

A famous example, originally stated by Snyder [135], illustrates the difference between subtyping and inheritance. Three common *abstract data types* for storing data objects are *queue*, *stack*, and *dequeue*. A queue normally has two operations, *insert* and *delete*, which stores and returns objects in a *first-in-first-out (FIFO)* manner. A stack has the same operations, but is using a *last-in-first out (LIFO)* principle. A dequeue can operate as both a stack and a queue, and is normally implemented as a list, which allows inserts and removals at both the front and the end of the list.

Figure 5.3 shows two C++ classes modeling the properties of a dequeue and a stack. Because the class `Dequeue` implements the properties also needed for a stack, it seems natural to create a subclass `Stack` that inherits the implementation from `Dequeue`. In C++, it is possible to use so called *private inheritance* to model inheritance with an *exclude operation*, i.e., to inherit some, but not all properties of a base class. In the example, the public methods `insFront`, `delFront`, and `delRear` in class `Dequeue` are inherited to be private in the subclass `Stack`. However, by adding new methods `insFront` and `delFront` in class `Stack`, we have created a subclass, which has the property of a stack by excluding the method `delRear`. `Stack` is obviously a subclass of `Dequeue`,

```
class Dequeue{
public:
  void insFront(int e);
  int delFront();
  int delRear();
};

class Stack : private Dequeue{
public:
  void insFront(int e)
    {Dequeue::insFront(e);}
  int delFront()
    {return Dequeue::delFront();}
};
```

Figure 5.3: *C++ example, where inheritance does not imply a subtype relationship.*

but is it a subtype? The answer is no because an instance of `Stack` cannot be safely used when `Dequeue` is expected. In fact, the opposite is true, i.e., `Dequeue` is a subtype of `Stack` and not the other way around. However, in the following section we will see that C++ does not treat such a subtype relationship as valid, but the type system of Modelica would do so.

5.1.4 Structural and Nominal Type Systems

During type checking, regardless if it takes place at compile-time or runtime, the type checking algorithm must control the relations between types to see if they are correct or not. Two of the most fundamental relations are *subtyping* and *type equivalence*.

Checking of type equivalence is the single most common operation during type checking. For example, in Modelica it is required that the left and right side of the equality in an equation have the same type, which is shown in the following type rule.

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 = e_2 : Unit} \text{ (t-equ)}$$

Note that type equivalence has nothing to do with equivalence of values, e.g., equation `4 = 10` is type correct because integers 4 and 10 are type equivalent. However, this is of course not a valid equation because the values on the right and left side are not the same.

The *Unit* type (not to confuse with physical units), shown as the resulting type of the equation, is often used as a type for uninteresting result values.

A closely related concept to type equivalence is *type declaration*, i.e., when a type is declared as a specific *name* or *identifier*. For example, the following Modelica record declaration

```
record Person
  String name;
  Integer age;
end Person;
```

declares a type with name `Person`. Some languages would treat this as a new unique type that is not equal to any other type. This is called *opaque* type declaration. In other languages, this declaration would simply mean that an alternative name is given to this type. However, the type can also be expressed by other names or without any name. This latter concept is commonly referred as *transparent* type declaration.

In a pure *nominal type system*, types are compared (subtyping and type equivalence) by using the *names* of the declared types, i.e., opaque type declarations are used. Type equivalence is controlled by checking that the same declared name is used. Furthermore, the subtype relation in OO languages is checked by validating the inheritance order between classes. The C++ language is mainly using a nominal type system, even if parts of the language does not obey the strict nominal structure.

Consider the listing in Figure 5.4, which illustrates a C++ model similar to the resistor example earlier given as Modelica code in Figure 5.2. In this case, `Resistor` is a subclass of `TwoPin` and the type of `Resistor` is therefore also a subtype of `TwoPin`'s type. However, the type of `Inductor` is not a subtype to the type of `TwoPin` because `Inductor` does not inherit from `TwoPin`. Moreover, `Resistor2` is not type equivalent to `Resistor` even if they have the same structure and inherit from the same base class because they are opaquely declared.

In a *structural type system* [124], declarations are introducing new names for type expressions, but no new types are created. Type equivalence and subtype relationship is only decided depending on the structure of the type, not the naming.

The Modelica language is inspired by the type system described by Abadi and Cardelli [2] and is using transparent type declarations, i.e., Modelica has a structural type system.

```
class Pin{
public:
  float v, i;
};

class TwoPin{
public:
  TwoPin() : v(0),i(0){};
  float v, i;
  Pin p, n;
};

class Resistor : public TwoPin{
public:
  Resistor() : r(100) {};
  float r;
};

class Resistor2 : public TwoPin{
public:
  Resistor2() : r(200) {};
  float r;
};

class Inductor{
public:
  Inductor() : v(0),i(0){};
  float v, i;
  Pin p, n;
  const float L;
};
```

Figure 5.4: *`Resistor` inheritance example in C++.*

Consider the `Resistor` example given in Figure 5.2 and the two complementary models `Inductor` and `Resistor2` in Figure 5.5. Here, the same relations hold between `TwoPin` and `Resistor`, i.e., the type of `Resistor` is a subtype of `TwoPin`'s type. The same holds between `TwoPin` and `Resistor2`. However, now `Resistor` and `Resistor2` are type equivalent because they have the same structure and naming of their public elements. Furthermore, the type of `Inductor` is now a valid subtype of `TwoPin`'s type because `Inductor` contains all public elements (type and name) of the ones available in `TwoPin`.

It is important to stress that *classes* and *types* in a structural type system are **not** the same thing, which also holds for Modelica. The type of a class represents the interface of the class relevant to the language's type rules. The type does not include implementation details, such as equations and algorithms.

Note that a nominal type system is more restrictive than a structural type system,

```
model Resistor2
  extends TwoPin;
  parameter SI.Resistance R=200;
equation
  R*i = v;
end Resistor;

model Inductor
  Pin p, n;
  SI.Voltage v;
  SI.Current i;
  parameter SI.Inductance L=1;
equation
  L*der(i) = v;
end Inductor;
```

Figure 5.5: *Complementary* `Inductor` *and* `Resistor2` *models to the example in Figure 5.2.*

i.e., two types that have a structured subtype relation can always have a subtype relation by names (if the language's semantics allows it). However, the opposite is not always true. Recall the `Dequeue` example listed in Figure 5.3. The class `Stack` has a subclass relation to `Dequeue`, but a subtype relation cannot be enforced, due to the structure of the class. The converse could be true, but the type system of C++ would not allow it because it is nominal and subtype relationships are based on names. Hence, a structural type system can be seen as more *expressive* and *flexible* compared to a nominal one, even if both gives the same level of language type safety.

5.2 Polymorphism

A type system can be *monomorphic* in which each value can belong to at most one type. A type system, as illustrated in Figure 5.1, consisting of the distinct types function, integer, real, and boolean is a monomorphic type system. Conversely, in a *polymorphic* type system, each value can belong to many different types. Languages supporting polymorphism are in general more expressive compared to languages only supporting monomorphic types. The concept of polymorphism can be handled in various forms and have different naming depending on the paradigm where it is used. Following John C. Mitchell's categorization, polymorphism can be divided into the following three main categories [99]:

- Subtype Polymorphism
- Parametric Polymorphism
- Ad-hoc Polymorphism

There are other similar categorizations, such as given by Cardelli and Wegner's [33], where the ad-hoc category is divided into *overloading* and *coercion* at the top level of categories.

5.2.1 Subtype Polymorphism

Subtyping is an obvious way that gives polymorphic behavior in a language. For example, an instance of `Resistor` can be represented both as a `TwoPin` type and a `Resistor` type. This statement can also be shown according to the rule of subsumption (t-sub) described in Section 5.1.2.

When a value is changed from one type to some supertype, it is said to be an *up-cast*. Up-casts can be viewed as a form of *abstraction* or *information hiding*, where parts of the value becomes invisible to the context. For example, an up-cast from `Resistor`'s type to `TwoPin`'s type hides the parameter `R`. Up-casts are always type safe, i.e., the runtime behavior cannot change due to the upcast.

However, for subtype polymorphism to be useful, typically types should be possible to *down-cast*, i.e., to change to a subtype of a type's value. Consider function `Foo`

```
function Foo
  input TwoPin x;
  output TwoPin y;
end Foo;
```

where we assume that down-casting is allowed[3]. It is in this case valid to pass either a value of type `TwoPin` (type equivalence) or a subtype to the type of `TwoPin`. Regardless if a value of `TwoPin`'s or `Inductor`'s type is sent as input to the function, a value of `TwoPin`'s type will be returned. It is not possible for the static type system to know if this is a `TwoPin`, `Resistor` or a `Inductor` type. However, for the user of the function, it might be crucial to handle it as an `Inductor`, which is why a down-cast is necessary.

Down-casting is however not a safe operation because it might cast down to the wrong subtype. In Java [65], before version 1.5 when *generics* were introduced, this safety issue could only be handled using dynamic checks and raising dynamic exceptions if an illegal down-cast was made. Subtype polymorphism is sometimes called "poor-man's polymorphism" because it enables polymorphic behavior, but the safety of down-casts must be handled dynamically [124].

The Modelica language supports subtyping as explained previously, but does not have any operation for down-cast. Because the language does not include this unsafe operation, only a limited form of subtype polymorphism can be used with functions. For example, a function can operate on a polymorphic type as input, such as `TwoPin`, but it only makes sense to return values of a type that can be directly used by the caller.

However, subtype polymorphism is more extensively used when reusing and replacing components in models, i.e., by using the `redeclare` keyword.

[3] This function type or example is not valid in the current Modelica standard. It is used only for the purpose of demonstrating subtype polymorphism.

5.2.2 Parametric Polymorphism

The term *parametric polymorphism* means that functions or classes can have *type parameters*, to which types or *type expressions* can be supplied. The term parametric polymorphism is often used in functional language communities, while people related to object-oriented languages tend to use the term *generics*.

The C++ *template* mechanism is an example of *explicit parametric polymorphism*, where the type parameter must be explicitly declared. Consider for example Figure 5.6, where a template function `swap` is implemented. The type parameter `T` must be explicitly stated when declaring the function. However, the type argument is not needed when calling the function, e.g., both `int x,y; swap(x,y);` and `float i,j; swap(i,j)` are valid uses of the function.

```
template<typename T>
void swap(T& x, T& y){
  T tmp = x;
  x = y;
  y = tmp;
}
```

Figure 5.6: *Explicit parametric polymorphism in C++.*

Standard ML on the other hand is making use of *implicit parametric polymorphism*, where the type parameters do not need to be explicitly stated when declaring the function. Instead, the *type inference algorithm* computes when type parameters are needed.

A notable difference of parametric and subtype polymorphism is that all type checking of parametric polymorphism can take place at compile-time and no unsafe down-cast operation is needed.

Standard ML and and C++ are internally handling parametric polymorphism quite differently. In C++ templates, instantiation to compiled code of a function is done at link time. If for example function `swap` is called both using `int` and `float`, different code for implementations and calls of the function is generated for the two function calls. Standard ML on the other hand is using a *uniform data representation*, where all data objects are represented internally as pointers/references to objects. Therefore, there is no need to create different copies of code for different types of arguments.

Modelica can be seen to support a variant of parametric polymorphism, by using the *redeclare* construct on class declarations.

5.2.3 Ad-hoc Polymorphism

In parametric polymorphism the purpose is to declare one implementation that can be used with different types of arguments. *Ad-hoc polymorphism*, by contrast, allows a polymorphic value to be used differently depending on which type the value is viewed to have.

There are several language concepts that fall under the concept of ad-hoc polymorphism [33], where *Overloading* and *Coercion* are most notable. Other related concepts

that also fall under this category are Java's `instanceOf` concept and different form of *pattern matching* [124].

Overloading

A symbol is *overloaded* if it has two or more meanings, which are distinguished by using types. That is, a single function symbol or identifier is associated with several implementations.

An example of overloading that exists in many programming languages is *operator overloading* for built in types. For example, the symbol + is using infix notation and has two operands associated with it. The type of these operands decides how the operation should be carried out, i.e., which implementation should be used.

Overloading can take place at either compile-time or at runtime. Overloading used at runtime is often referred to as *dynamic lookup* [99], *dynamic dispatch* or *multi-method dispatch*. In most cases, the single term overloading refers to static overloading taking place at compile-time. The distinction becomes of course vague, if the language is *interpreted* and not compiled.

Another form of overloading available in some languages is user-defined *function overloading*, where a function identifier can represent several implementations for different type arguments. Modelica is currently not supporting any form of user defined overloading.

Coercion

Another form of ad-hoc polymorphism is *coercion* or *implicit type conversion*, which is runtime conversion between types, typically performed by code automatically inserted by the compiler. The distinction between overloading and type coercion is not always clear, and the two concepts are strongly related. Consider the following four expressions of multiplication [33]:

```
7   * 9   //Integer * Integer
6.0 * 9.1 //Real * Real
6   * 5.2 //Integer * Real
6.0 * 8   //Real * Integer
```

All four of these expressions are valid Modelica expressions, but they can in the context of coercion and overloading be interpreted in three different ways:

- The multiplication operator is overloaded four times, one for each of the four expressions.
- The operator is overloaded twice; one for each of the the first two expressions. If the arguments have different types, i.e., one is `Real` and the other one `Integer`, type coercion is first performed to convert the arguments to `Real`.
- Arguments are always implicitly converted to `Real`, and the operator is only defined for `Real`s.

Type conversions can also be made *explicit*, i.e., code is inserted manually by the programmer that converts the expression to the correct type.

In Modelica, implicit type conversion is used when converting from `Integer` to `Real`. Of the three different cases listed above, the second one applies to the current Modelica 3.2 standard.

5.3 Modelica Types

In the previous sections we described different aspects of types for various languages. In this section we will present a concrete syntax for describing Modelica types, followed by rules stating legal type expressions for the language.

The current Modelica language specification [104] specifies a formal syntax of the language, but the semantics including the type system are given informally using plain English. There is no explicit definition of the type system, but an implicit description can be derived by reading the text describing relations between types and classes in the Modelica specification. This kind of implicit specification makes the actual specification open for interpretation, which may result in incompatible compilers; both between each other, but also to the specification itself. Our work in this section should be seen as a first step to formalize what a type in Modelica actually is. Previous work has been performed to formally specify the semantics of the language [85]. The formal specification of the semantics includes the meaning of a Modelica type. However, the earlier work was not as precise as a distinction between class type and object type was not made.

Why is it then so important to have a precise definition of the types in a language? As we have described earlier, a type can be seen as an interface to a class or an object. The concept of interfaces forms the basis for the widely accepted approach of separating *specification* from *implementation*, which is particularly important in large scale development projects. To put it in a Modelica modeling context, let us consider a modeling project of a car, where different modeling teams are working on the wheels, gearbox and the engine. Each team has committed to provide a set of specific attributes for their component, which specifies the interface. The contract between the teams is not violated, as long as the individual teams are following this commitment of interface (the specification) by adding / removing equations (the implementation). Because the types state the interfaces in a language with a structural type system, such as Modelica, it is obviously decisive that they have a precise definition.

Our aim here is to define a precise notation of types for a subset of the Modelica language, which can then further be extended to the whole language. Because the Modelica language specification is open for interpretation, the presented type definition is our interpretation of the specification.

5.3.1 Class Types and Object Types

Now, let us study the types of some concrete Modelica models. Consider the following model `B`, which is rather uninteresting from a physical point of view, but demonstrates some key concepts regarding types.

```
model B
  parameter Real s=-0.5;
  connector C
    flow Real p;
    Real q;
  end C;
protected
  Real x(start=1);
equation
  der(x) = s*x;
end B;
```

What is the type of model `B`? Furthermore, if `B` was used and instantiated as a component in another model, e.g., `B b;`, what would the resulting type for element `b` be? Would the type for `B` and `b` be the same? The answer to the last question is definitely no. Consider the following listing, which illustrates the type of model `B`.

```
model classtype //Class type of model B
  public parameter Real objtype s;
  public connector classtype
    flow Real objtype p;
    nonflow Real objtype q;
  end C;
  protected Real objtype x;
end
```

This type listing follows the grammar syntax listed in Figure 5.7. The first thing to notice is that the name of model `B` is not visible in the type. Recall that Modelica is using a structural type system, where the types are determined by the structure and not the names, i.e., the type of model `B` has nothing to do with the name `B`. However, the names of the *elements* in a type are part of the type, as we can see for parameter `s` and variable `x`.

The second thing to observe is that the equation part of the model is missing in the type definition. The reason for this is that equations and algorithms are part of the implementation and not the model interface. Moreover, all elements `s`, `C` and `x` are preserved in the type, but the keywords `model`, `connector` and basic type `Real` are followed by new keywords `classtype` or `objtype`. This is one of the most important observations to make regarding types in a class based system using structural subtyping and type equivalence. As we can see in the example, the type of model `B` is a *class type*, but parameter `s` is an *object type*. Simply stated: A class type is the type of one of Modelica's restricted classes, such as `model`, `connector`, `record` etc., but an *object type* is the type of an instance of a class, i.e., an object. Now, the following shows the object type of `b`, where `b` represents an instance of model `B`:

```
model objtype //Object type of b
  parameter Real objtype s;
end
```

Obviously, both the type of connector `C` and variable `x` have been removed from the type of `b`. The reason is that an object is a runtime entity, where neither local classes (connector `C`) nor protected elements (variable `x`) are accessible from outside the instance. However, note that this is not the same as that variable `x` does not exist in a instance of `B`; it only means that it is not visible to the outside world.

Now, the following basic distinctions can be made between *class types* and *object types*:

- Classes can inherit (using extends) from class types, i.e., the type that is bound to the name used in an `extends` clause must be a class type and not an object type.
- Class types can contain both object types and class types, but object types can only hold other object types.
- Class types can contain types of protected elements; object types cannot.
- Class types are used for compile time evaluation, such as inheritance and redeclarations.

Let us now take a closer look at the grammar listed in Figure 5.7. The root non-terminal of the grammar is $type$, which can form a class or object type of the restricted classes or the built in types `Real`, `Integer`, `Boolean`, `String`, or `enumeration`. The grammar is given using a variant of *Extended Backus-Naur Form* (EBNF), where terms enclosed in brackets {} denote zero, one or more repetitions. Keywords appearing in the concrete syntax are given in bold font. All prefixes, such as `public`, `flow`, `outer` etc. can be given infinitely many times. The correct use of these prefixes is not enforced by the grammar, and must therefore be handled later in the semantic analysis. We will give guidelines for default prefixes and restrictions of the use of prefixes in the next subsection.

Now, let us introduce another model `A`, which extends model `B`:

```
model A
  extends B(s=4);
  C c1;
equation
 c1.q = -10*der(x);
end A;
```

The question is now what the type of model `A` is and if it is instantiated to an object, i.e., `A a;`, what is then the type of `a`? The following shows the type of model `A`.

```
model classtype //Class type of A
  public parameter Real objtype s;
  public connector classtype
    flow Real objtype p;
    nonflow Real objtype q;
  end C;
  public connector objtype
    flow Real objtype p;
    nonflow Real objtype q;
  end c1;
  protected Real objtype x;
end
```

First of all, we see that the type of model `A` does not include any `extends` keyword referring to the inherited model `B`. Because Modelica has a structural type system, it is the structure that is interesting, and thus a type only contains the collapsed structure of

```
type          ::= (model | record | connector |
                   block | function | package)
                   kindoftype
                   {{prefix} type identifier ;} end
                | (Real | Integer | Boolean |
                   String) kindoftype
                | enumeration kindoftype
                   enumlist
kindoftype    ::= classtype | objtype
prefix        ::= access | causality |
                   flowprefix | modifiability |
                   variability | outerinner
enumlist      ::= ( identifier {, identifier} )
access        ::= public | protected
causality     ::= input | output |
                   inputoutput
flowprefix    ::= flow | nonflow
modifiability ::= replaceable | modifiable |
                   final
variability   ::= constant | parameter |
                   discrete | continuous
outerinner    ::= outer | inner |
                   notouterinner
```

Figure 5.7: *Concrete syntax of partial Modelica types.*

inherited elements. Furthermore, we can see that the protected elements from `B` are still available, i.e., inheritance preserves the protected element after inheritance. Moreover, because model `A` contains an instance of connector `C`, this is now available as an object type for element `c1` in the class type of `A`. Finally, consider the type of an instance `a` of class `A`:

```
model objtype //Object type of a
  parameter Real objtype s;
  connector objtype
    flow Real objtype p;
    nonflow Real objtype q;
  end c1;
end
```

The protected element is now gone, along with the elements representing class types. A careful reader might have noticed that each type definition ends without a semi-colon, but elements defined inside a type such as `model classtype` ends with a semi-colon. A closer look at the grammar should make it clear that types themselves do not have names, but when part of an element definition, the type is followed by a name and a semi-colon. If type expressions were to be ended with a semi-colon, this recursive form of defining concrete types would not be possible.

5.3.2 Prefixes in Types

Elements of a Modelica class can be prefixed with different notations, such as `public`, `outer` or `replaceable`. We do not intend to describe the semantics of these prefixes here, instead we refer to the specification [102] and to the more accessible description by Fritzson [51]. Most of the Modelica language's prefixes have been introduced in the grammar in Figure 5.7. However, not all prefixes are allowed or have any semantic meaning in all contexts.

In this subsection, we present a partial definition of when different prefixes are allowed to appear in a type. In currently available tools for Modelica, such as Dymola [45] and OpenModelica [52], the enforcement of these restrictions is sparse. The reason for this can both be the difficulties to extract this information from the specification and the fact that the rules for the type prefixes are very complex.

In Figure 5.8 several abbreviations are listed. The lower case abbreviations a, c, c' etc. define sets of prefixes. The uppercase abbreviations M, R etc. together with a subscription of c for class type and o for object type, represents the type of an element part of another type. For example M_c is a model class type, and R_o is a record object type.

Now, consider the rules for allowed prefixes of elements shown in the tables given in Figure 5.9, Figure 5.10, and Figure 5.11.

In Figure 5.9 the intersection between the column (the type of an element) and the row (the type that contains this element) states the allowed prefixes for this particular element. This table shows which prefixes are allowed for a class type that is part of another class type. For example, recall the connector `C` in model `A`. When looking at the type of `A`, we have a class type (the model class type) that contains another class type (the connector class type), i.e., the allowed prefixes are given in the intersection of row 1 and

$M =$	**model**	
$R =$	**record**	
$C =$	**connector**	
$B =$	**block**	
$F =$	**function**	
$P =$	**package**	
$X =$	**Integer**, **Boolean**, **enumeration**, **String**	
$Y =$	**Real**	
$a =$	{**public**, **protected**}	Access
$a' =$	{**public**}	
$c =$	{**input**, **output**, **inputoutput**}	Causality
$c' =$	{**input**, **output**}	
$f =$	{**flow**, **nonflow**}	Flowprefix
$m =$	{**replaceable**, **modifiable**, **final**}	Modifiability
$m' =$	{**modifiable**, **final**}	
$v =$	{**constant**, **parameter** **discrete**, **continuous**}	Variability
$v' =$	{**constant**, **parameter** **discrete**}	
$v'' =$	{**constant**}	
$o =$	{**outer**, **inner**, **notouterinner**}	Outerinner

Figure 5.8: *Abbreviation for describing allowed prefixes. Default prefixes are underlined.*

	M_c	R_c	C_c	B_c	F_c	P_c	X_c	Y_c
M_c	amo	amo	amo	amo	amo	.	amo	amo
R_c	.	.	.	.	.	.	.	.
C_c	.	.	.	.	.	.	.	.
B_c	amo	amo	amo	amo	amo	.	amo	amo
F_c	.	am	.	.	am	.	am	am
P_c	am	amv''	am	am	am	$a'm$	am	am

Figure 5.9: *Prefixes allowed for elements of class type (columns) inside a class type (rows).*

column 3. In this case, *access* prefixes `public` and `protected`, *modifiability* prefixes `replaceable`, `modifiable`, and `final`, and *outer/inner* prefixes `outer`, `inner` and `notouterinner` are allowed.

We have introduced a number of new prefixes: `inputoutput`, `notouterinner`, `nonflow`, `modifiable`, and `continuous`. These new prefixes are introduced to enable a complete type definition, e.g., it should be possible to explicitly specify that a variable in a connector is not a flow variable by giving a `nonflow` prefix. However, for simplicity, sometimes it is more convenient to leave out some of the prefixes, and instead

	M_o	R_o	C_o	B_o	F_o	P_o	X_o	Y_o
M_c	amo	$acmo$	$acmo$	amo	amo	.	$acmv'o$	$acmvo$
R_c	.	mo	.	.	.	.	$mv'o$	mvo
C_c	.	mo	mo	.	.	.	m	$mcfvo$
B_c	amo	$ac'mo$	$ac'mo$	amo	amo	.	$ac'mv'o$	$ac'mvo$
F_c	.	$ac'm$	.	.	am	.	$ac'mv'$	$ac'mv$
P_c	.	amv''	.	.	.	.	amv''	amv''

Figure 5.10: *Prefixes allowed for elements of object type (columns) inside a class type (rows).*

	M_o	R_o	C_o	B_o	F_o	P_o	X_o	Y_o
M_o	o	$cm'o$	co	o	o	.	$cm'v'o$	$cm'vo$
R_o	.	$m'o$	.	.	.	.	$m'v'o$	$m'vo$
C_o	.	$m'o$	o	.	.	.	.	$cfm'vo$
B_o	o	$c'o$	$c'o$	o	o	.	$c'm'v'o$	$c'm'vo$
F_o	.	c'	.	.	.	.	$m'v'$	$m'v$
P_o	.	.	.	.	.	.	.	.

Figure 5.11: *Prefixes allowed for elements of object type (columns) inside an object type (rows).*

use default prefixes. The defined default prefixes are shown underlined in Figure 5.8. If no underlined prefix exists in a specific set, this implies that the prefix must be explicitly stated.

Analogously to the description of Figure 5.9, Figure 5.10 shows the allowed prefixes for elements of object types contained in a class type and Figure 5.11 shows object types contained in object types. There are no tables given for class types contained in object types for the simple reason that object types are not allowed to contain class types.

In some of the cells in the tables described above, a dot symbol is shown. This means that the specific type of element inside a certain type is not allowed. Hence, such a combination should not be allowed by the compiler at compile-time.

Now, let us observe some general trends between the allowed attributes. First of all, object types cannot contain class types, which is why there are only 3 tables. Secondly, access prefixes (`public`, `protected`) are only allowed in class types, which is why Figure 5.11 does not contain any abbreviation a. Thirdly, the `replaceable` prefix does not make sense in object types because redeclarations may only occur during object creation or inheritance, i.e., compile-time evaluation. Then when an object exists, the type information for replaceable is of no interest any more. Finally, we can see that package class types can hold any other class types, but no other class type can hold package types.

Note that several aspects described here are our design suggestions for simplifying and making the language more stringent from a type perspective. Currently, there are no limitations for any class to contain packages in the Modelica specification. Furthermore, there are no strict distinctions between object- and class types because elaboration and type checking are not clearly distinguished. Hence, redeclaration of elements in an object are in fact possible according to the current specification, even if it does not make sense in a class based type perspective.

5.3.3 Completeness of the Type Syntax

One might ask if this type definition is complete and includes all aspects of the Modelica language and the answer to that question is no. There are several aspects, such as arrays, partial and encapsulated classes, units, constrained types, conditional components and external functions that are left out on purpose.

The main reason for this work is to pinpoint the main structure of types in Modelica, not to formulate a complete type definition. As we can see from the previous sections, the type concept in the language is very complex and hard to define, due to the large number of exceptions and the informal description of the semantics and type system in the language specification.

The completeness and correctness of the allowed type prefixes described in the previous section depend on how the specification is interpreted. However, the notation and structure of the concrete type syntax should be consistent and is intended to form the basis for incorporating this improved type concept tighter into the language.

Finally, we would like to stress that defining types of a language should be done in parallel with the definition of precise semantic and type rules. Because the latter information is currently not available, the precise type definition is obviously not possible to validate.

5.4 Chapter Summary and Conclusions

We have in this chapter given a brief overview of the concept of types and how they relate to the Modelica language. The first part of the Chapter described types in general, and the latter sections detailed a syntax definition of how types can be expressed for the Modelica language.

6

Over- and Under-Constrained Systems of Equations

A model in an EOO language needs to have the same number of equations as unknowns. This chapter describes a novel technique to determine over- and under-constrained systems of equations in models, based on a concept called *structural constraint delta*, denoted C_Δ. Our approach makes use of static type checking and consists of a type checking algorithm, which determines if a model is under- or over-constrained without elaborating its subcomponents. This is essential if separate compilation of components is introduced in an EOO language. Furthermore, the concept also allows detection of constraint-errors at the subcomponent level and improves the possibilities of isolating the source of the errors. We have implemented it for a subset of the Modelica language, and successfully validated it on several examples. However, the idea is not limited to Modelica and should be possible to generalize to other EOO languages. The remainder of this chapter is structured as follows.

- We describe the problem and motivation for determining over- and under-constrained systems of equations (Section 6.1).
- We introduces a minimal EOO language called Featherweight Modelica (FM), its syntax and informal description of semantics and type system (Section 6.2).
- We define the concept of structural constraint delta, the algorithms used for constraint checking and debugging, and how these concepts fit into the FM language's type system (Section 6.3).
- We describe how we validate the approach by making use of a prototype implementation (Section 6.4).

The work of this chapter was published in 2006 [28]. Later revisions of the Modelica specification (version 3.0 released in 2007 [103]) included a similar concept with balanced models.

6.1 Problem and Motivation

While EOO languages provide attractive advantages, they also present new challenges in the areas of static analysis, type systems, and debugging. This chapter deals with specific problems arising with EOO languages in two areas:

- Constraint checking of separately compiled components.
- Error detection and debugging.

The continuous-time behavior of a EOO model is typically described by a DAE. The existence of a unique solution requires that the number of equations and variables (unknowns) are equal[1]. If the number of equations is greater than unknowns, the model is said to be *over-constrained*. Conversely, if the number of unknowns is greater than equations, it is *under-constrained*.

In an EOO model, variables and equations can be specified in different subcomponents of the model. To find out if a model has the same number of equations as variables, the model has traditionally been elaborated into a flat system of equations, where the number of variables and equations can be counted. However, this simple counting approach is not possible in the general case when one or more components in the model have been separately compiled.

Figure 6.1 outlines a potential architecture for separate compilation of Modelica models. Because symbolic transformation always needs to take into account the whole equation system, it is performed after the linking phase. Consider a simple model of a car, consisting of axis, gearbox, and an engine. In order to find out if the car model has the same number of equations as unknowns, we have to translate it into one large system of equations and count the number of variables and equations in that system. In the simple case the compiled engine always generate the same set of equations. However, models can also typically be parameterized with other models (using for example Modelica's `redeclare`-construct), resulting in that a separately compiled model can not always result in a flat DAE[2].

Moreover, if a model intended for simulation has not the same number of equations as variables, it is an error. This can be detected (trivially) after compiling the model into a system of equations. However, it is non-trival to isolate the fault of the error, i.e., to help the user to pinpoint where or which components the error is located in. Consider again the car model discussed above. When the model is compiled (translated into equations), the user might be presented with an error message such as: "There are 20237 equations and 20235 variables". Debugging the car model with only this message and a listing of equations and variables is extremely hard. There exist software tools [45] and methods [29] that help the user in this process, but they require information of the model's whole system of equations, i.e., the tools need the flat hybrid DAE.

[1] This means that the incidence matrix associated with the system of equations is square, which is a necessary but not sufficient condition for the equation system to be structurally non-singular.

[2] In this thesis, we do not make any statement on how to handle the problem of separate compilation. We will only argue for that our approach can enable static checking of separately compiled models.

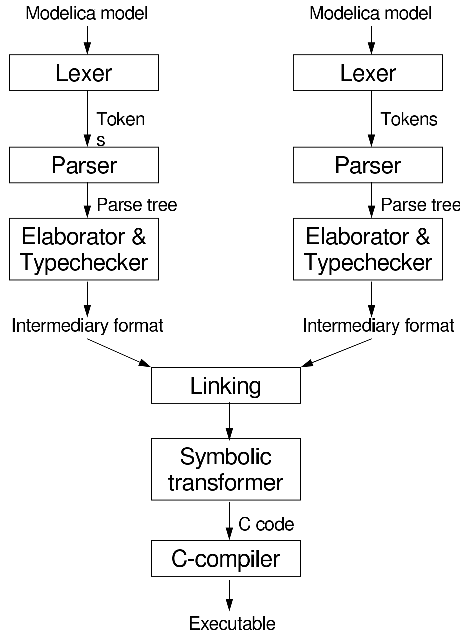


Figure 6.1: *Separate compilation in Modelica.*

To summarize, there are two deficiencies with the current practice in the Modelica compilers before Modelica 3.0[3] that we would like to stress.

1. Complete elaboration of all elements in a model is required to determine if the model is under- or over-constrained.

2. If the model turns out to be under- or over-constrained, it is very hard to find the bug because the error is detected at the level of flat system of equations rather than at a component/model level.

6.2 Featherweight Modelica

Modelica is a large and complex language that includes many concepts such as discrete simulation, algorithm sections, and functions, which are not central for our purpose. Consequently, we have designed and extracted a subset of the Modelica language, which models important aspects of the continuous and object-oriented parts of the language. We call this language Featherweight Modelica (FM). This section will informally present the language.

[3]The approach presented in this chapter was published in 2006, i.e., before Modelica 3.0 was released.

6.2.1 Syntax and Semantics

A model designed in FM can express continuous behavior by using Differential Algebraic Equations (DAEs). Reuse is achieved by the `extends` and `redeclare` constructs.

In Figure 6.2 the syntax grammar of FM is listed using a variant of extended Backus-Naur Form (EBNF). Alternatives are separated using the '|' symbol, optional arguments are given using square brackets ($[\cdots]$) and the curly brackets ($\{\cdots\}$) denote that the enclosed elements can be repeated zero or more times. Terminals are highlighted in bold-face.

The non-terminal $root$ gives the starting point of a model definition. The metavariable M ranges over names of models and m over names of instances of models; C ranges over names of connectors and c over names of instances of connectors; R ranges over names of records and r over names of instances of records; x ranges over variable names of type `Real`. Note that numeric subscripts are used to differentiate between meta-variables. All bold strings are keywords in the language except for `Real`, which is the built in type for $\mathbb{R}$.

The foundation of the language is the *class* concept, where `model`, `connector`, and `record` are special forms of classes. By observing the grammar, we can see that only models are allowed to have connections or to contain elements that can be redeclared or modified. Connectors are the only classes whose instances can be part of a `connect`-equation, while `Real` types and `record` instances can be part of equations. Note that this can be seen in the grammar by considering the meta-variables.

There are two kinds of prefixes: *access* and *modifiability*. Access prefixes state if an element in a model can be defined to be `public` or `protected`. The latter is only visible outside the model by a model extending from the class. The second prefix category is modifiability, defining how an element can be modified. Declaring an element replaceable makes it possible for a user to redeclare the element. Setting the prefix of an element to `final` means that the element can neither be modified nor redeclared. Only models can be redeclared and only `Reals` can be modified in FM.

6.2.2 Type-Equivalence and Subtyping

Modelica is using a so called *structural type system* [124], where the type of a class is determined by the structure of its components. However, other object-oriented languages, such as Java, are using primarily a *nominal type system*, where the name of a declared class identifies its type.

The Modelica language specification [102] is informally describing the semantics and type system of the language. From the specification, the following definition of *type equivalence* can be extracted:

Definition 6.2.1 (Type Equivalence). Two types T and U are equivalent if T and U denote the same built-in type, or T and U are types of classes, T and U contain the same public declaration elements (according to their names), and the elements' types in T are equivalent to the corresponding element types in U.

Note that a *class* `C` is not the same as the *type* of class `C` because the type only represents the *interface* of the class and not the private *implementation* or *semantic* part, such as equations.

$$
\begin{array}{rcl}
\mathit{root} & ::= & \{\mathit{model} \mid \mathit{connector} \mid \mathit{record}\} \\
\mathit{model} & ::= & \texttt{model}\ M_1 \\
& & \{\texttt{extends}\ M_2\ [\mathit{modification}]\ \texttt{;}\} \\
& & \{[\mathit{access}]\ [\mathit{modifiability}] \\
& & (M_3\ m\ [\mathit{modification}] \mid \\
& & C\ c \mid R\ r \mid \texttt{Real}\ x\ [\texttt{=}\ \mathit{lnum}])\ \texttt{;}\} \\
& & [\texttt{equation}\ \{\mathit{equation}\texttt{;}\}] \\
& & \texttt{end}\ M_1\ \texttt{;} \\
\mathit{connector} & ::= & \texttt{connector}\ C_1\ \{\texttt{extends}\ C_2\ \texttt{;}\} \\
& & \{[\texttt{flow}]\ \texttt{Real}\ x\ \texttt{;}\} \\
& & \texttt{end}\ C_1\ \texttt{;} \\
\mathit{record} & ::= & \texttt{record}\ R_1\ \{\texttt{extends}\ R_2\ \texttt{;}\} \\
& & \{(R_3\ r \mid \texttt{Real}\ x)\ \texttt{;}\} \\
& & \texttt{end}\ R_1\ \texttt{;} \\
\mathit{modification} & ::= & \texttt{(}\mathit{modification}'\ \{\texttt{,}\ \mathit{modification}'\}\texttt{)} \\
\mathit{modification}' & ::= & \texttt{redeclare}\ M\ m\ [\mathit{modification}] \\
& \mid & x\ \texttt{=}\ \mathit{lnum} \\
\mathit{access} & ::= & \texttt{public} \mid \texttt{protected} \\
\mathit{modifiability} & ::= & \texttt{replaceable} \mid \texttt{modifiable} \\
& \mid & \texttt{final} \\
\mathit{equation} & ::= & \texttt{connect(}c_1\texttt{,}c_2\texttt{)} \mid e_1\ \texttt{=}\ e_2 \\
e & ::= & e_1\ \texttt{+}\ e_2 \mid e_1\ \texttt{-}\ e_2 \mid e_1\ \texttt{*}\ e_2 \mid e_1\ \texttt{/}\ e_2 \\
& \mid & \texttt{-}e \mid \texttt{(}\ e\ \texttt{)} \mid \mathit{lnum} \mid \texttt{der(}x\texttt{)} \mid x \mid r \\
& \mid & \texttt{time} \mid \texttt{sin(}e\texttt{)}
\end{array}
$$

Figure 6.2: *Syntax of Featherweight Modelica.*

Besides type equivalence, the Modelica language defines subtyping relationships between types of classes.

Definition 6.2.2 (Subtyping). For any types S and C, S is a supertype of C and C is a subtype of S if they are equivalent or if: every public declaration element of S also exists in C (according to their names) and those element types in S are supertypes of the corresponding element types in C.

In the following text, we will use the notation of `C <: S`, meaning that the type of class `C` is a subtype of the class `S`'s type.

Now, consider the three models given in Figure 6.3. According to Definition 6.2.2, we can see that `B <: A` because the public elements `p` and `c` that exist in `A` also exist in `B`. We can see that `C` extends `A`, i.e., `C` inherits all components and equations from `A`. Furthermore, `C` defines an element `q`, which makes `C <: A`. In addition because both `B`

```
model A
  Real p;
  Real c;
equation
  c = 2;
  der(p) = -c*p;
end A;
```

```
model B
  Real p;
  Real c;
  Real q;
equation
  c = 2;
  der(p) = -c*p;
end B;
```

```
model C
  extends A;
  Real q;
equation
  q = p*p;
end C;
```

Figure 6.3: *Three different Modelica models.*

and C hold the same public elements, it can be concluded from Definition 6.2.1 that B and C are type equivalent.

Subtyping is a fundamental concept in many programming languages. Generally, it means that if a type S has all the properties of another type T, then S can be safely used in all contexts where type T is expected. This view of subtyping is often called *the principle of safe substitution* [124]. Now the question arise if this is true for the type system and examples described above. The main question is what we mean by *safe substitution* in the context of equation-based object-oriented languages. If we count the number of variables and equations in each of the models in Figure 6.3, we can see that model A has 2 variables and 2 equations, model B has 3 variables and 2 equations and finally model C has 3 variables and 3 equations. In the current type system of Modelica, both B and C are said to be safe replacements of A. However, in this case we know that replacing A with C gives us a potentially solvable system with 3 variables and 3 equations, but replacing A with B results in a under-constrained system with 3 variables and 2 equations, which will not give a unique solution. Can we after these observations still regard B as a safe replacement of A? We think not, and will in the next subsections propose a solution.

6.3 The Approach of Structural Constraint Delta

In this section, we will present an approach that addresses the problem of determining under- and over-constrained components without performing complete elaboration. We start by giving a definition:

Definition 6.3.1 (Structural Constraint Delta, C_Δ). Given an arbitrary class C, containing components, equations, and connections, the type of C has a defined integer attribute called structural constraint delta, C_Δ. This attribute states, for C and all its subcomponents, the integer difference between the total number of defined equations and variables.

The term *structural* indicates that the equations and variables are counted as they are declared in the model. For example, two linearly dependent equations in an equation system will still be counted as two separate equations. Hence, $C_\Delta = 0$ for a system of equations does not guarantee a unique solution, it will only indicate that a single solution might exist. If $C_\Delta < 0$, we have an under-constrained problem with more unknowns

than equations, which might give an infinite number of solutions. If $C_\Delta > 0$, we have an over-constrained system of equations, which most likely will not give a unique solution. However, because the algorithm for computing C_Δ does not check if equations are linearly independent or not, a system with $C_\Delta > 0$ may be solvable. To be able to guarantee that a system of equations has a unique solution, complete knowledge of the entire system of equation must be available. Because this is obviously not possible when inspecting components separately, the value of C_Δ only provides a good indication whether a system of equations has a unique solution or not.

For example, if C_Δ is to be calculated for the types of the models given in Figure 6.3, the difference between the number of equations and variables in the model gives the value of C_Δ. In this case, $C_\Delta = 0$ for `A` and `C`, but $C_\Delta = -1$ for `B`. Because our models so far only contain variables and equations, calculating C_Δ is straightforward. However, if a model contains hundreds of subcomponents, using connections, connectors, and records, the resulting flattened system might consist of thousands of equations. To be able to formulate algorithms for calculating C_Δ, we need another definition:

Definition 6.3.2 (Constraint Delta Effect, E_Δ). Let C be an arbitrary class containing two elements `c1` and `c2` that are instances of classes C1 and C2, which contain only elements and no equations or connections. Given an equation or connection E located in C representing a relation between c1 and c2, the constraint delta effect E_Δ is a type attribute of both C1 and C2, which states the effect E has when computing C_Δ of C.

Note that C_Δ is not the same as E_Δ. Simply stated, we say that E_Δ of two elements represents the change of the current model's C_Δ when an equation or connection is introduced between the two elements. For example, if we in model `B` in Figure 6.3 introduce a new equation `q = 2 * p`, this equation will have the effect of changing model `B`'s C_Δ from -1 to 0. Therefore, involved variables `q` and `p`, are said to have $E_\Delta = 1$ (or to be precise; the attributes to the types of the elements). However, we will soon see that elements do not always have $E_\Delta = 1$.

6.3.1 Algorithms for Computing C_Δ and E_Δ

In this section, we present algorithms for calculating C_Δ and E_Δ. Even if the algorithms for calculating the type attributes C_Δ and E_Δ could be stated by using a formal type system, we have chosen to illustrate the algorithm more informally using pseudo-code algorithms. The main reasons for this are that the Modelica language itself has currently no formal semantics or type system and the target audience of this chapter is not only computer scientists, but also engineers from the modeling and simulation community.

It is important to stress that C_Δ and E_Δ are defined as attributes to the *types* of the classes, and not for the classes themselves. This implies that when calculating the value for a specific class `C`, we do not need to recursively calculate C_Δ and E_Δ for each subelement because they are already defined by the type of the elements[4]. The process of calculating C_Δ and E_Δ is a form of *type inference*, i.e., the type attributes are inferred from equations given in the class and types of the elements in the class.

[4] In FM, we have made the assumption that variables are always bound to a value without circular dependencies. Unfortunately cannot this be guaranteed in full Modelica.

Algorithm 1: Compute C_Δ of a class

Input: An arbitrary $Class$
Output: C_Δ of the class

```
1  C_Δ ← 0
2  switch Class do
3      case model
4          foreach e ∈ getElements(Class) do
5              C_Δ ← C_Δ+ getDelta(e)
6              if hasDefaultValue(e) then
7                  C_Δ ← C_Δ+ 1
8              foreach m ∈ getModifiedElements(e) do
9                  if not hasDefaultValue(m) then
10                     C_Δ ← C_Δ+ 1
11         foreach e ∈ getEquations(Class) do
12             C_Δ ← C_Δ+ getEffect(e)
13         foreach c ∈ getConnectors(Class) do
14             P_outside ← FALSE
15             P_inherited ← FALSE
16             if not isVisited(c) then
17                 traverseConnectorGraph(c)
18                 if P_outside then
19                     C_Δ ← C_Δ+ getOutsideAdjustment(c)
20         foreach b ∈ getBaseClasses(Class) do
21             foreach m ∈ getModifiedElements(b) do
22                 if not hasDefaultValue(m) then
23                     C_Δ ← C_Δ+ 1
24             C_Δ ← C_Δ+getDelta(b)
25     case record
26         foreach e ∈ getElements(Class) do
27             C_Δ ← C_Δ+ getDelta(e)
28         foreach b ∈ getBaseClasses(Class) do
29             C_Δ ← C_Δ+ getDelta(b)
30     case connector
31         foreach e ∈ getElements(Class) do
32             if not hasFlowPrefix(e) then
33                 C_Δ ← C_Δ+ getDelta(e)
34             foreach b ∈ getBaseClasses(Class) do
35                 C_Δ ← C_Δ+ getDelta(b)
36     case variable
37         C_Δ ← −1
38 end
```

The algorithm for computing C_Δ is given in Algorithm 1. This algorithm uses a help function defined in Algorithm 2. The algorithm for computing E_Δ is listed in Algorithm 3. Note that the indentation of the algorithms is significant and delimits blocks for the `foreach`, `if`, and `switch` statements.

Algorithm 2: traverseConnectorGraph(c_1)

```
   Input: Connector c1 from which graph traversal starts
   Output: Global variables P_outside, P_inherited, and C_Δ
 1 if ((isOutside(c1) and isInherited(c1)) or ((isOutside(c1)
 2     or isInherited(c1)) and (P_outside or P_inherited)) then typeCheckingFailed()
 3 else
 4     markAsVisited(c1)
 5     P_outside ← P_outside or isOutside(c1)
 6     P_inherited ← P_inherited or isInherited(c1)
 7     foreach c2 ∈ getAdjacencyConnectors(c1) do
 8         if not isVisited(c2) then
 9             C_Δ ← C_Δ + getEffect(getTypeOf(c2))
10             traverseConnectorGraph(c2)
```

Algorithm 3: Compute E_Δ of a class

```
   Input: An arbitrary Class
   Output: E_Δ of the class
 1 E_Δ ← 0
 2 switch Class do
 3     case record
 4         foreach e ∈ getElements(Class) do
 5             E_Δ ← E_Δ + getEffect(e)
 6         foreach b ∈ getBaseClasses(Class) do
 7             E_Δ ← E_Δ + getEffect(b)
 8     case connector
 9         foreach e ∈ getElements(Class) do
10             if hasFlowPrefix(e) then
11                 E_Δ ← E_Δ − getEffect(e)
12             else E_Δ ← E_Δ + getEffect(e)
13         foreach b ∈ getBaseClasses(Class) do
14             E_Δ ← E_Δ + getEffect(b)
15     case variable
16         E_Δ ← 1
17 end
```

To make the algorithms more easy to follow, the following help functions are defined:

- **getAdjacencyConnectors** (c) - the set of connectors that are directly connected to c by connect-equations declared in the local class.
- **getBaseClasses** (C) - the set of types for the base classes to C.
- **getConnectors**(C) - the set of accessible connectors that are used by connections in class C. All connectors are initially marked as unvisited.
- **getDelta**(t) - attribute C_Δ part of type t.
- **getElements**(C) - the set of types for elements part of class C.
- **getEquations**(C) - the set of equations part of the local class C, excluding connect-equations

and equations from base classes. Each element in the set represents the type of the expressions declared equal by the equation.

- **getEffect**(t) - the attribute $E_\triangle$ part of type t.
- **getModifiedElements**(e) - the set of elements' types in e, which is modified by modification equations.
- **getOutsideAdjustment**(c) - an integer value representing adjustments to be made if connector c is part of a connector set that is connected to an outside connector. The integer value is equal to the positive number of flow variables inside connector c.
- **getTypeOf**(c) - the type of connector c.
- **hasDefaultValue**(e) - TRUE if element type e has a defined default value.
- **hasFlowPrefix**(e) - TRUE if element e is prefixed with keyword `flow`.
- **isInherited**(c) - TRUE if connector c is inherited from a base class.
- **isVisited**(c) - TRUE if connector c is marked as visited.
- **isOutside**(c) - TRUE if connector c is seen as an outside connector in the local class.
- **markAsVisited**(c) - mark connector c as visited.
- **typeCheckingFailed**() - terminates the type because two outside or inherited connectors are connected, or a connected connector is both outside and inherited.

Computing $C_\triangle$ - Equations, Inheritance, and Modification

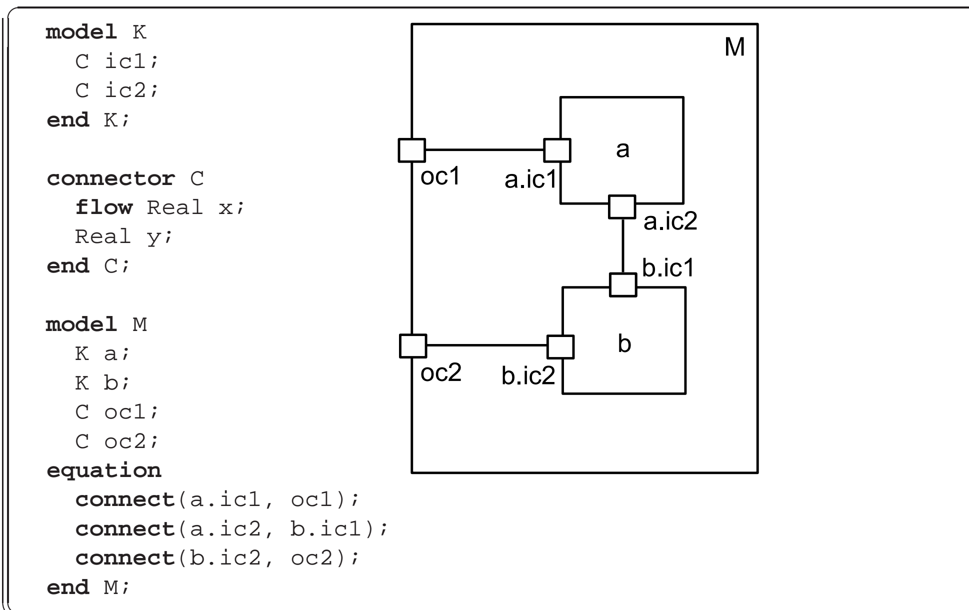
We start by illustrating the algorithms using trivial examples, where the models only contain equations, records, and variables. Consider the following FM listing:

```
record R            C△=-2  E△=2
  Real p;           C△=-1  E△=1
  Real q;           C△=-1  E△=1
end R;

model A             C△=-3
  R r1;             C△=-2  E△=2
  R r2;             C△=-2  E△=2
  Real p;           C△=-1  E△=1
equation
  r1 = r2;
end A;

model B             C△=0
  Real y=10;        C△=0  E△=1
end B;

model M             C△=-1
  extends A(p=1);   C△=-2
  B b1(y=20);       C△=0
  B b2;             C△=0
equation
  b1.y = p;
end M;
```

Model `M` extends from model `A`, which implies that all equations and elements in `A` will be merged into `M`. Model `A` contains two instances of record `R`. If each of these models were to be compiled separately, we would need to calculate C_Δ for each of the models without any knowledge of the internal semantics of the subcomponents, i.e., the equations. Calculated C_Δ and E_Δ for every class and element are given to the right in the listing.

Consider Algorithm 1, which takes an arbitrary class as input and calculates the C_Δ value for this class. First, we can see that calculating C_Δ of a record simply adds the C_Δ value for each element (rows 26-27), which in the case of record `R` gives $C_\Delta = -2$ because `R` holds 2 variables. In Algorithm 3, we can see that calculating the effect of `R` gives $E_\Delta = 2$. But what does this mean? Recall that E_Δ, given in Definition 6.3.2, states the effect on C_Δ when connecting two elements. In model `A`, an equation `r1 = r2` is given, which uses record `R`. This equation will after elaboration generate two equations, namely `r1.p = r2.p` and `r1.q = r2.q`, which is why E_Δ for `R` is 2. The rest of the procedure for computing C_Δ of model `A` should be pretty straightforward by following Algorithm 1. Note that only C_Δ and not E_Δ is given for models because models are not allowed to be interconnected.

The more interesting aspects of calculating C_Δ in this example are shown in model `M`. First of all, we can see that model `M` extends from `A`, which results in that C_Δ of `A` is added to C_Δ of `M` (see rows 20-24 in Algorithm 1). Because variable `p` is modified with `p=1`, we see that C_Δ is increased by E_Δ of the type of `p`, i.e., `Real`. Hence, the C_Δ contribution from base class `A` is -2. The C_Δ value for model `B` is 0. When instantiated to element `b1` in model `M`, its element `y` is modified with `y=20`. However, this modification does not




```
model K
  C ic1;
  C ic2;
end K;

connector C
  flow Real x;
  Real y;
end C;

model M
  K a;
  K b;
  C oc1;
  C oc2;
equation
  connect(a.ic1, oc1);
  connect(a.ic2, b.ic1);
  connect(b.ic2, oc2);
end M;
```

Figure 6.4: *Model `M` with inside connectors (e.g. `a.ic1` and `b.ic2`) and outside connectors (`oc1` and `oc2`).*

effect C_Δ because `y` already has a default value (see rows 8-10 in Algorithm 1). Finally, we can see that the total calculation of `M` will result in a C_Δ value of -1.

Computing C_Δ - Connectors, Connections, and Flow Variables

Consider the source code listing and graphical representation given in Figure 6.4. Model `M` contains components `a` and `b`, which are instances of model `K`. Each model consist of several connector instance all instances of a connector class `C`.

The semantics of the Modelica language distinguish between *outside connectors* and *inside connectors*, where the former are connector instances denoting the border of a model, e.g., `oc1` and `oc2`, and the latter represent connectors available in local components, e.g., `a.ic1`, `a.ic2`, `b.ic1`, and `b.ic2`. Note that a connector instance can be seen as both an outside and an inside connector, depending which model is being processed. In this example we are looking at model `M`.

Calculating C_Δ of connector `C` can be achieved by using rows 30-35 in Algorithm 1. On row 32, we can see that C_Δ is only added if the variable has not got a flow prefix. The reason for this is that an unconnected flow variable has always a defined default equation, setting its value to 0. Hence, introducing a flow variable gives one extra variable and one equation, i.e., $C_\Delta = 0$. Further inspection of the algorithm yields $C_\Delta = -2$ for model `K`.

Calculating C_Δ of `M` is more complicated. On row 13 in Algorithm 1 it is stated that we iterate over all involved connectors, in this case `a.ic1`, `a.ic2`, `b.ic1`, `b.ic2`, `oc1`, and `oc2`. Variable $P_{outside}$ becomes TRUE if the algorithm has passed an outside connector, and $P_{inherited}$ becomes TRUE if it has passed an inherited element. The latter case will not be illustrated in this example. The first thing to notice is that the connector graph is traversed by using the recursive function traverseConnectorGraph(), listed in Algorithm 2. The algorithm performs a *depth-first search* visiting each connector (vertex) only once, by marking it as visited. Note that function traverseConnectorGraph() has side effects and updates the variables $P_{outside}$, $P_{inherited}$, and C_Δ. Each connect-equation (edge) in the graph contributes to the C_Δ of the class being computed, by adding E_Δ of a connector in the connection (see row 9 in Algorithm 2). Because all connectors traversed in one iteration of the foreach loop are connected (row 13-19 in Algorithm 1), all types of the connectors hold the same value of E_Δ.

By using Algorithm 3, rows 9-12, we can see that $E_\Delta = 0$ for connector `C`. Consequently, all the connections in model `M` will not change the value of C_Δ. Why is this the case? We know that connecting non-flow variables will always result in an extra equation, i.e., for non-flow variables, E_Δ must be 1. However, when connecting two flow variables, one equation is added, but two default equations are removed. For example in `connect(a.ic2, b.ic1)`, the two default equations `a.ic2.x=0` and `b.ic1.x=0` are removed and replaced with the sum-to-zero equation:

```
a.ic2.x + b.ic1.x = 0
```

Hence, the effect of connecting two flow variables is $E_\Delta = -1$.

There are several aspects covered by the algorithms that we have not covered in detail. The following items briefly describe some of these issues:

- If cycles appear in the connector graph, there exists a redundant connect-equation

which does not contribute to the value of $C_\triangle$. For example, if connections `connect(oc1,b.ic1)` and `connect(a.ic1,a.ic2)` would be introduced in `M`, one connection would be redundant. This issue is handled by making sure that connectors are only visited once (see rows 7-10 in Algorithm 2.)

- Connecting an inside connector to an outside connector does not give the same effect on $C_\triangle$ as connecting inside to inside. For example, when connecting `oc1` to a local connector inside `M`, the default variable `oc1.x=0` will not be removed. This default equation will only be removed when `oc1` is connected outside model `M`, i.e., when another model is using `M` as a component. This issue is managed on rows 18-19 in Algorithm 1.
- The algorithm does not allow direct or indirect connections between outside connectors. For example, a connection `connect(oc1,b.ic2)` would generate a type checking error (see row 1-2 in Algorithm 2). The same semantics hold for connections between connectors inherited from base classes. We use this conservative approach because without it, the type of a class must be extended with information regarding the connectors that are connected.

6.3.2 Extending the Type System with $C_\triangle$

The previous sections describe how we can calculate $C_\triangle$ and $E_\triangle$ of classes, resulting in value attributes for types in the language. However, this is of no use if we do not apply this new information to the type system. A new extended version of the Featherweight Modelica language, denoted $\text{FM}_\triangle$, is defined by extending Definition 6.2.1 and Definition 6.2.2 for type-equivalence and subtyping with the following definitions:

Definition 6.3.3 (Type-equivalence and $C_\triangle$). For any types T and U to be type-equivalent, Definition 6.2.1 must hold and the $C_\triangle$-value of T and U must be equal.

Definition 6.3.4 (Subtyping and $C_\triangle$). For any types S and C, S is a supertype of C and C is a subtype of S if Definition 6.2.2 holds and the $C_\triangle$-value of S is equal to that of C.

Hence, the extended language $\text{FM}_\triangle$ guarantees that the difference between declared variables and equations does not change when using the rule of subsumption. If we recall the models listed in Figure 6.3, we can now see that model `C` is a subtype of model `A`, but model `B` is not.

6.4 Prototype Implementation

To validate and verify our algorithms, a prototype Featherweight Modelica Compiler (`fmc`) was implemented consisting of a type-checker for $\text{FM}_\triangle$, where $C_\triangle$ and $E_\triangle$ are automatically inferred and represented as attributes to the types. The prototype compiler was implemented as a batch-program, which takes an $\text{FM}_\triangle$ `.mo`-file (containing $\text{FM}_\triangle$ models) as input and returning to standard output the pretty-printed type of the last model defined in the `.mo`-file.

To validate the correctness of our solution, the following procedure has been used:

1. Create relevant models in FM_Δ.
2. Run the prototype compiler for FM_Δ on the models. The output is the listed type of the model including C_Δ information.
3. Elaborate the model and manually inspect the flat Modelica code generated by the compilers Dymola version 6 [45] and OpenModelica version 1.4.1 [52].

The above approach gives us confidence that the algorithm is correct with respect to the Modelica semantics, but it does not give any guarantees. The best option would of course have been to be able to prove the correctness of the algorithm. However, we face two problems with this. Firstly, the algorithm itself is fairly complicated, which is the effect of complications of the Modelica connect semantics using connection graphs. The question arises naturally if the connection semantics needs to be that complicated. In Section 11.2 we will propose an alternative semantics for describing connections compared to Modelica, which we argue has a simpler semantics. Secondly, to be able to prove correctness compared to Modelicas elaboration semantics, we need a formalization of that elaboration semantics. Because this is not available, we have instead justified the correctness by implementation and testing.

We will now analyze, by using a simple circuit example, how the concept of structural constraint delta attacks the problems of constraint checking with separately compiled model components, and error detection and debugging. In the examples, `fmc` and Dymola version 6 are used when testing the models.

6.4.1 Constraint Checking of Separately Compiled Components

Consider the following listing, stating the model `Resistor`, a connector `Pin` and a base class `TwoPin`:

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

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

model Resistor
  extends TwoPin;
  Real R = 100;
equation
 R*i = v;
end Resistor;
```

When using `fmc`, each of these models is separately type checked. For example, when typechecking model `Resistor`, model `TwoPin` and connector `Pin` are not elaborated. Instead, only the types of `TwoPin` and `Pin` are used. This information is available after these classes are compiled.

Below the output generated by `fmc` is listed, with some pretty printing added for readability:

```
model classtype C∆=0
  public final connector objtype C∆=-1 E∆=0
```

```
    nonflow Real objtype v;
    flow Real objtype i;
  end p;
  public final connector objtype C△=-1 E△=0
    nonflow Real objtype v;
    flow Real objtype i;
  end n;
  public modifiable Real objtype v;
  public modifiable Real objtype i;
  public modifiable Real objtype* R;
end
```

The lines above represent the type of model `Resistor`. Note the difference made between *class type* (the type of a class that can be instantiated), and a *objtype* (the type of an object that has been instantiated by a class). The type's of elements `p` and `n` have $C_\triangle = -1$ and $E_\triangle = 0$. The latter indicates that when the `Resistor` model is used by connecting `p` or `n`, $C_\triangle$ will not change. Finally, we can see that that $C_\triangle = 0$ for the whole type of `Resistor`.

Now, if the following code is added to our `.mo`-file, we have a complete model named `Circuit` that we can simulate.

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

```
model VsourceAC
  extends TwoPin;
  Real VA = 220;
  Real f = 50;
  Real PI = 3.1416;
equation
  v = VA*sin(2*PI*f*time);
end VsourceAC;
```

```
model Inductor
  Pin p;
  Pin n;
  Real v;
  Real i;
  Real L = 1;
equation
  L*der(i) = v;
end Inductor;
```

```
model Circuit
protected
  replaceable Resistor R1(R=10);
  replaceable Inductor L(L=0.1);
  VsourceAC AC;
  Ground G;
equation
  connect(AC.p, R1.p);
  connect(R1.n, L.p);
  connect(L.n, AC.n);
  connect(AC.n, G.p);
end Circuit;
```

Trying to simulate the above model `Circuit` in the commercial Modelica environment Dymola, the error feedback states that it is not possible to simulate it because there are 22 equations and 25 variables in the flattened equation system.

Executing the model in `fmc`, we get the response that model `circuit` has $C_\triangle = -3$, which corresponds to the message Dymola reported. Note that Dymola had to elaborate all the models to a flattened system of equation to get to this result. `fmc` on the other hand could use the separately type checked components and just use the types of these

components to get the same result. Hence, this example illustrates how our approach can be used to enable separate compilation of components.

6.4.2 Error Detection and Debugging

Now the following question arise: How can we know where the problem is located? The user needs to either analyze the model code or to inspect the flat system of equations. In both cases, this problem seems hard to manage.

If we run this model in `fmc`, we get the following type information for model `Circuit` (for readability, parts of the type are replaced by a dotted line):

```
model classtype C△=-3
  protected replaceable model objtype C△=0
     ...
  end R1;
  protected replaceable model objtype C△=-3
     ...
  end L;
  protected modifiable model objtype C△=0
     ...
  end AC;
  protected modifiable model objtype C△=0
     ...
  end G;
end
```

Analyzing the type information, it indicates that it is component `L`, which is an instance of `Inductor` that probably causes the under-constrained system. After a closer look, we notice that `Inductor` is not extending from `TwoPin`, as it should. After replacing the old `Inductor` model with

```
model Inductor
  extends TwoPin;
  Real L = 1;
equation
  L*der(i) = v;
end Inductor;
```

it is possible to simulate the model.

Now, let us assume that we want to build a larger model having model `Circuit` as a subcomponent. However, this time we do not want to use a `Resistor` in `Circuit`. Instead, the goal is to redeclare `R` with a temperature dependent resistor called `TempResistor`. Consider the following models:

```
model TempResistor
  extends TwoPin;
  Real R;        // Resistance at.  reference temp.
  Real RT=0;     // Temp.  dependent resistance
  Real Tref=20;  // Reference temperature
  Real Temp;     // Actual temperature
equation
```

```
  v = i * (R + RT * (Temp-Tref));
end TempResistor;

model Circuit2
  extends Circuit(redeclare TempResistor R1(R=35));
end Circuit2;
```

Trying to simulate this model in Dymola results in a flattened model with 28 variables and 27 equations, which cannot be simulated. By elaborating all components and analyzing the system of equations, Dymola hints that R1 is structurally singular.

However, using `fmc`, this model does not even pass the type checker. The compiler reports that C_Δ for the original type is 0 (`Resistor`), but the redeclaring model's type is -1 (`TempResistor`). Hence, the subtyping rule is not legal and the redeclaration is incorrect. The following listing shows a correct redeclaration, where the temperature parameter `Temp` has been assigned a value.

```
model Circuit3
  extends Circuit
    (redeclare TempResistor R1(R=35, Temp=20));
end Circuit3;
```

Consequently, our approach finds the incorrect model at an early stage during type checking. Furthermore, because the type checking was performed on precompiled models, there is no need for elaborating the model's subcomponents. Hence, this approach is not only useful for separate compilation, but also for users when locating errors in models.

6.5 Chapter Summary and Conclusions

We have presented the concept of structural constraint delta (C_Δ) for equation-based object-oriented modeling languages. Algorithms for computing C_Δ were given, and it was shown how C_Δ is used to determine if a model is under- or over-constrained *without* having to elaborate a model's components.

We have also illustrated how the concept of C_Δ allows the user to detect and pinpoint some model errors. The concept has been implemented for a subset of the Modelica language and successfully tested on several models.

Part II

The Modeling Kernel Language

7

Introduction to Functional Programming in MKL

FROM our study of the Modelica language in Part I, we have seen that the language is large and complex, providing many possibilities for advanced modeling. However, we have also concluded that it is hard to formalize because it is large and complex.

In this second part of the thesis, we present a new language called the Modeling Kernel Language (MKL). Instead of taking a top-down approach of trying to formalize a large language such as Modelica, this research is following a bottom-up approach, where we formally define a small language. The objective and hope is then that the small kernel language can be extended with both new modeling constructs, as well as functions for manipulating and making use of the mathematical models. The approach that we are exploring is that these extensions are not added to the core of the language itself, but are instead added as library functions written in MKL. This approach concerns the research questions stated in Section 1.4.3 regarding the problem of designing an expressive and extensible EOO language that is formally defined.

The MKL language is fundamentally a statically typed *functional language*. The rationale for designing a functional language is because we would like to base it on a well known and proven theory, in this case the lambda calculus [12].

In this introductionary chapter to Part II, we first explain the fundamentals of functional programming by introducing the syntax and the standard functional parts of MKL, i.e., language constructs that can be found in languages such as Haskell [134] or Standard ML [98] (Section 7.1). This is followed by a brief introduction to the lambda calculus (Section 7.2).

Part II consists of the following chapters:

- **Chapter 8 - Modeling in MKL**. In this chapter we first give a brief description of basic modeling in MKL. This is followed by introducing the concept of higher-order acausal modeling (HOAM). By providing a simple model in the electrical and mechanical domains, we show the expressive power of the concept. This chapter also informally introduces the modeling capabilities of the MKL.

- **Chapter 9 - Intensional Analysis of Models**. In this chapter, we demonstrate how MKL can also be used for intensional analysis of the models, i.e., to inspect and analyze the equation system of the model and to synthesize output for different purposes.
- **Chapter 10 - Semantics of MKL**. In this chapter, we formally define the syntax and semantics of the core of MKL, which is an extension of the simply typed lambda calculus. The semantics forms the foundation of both the modeling capabilities and the ability of intensional analysis of models. Both a small-step dynamic semantics and a static type system are formally defined. We also prove type safety for the core language.
- **Chapter 11 - Elaboration Semantics**. We describe and give a formal definition of the elaboration semantics used for elaborating models down to equation systems. The problem of extracting information from models is discussed and a solution is proposed.
- **Chapter 12 - Implementation, Verification, and Evaluation**. In this final chapter we first give an overview of the prototype implementation of MKL. This is followed by two examples of the use of models - one for direct simulation and one for exporting to Modelica code. We explain how we verify the solution and we discuss and evaluate our approach with regards to safety, expressiveness, extensibility, and performance aspects.

7.1 Functional Programming in MKL

There are many different opinions of what functional programming actually means. To give an intuition, we quote Hutton [69, p. 2]:

> "functional programming can be viewed as a *style* of programming in which the basic method of computation is the application of functions to arguments. In turn, a functional programming language is one that *supports* and *encourages* the functional style"

The modeling kernel language (MKL) is a functional programming language, specially designed for providing EOO language functionality within libraries. The current version that we present in this thesis should be seen as a research language for exploring this approach.

MKL has a syntax style where expressions are similar to OCaml [74], and types have similarities to Haskell [134], and, to some extent, Clean [125]. The subset that is presented in this chapter only consists of standard constructs, available in most functional programming languages. The extensions are presented in the following chapters.

The functional programming style encourages declarative programming, where function applications do not have *side effects*, i.e., a function applied to the same argument should always return the same value regardless of its context. Languages where function applications do not have side effects are often called *pure* functional languages (e.g.,

Haskell or Clean).[1] In other functional languages (e.g., OCaml or Standard ML [98]) pure functional programming is encouraged, but the language also supports effectful constructs, i.e., constructs with side effects, e.g., reference updates and destructive arrays. MKL is within the latter category, where handling of unknowns (explained in coming chapters) and destructive arrays are effectful constructs that are part of the language.

A functional languages have a defined evaluation strategy. The language can be *strict (eager)* meaning that arguments to functions are evaluated before they are passed to the function at function call. Examples of functional programming languages within this category are OCaml, Standard ML, and Common LISP [137]. A FP language can also use an evaluation strategy when arguments are supplied to a function without first evaluating the arguments. In its simplest form, this is called *call-by-name*, but to avoid recomputation of expressions a more efficient approach is often referred to as *call-by-need*. This approach is used by languages such as Haskell and Clean. MKL is within the former class, i.e. a strict functional language.

A program in MKL is an *expression*. For example `4 + 5 * 3` is an expression for adding and multiplying integers. When executing a program, the expression is *evaluated* to a *value*. We write $e \longrightarrow^* v$ meaning that e is evaluated to the value v in zero or more steps, and $e \longrightarrow e'$ meaning that e is reduced to e' in one step. For example `4 + 5 * 3` $\longrightarrow^*$ `19` and `4 + 5 * 3` $\longrightarrow$ `4 + 15`.

Each expression e has a (or belongs to a) *type* T, written $e : T$. That is, the type predicts the kind of value that the expression reduces to. For example `4 + 5 * 3 : Int`, and `19 : Int` have both type `Int` because the type of an expression is preserved during evaluation. MKL is *statically typed*[2], meaning that all expressions in a program can be given a type statically, before evaluation.

In MKL, there are four basic types: `Int`, `Real`, `Bool`, and `String`. Overloading is not allowed and therefore different operators are used for different types. Similar to OCaml, operators for real types are given a dot suffix. For example, expression `12.10 *. 10. -. 50.` is evaluated to `52.10`. Boolean literals are **`true`** and **`false`**, with infix logical operators `&&` for *and*, `||` for *or*, and `!` for negation. Two strings can be concatenated using infix operator `++`, e.g., `"My " ++ "string\n"` is evaluated to `"My string\n"`. Escape sequences, such as new line `\n` can be given inside a string. For a full list of available functions and operators that are built-in to MKL, see Appendix D.1.

7.1.1 Higher-Order Functions and Currying

In functional languages, the most fundamental language construct is a *function*. Functions correspond to partial mathematical functions, i.e., a function $f : A \to B$ gives a mapping from (a subset of) the domain A to the codomain B. *Anonymous functions* (also called lambda abstractions) are functions without names. Such functions are also expressions. An anonymous function can be expressed as

$$\texttt{\textbf{fun}}\ x\texttt{:}T\ \texttt{->}\ e$$

[1] Haskell is using monads and Clean uniqueness types for handling side effects, such as keyboard input, file I/0 etc. without compromising function purity)

[2] We will later in Part II also introduce a certain level of dynamic typing.

where x is the parameter name, T the parameter's type, and e the body of the function. MKL is an *explicitly typed* language, meaning that the types of parameters must be given by the user explicitly in the program. A lambda abstraction has always just one parameter and its type is written using arrow syntax T_1 `->` T_2, where the left hand side of the arrow is the type of the parameter, and the right hand side the type of the return value. For example, the anonymous function `fun a:Int -> a + 1`, where the arrow points to the function body, has a parameter `a` of type `Int`, and a body `a + 1`. Because the type of the body is also `Int`, the type of the whole function expression is `Int -> Int`.

In a function that takes more than one parameter, these parameters are not written as in Java or C/C++ as a comma separated list enclosed in parenthesis. Instead, multiple parameters are defined using *currying*, meaning that several parameters are defined by composing several lambda abstractions. For example, consider

```
fun a:Int -> fun b:Int -> a * b
```

which is equivalent to

```
fun a:Int -> (fun b:Int -> a * b)
```

This function expression, of type `Int->Int->Int`, which takes two arguments as input and multiplies their values, can be *partially applied*, meaning that if only the first argument is supplied, a new function with the remaining parameter is returned, e.g., the expression

```
(fun a:Int -> fun b:Int -> a * b) 5
⟶ fun b:Int -> 5 * b
```

is reduced to a new function with type `Int -> Int`. Note that the expression in the function body cannot be reduced any more because no argument has been supplied to the function parameter `b`. Lambdas (anonymous functions denoted λ in the original calculus instead of `fun`), currying, and partial applications are some of the key concepts that make functional programming expressive and useful.

However, it is often convenient to give names to values. A MKL source code file consists of a sequence of *top-level* definitions giving names to expressions. If the same name is defined more than once, the last binding is used. For example, in

```
let p = 3
let q = 4
let p = 5
let r = p + q
```

variable `r` will be bound to the value 9.

Because functions are values, they can be given names in the same way, i.e., by binding an anonymous function to a name using the **`let`**-construct:

```
let multiply = fun a:Int -> fun b:Int -> a * b
```

Alternatively, the function parameters can be moved to the left of the equal sign, thus defining a function directly.

```
let multiply x:Int -> y:Int -> Int =  x * y
```

Note that this notation is non-standard and can be seen as a mixture of Haskell's convention of giving the arrow syntax above a function definition and OCaml's syntax of defining functions using **`let`**-binding. We are using the arrow notation both to name the parameters, and to relate types to the parameter names. We have found this syntax fairly intuitive and assume that its meaning is clear.

The type of `multiply` is `Int -> Int -> Int`, which is explicitly stated in the last **`let`**-binding. The arrow notation is right associative, i.e., `Int -> (Int -> Int)` is the same type, but `(Int -> Int) -> Int` is not. The latter is the type of a function that takes a function as argument and then returns an integer.

A careful reader might have noticed that the return type (the last type definition after the last arrow) of the `multiply` definition is not necessary for the type checker because it can be derived from the expression `x * y` because `*` is an integer multiplication operator that always return results of type `Int`. This is true as long as the function is non-recursive. For example, in the following definition of the factorial function

```
let fact n:Int -> Int =
  if n == 0 then 1 else n * (fact (n-1))
```

the type checker needs the return type to be able to type check the function[3] because the function is recursive (`fact` refers to itself in the false branch of the **`if`**-expression).

Note that the syntax for function application does not use parentheses. Instead the function to be applied is separated from its supplied arguments by one or more spaces. A function call that in standard mathematical calculus or in languages such as C or Java would appear as multiply(3,4) here appears as multiply 3 4. Parentheses are only used for grouping and disambiguation. This is a function call syntax used by most functional languages.

Function application has higher precedence than infix operators. For example, in the expression

```
fact (multiply (1+2) 3)
```

the expression `(1+2)` is the first argument supplied to `multiply` and `3` the second argument. The expression

```
fact ((multiply (1+2)) 3)
```

is equivalent because applications are left associative, but

```
fact (multiply 1+2 3)
```

would give a parse error.

In many situations, it is useful to pass a function as an argument to another function, or to return a function as a result of executing a function. When functions are treated as values and can be passed around freely as any other value, they are said to be *first-class citizens*. In such a case, the language supports *higher-order functions*.

Definition 7.1.1 (Higher-Order Function).
A higher-order function is a function that

1. takes another function as argument, or

[3] If the types were inferred using, e.g., Hindley-Milner type inference, this would of course not be necessary.

2. returns a function as its result.

Let us first show the former case where functions are passed as values. Consider the following function definition of `twice`, which applies the function `f` two times to `y`, and then returns the result.

```
let twice f:(Real -> Real) -> y:Real -> Real =
    f (f y)
```

The function `twice` can then be used with a function `f` that has type `Real -> Real`. We can now define a function `power2`

```
let power2 x:Real -> Real = x *. x
```

and apply function `twice` to `power2` (first argument) and a value `3.` (second argument):

```
twice power2 3.
⟶ power2 (power2 3.)
⟶ power2 (3.*3.)
⟶ power2 9.
⟶ 9. *. 9.
⟶ 81.
```

Because `twice` can take any function as an argument, we can apply `twice` to an anonymous function, passed directly as an argument to the function `twice`.

```
twice (fun x:Real -> 2. *. x -. 3.) 5.
⟶ (fun x:Real -> 2. *. x -. 3.)((fun x:Real -> 2. *. x -.3.)5.)
⟶ (fun x:Real -> 2. *. x -. 3.)(2. *. 5. -. 3.)
⟶ (fun x:Real -> 2. *. x -. 3.) 7.
⟶ 2. *. 7. -. 3.
⟶ 11.
```

Let us now consider the second part of Definition 7.1.1, i.e., a function that returns another function as its result. In mathematics, functional composition is normally expressed using the infix operator $\circ$. Two functions $f : X \to Y$ and $g : Y \to Z$ can be composed to $g \circ f : X \to Z$, by using the definition $(g \circ f)(x) = g(f(x))$.

The very same definition can be expressed in a language supporting higher-order functions:

```
let compose g:(Real->Real) -> f:(Real->Real) -> (Real->Real) =
    fun x:Real -> g (f x)
```

This example illustrates the creation of a new anonymous function and returning it from the `compose` function. The function composes the two functions given as parameters to `compose`. Hence, this example illustrates both that higher-order functions can be applied to functions passed as arguments (using formal parameters `f` and `g`), and that new functions can be created and returned as results (the anonymous function).

To illustrate an evaluation trace of the composition function, we first define another function `add7`

```
let add7 x:Real = 7. +. x
```

and then compose power2 and add7 together, forming a new function foo:

```
let foo = compose power2 add7
⟶ let foo = (fun x:Real -> power2 (add7 x))
```

Note how the function compose applied to power2 and add7 evaluates to an anonymous function. Now, the new function foo can be applied to some argument, e.g.,

```
foo 4.
⟶ (fun x:real -> power2 (add7 x)) 4.
⟶ power2 (add7 4.)
⟶ power2 (7. +. 4.)
⟶ power2 11.
⟶ 11. *. 11.
⟶ 121.
```

7.1.2 Tuples, Lists, and Pattern Matching

Tuples are the simplest form of a compound type (also called product type), containing a fixed number of ordered expressions, where each expression can have a different type. Tuples can be viewed as a simple form of records without record field names. A tuple expression is given as a comma separated list of expressions enclosed in parentheses. For example, the tuple (21,"str",**false**) has three elements and has type (Int,String,Bool). Hence, we write the type of tuples in the form (T_1, ... ,T_n) where n is the number of elements in the tuple.

A *unit type*[4] is similar to void in languages such as C. Both the *unit* expression and its type are written as an empty tuple, i.e., (). The unit type is used as the result type when a function has side effects and does not return an actual value, e.g., expression print "a string\n" prints out a string to the standard output. The type of the function print is String -> (), i.e., it takes a string as input and returns the unit expression, i.e., no value. Hence, it is the side effect of printing the string to standard output that is of interest in the computation.

A *list* is a sequence of expressions, where each expression has the same type. We write [T] for the type of a list, whose elements have type T. A list expression is written as a comma-separated list of expressions (the elements of the list), enclosed within brackets. For example, [3,1,7,0,3] is list of integers with type [Int], and ["this","is","a","list"] is a list of strings with type [String]. The expression ["text",32.10] will be rejected by the type checker because the list contains elements of both type String and Real. A list can contain other lists, for example, the expression [[1,5,2],[3,2,5],[6,2,4]] is a 3×3 matrix represented as a nested list.

A list is built up of *cons* cells, i.e., element pairs where the first element is an element of the list and the second element the rest of the list. A cons constructor is written using the infix operator ::. For example 3::9::20::[] is a list of 3 elements. The last expression [] is the empty list, which is always the last element of a cons sequence. The cons operator is right associative, i.e., the expression 3::(9::(20::[])) denotes the

[4] The name *unit type* is often used in literature for functional languages. Note that this has nothing to do with physical units of measurement.

same expression. The syntactic form of a comma separated list is just syntactic sugar for the cons cells, i.e., [3,9,20] is just another syntactic form of the same expression.

Pattern matching is a way for deconstructing tuples and lists, i.e., to take them apart. A **match**-expression in its simplest syntactic form appears as follows

$$\texttt{match}\, e\, \texttt{with} \;\; \mid p_1 \texttt{ -> } e_1 \; \ldots \; \mid p_n \texttt{ -> } e_b$$

where e is the expression to be matched, bar | preceding each matching case, e_k the chosen expression if the pattern p_k is matched, where $k \in \{1 \ldots n\}$ for n matching cases. For example,

```
let weekend day:Int -> String =
  match day with
  | 6 -> "Saturday"
  | 7 -> "Sunday"
  | x -> "Number " ++ (int2string x) ++ " is not a weekend day"
```

defines a function weekend that returns a string with the name of a weekend day. Each matching case is tried in order, and if the pattern matches, the expression on the right hand side of -> is evaluated and returned. If none of the literal values 6 or 7 matches, the *pattern variable* x matches any value and is bound to the matching value in the matching process. In this case, the built-in function int2string is used to generate a return string.

Pattern matching can be used together with recursion and higher-order functions.

```
let filter p:(Int -> Bool) -> list:[Int] -> [Int] =
  match list with
  | x::xs -> if p x then x::(filter p xs) else filter p xs
  | [] -> []
```

The function filter takes a predicate p, a list of integers list and returns a new list, with all elements that satisfy the predicate. The first case of the **match**-expression takes apart a list using the cons operator, where x is the first element of the list and xs matches the rest of the list. If the list is not empty, this matching case is used and the guard of the **if**-expression is evaluated, i.e., the predicate function p is applied to the element x. If it returns true, then a new cons cell is created with element x and with a recursive call filter with the rest of the list xs. If the guard of the **if**-expression is false, the recursive call is performed without adding x to the list, i.e., the element is filtered out. If the list is empty (matching the empty list [ ]), the empty list is returned. Expression

```
filter (fun x:Int -> x < 10) [3,22,8,99,4,12]
⟶* [3,8,4]
```

shows how an anonymous function is used as a predicate to filter integers smaller than 10.

Consider now the following example that illustrates several other properties of the matching construct:

```
let foo t:([String],Int) -> String =
  match t with
  | (["magic"],x) when x > 77 -> "The magic combination"
  | ([x,y],_) -> x ++ y
```

First, it shows that patterns can be nested in an arbitrary way. In this case we have a list `["magic"]`and a number represented by `x` as part of a tuple. In the first line, we match a singleton list with a specific value `"magic"`. The pattern variable `x` is bound and then used in the **`when`** *pattern guard*, meaning that the case will only be selected if `x` is greater than 77. The second case shows how a list with two pattern variables `x` and `y` is bound if the list has exactly two elements. We also introduce the *wildcard* pattern _ which matches any value. Some example expressions

```
foo (["magic"],100)           ⟶*  "The magic combination"
foo (["two ","strings"],10)   ⟶*  "two strings"
foo (["one"],12)              ⟶*  error
```

The last example shows an example where none of the cases matches. In such a case, a runtime error is generated.[5]

If certain types are used in several functions, it could be practical to give new names to types. At the top-level, new *type aliases* can be given, e.g.,

```
type Vector = (Real,Real,Real)
```

defines a vector in the Cartesian coordinate system. Because tuples have a defined shape at compile time (in contrast to lists which can be either cons or the empty list), the **`let`**-construct can be used directly as syntactic sugar to deconstruct a tuple. The following function defines the cross product (or vector product):

```
let crossProduct a:Vector -> b:Vector -> Vector =
  let (a1,a2,a3) = a in
  let (b1,b2,b3) = b in
  (a2 *. b3 -. a3 *. b2,
   a3 *. b1 -. a1 *. b3,
   a1 *. b2 -. a2 *. b1)
```

Note that we are here using both **`let`**-expressions at the top-level, as well as local **`let`**-expressions inside a function. In the latter case, the **`in`** keyword is used to separate the expressions, i.e., syntax **`let`** p = e_1 **`in`** e_2 states that expression e_1 is matched against pattern p. The free pattern variables in p are bound and then available in e_2. Finally, we show how the `crossProduct` function can be applied to some example vectors:

```
crossProduct (2.,8.,33.) (15.,22.,9.)
⟶* (-654.,477.,-76.)
3
```

7.1.3 Equality, Abstract Data Types, and Modules

In the current version of MKL, there is no support for *parametric polymorphism*. The main rationale for excluding such useful language property is that further research is needed to study the relation of polymorphism with the language extensions presented in the next chapters.

[5] In future versions, exhaustive checking of matching rules could be done at compile time. However, this is not available in the current prototype.

However, to be able to evaluate the approach taken in the thesis, a fairly efficient implementation is needed. For that reason polymorphic built-in *abstract data types (ADTs)* for handling finite sets, mapping between key value pairs, and efficient random access arrays have been added to the language. Appendix B defines their interfaces and the rest of this subsection gives a brief overview of their uses.

MKL has a built-in polymorphic infix operator `==` for structural equivalence test between two values. It is polymorphic in the sense that the expression on the left and right hand sides can have any type, as long as the types are equal. Hence, an implementation traverses data structures and/or functions to determine if they are equal or not. Functions are syntactically compared for equality up to α-conversion, i.e., the identifier names for function parameters do not matter. For example, the two expressions

```
321 == 150 + 171
(fun x:Int -> 1 + x,"text") == (fun y:Int -> 1 + y,"text")
```

both evaluate to `true`, but the expressions

```
[1,4,9,2] == [90,21,3]
(fun x:Bool -> true && false) == (fun x:Bool -> false)
```

both evaluate to `false`. Note that comparison of equality is always performed on values, and therefore the right hand side of the first expression is first evaluated to the value `321`. The second expression evaluates to true due to α-conversion, i.e., that bound variables can be renamed. The third is obviously false because both the size of the list and their elements differ. Finally, the fourth expression evaluates to false because functions are values and MKL is using weak reduction, i.e., evaluation is not performed under function abstractions. Besides equality of expressions, the functions of the ADTs need a comparison function to determine the total order over all types. This comparison function is not reachable as a user function, but is used internally by the ADTs. Equality and comparison of expressions are further discussed in Section 10.5.4.

All operations available within the set ADT are reached by prefixing the operator with `Set` followed by a dot. An empty set is created with expression `Set.empty`. Elements can be added by using `Set.add`, where the first argument is the element and the second the set that should be extended. The returned value is the new set. Consider the following example

```
let s1 = Set.add 10 (Set.add 20 (Set.empty))
let m1 = Set.mem 10 s1
let s2 = Set.remove 10 s1
let m2 = Set.mem 10 s2
let l = Set.toList s1
```

where `s1` evaluates to $\{10, 20\}$, `m1` evaluates to true, `s2` to $\{20\}$, `m2` to false and `l` to `[20,10]`, where we have used the notation $\{e_1, e_2, ...\}$ to denote a set constructed from elements e_1, e_2, etc. It should be noted that set operations are purely functional, i.e., no destructive updates occur. For example, the remove operations did not remove the element from `s1`; it just returned a new set `s2` where the element was removed. The type of a set is written `Set` T, where T is the type of the element in the set.

The `Map` ADT is used for storing a finite mapping of key - value pairs. The implementation is purely functional, i.e., no destructive updates occur and update operations always

return a new map. We use a double arrow to denote the type of a map T_1 `=>` T_2, where T_1 is the type of the keys and T_2 the type of the values. The operations are similar to the set, e.g. `Map.empty` creates a new empty map, `Map.add` $k\ v\ m$ adds a key k with the value v to the map m, and then returns the new updated map. If the key already exists, the returned map holds the new binding with value v. The operator `Map.find` $k\ m$ returns the value associated with the key k in m, if found. If not, a runtime error is reported. To avoid this to happen, `Map.mem` $k\ m$ should first be called to check if the key k exists in m.

The `Array` ADT implements random access of fixed sized arrays. We write $\{\ T\ \}$ for the type of an array. For example, `{(Real,Real)}` is the type of an array of tuples, whose elements are of type `Real`. Arrays differs from the other ADTs in that all of its operations are not purely functional. For example, `Array.set` $a\ p\ e$ destructively replaces the element at position p in the array a with the element e. If an access is out of bounds, a runtime error is reported.

Finally, the current version of MKL has a very simple system for separating code into different modules. The prototype implementation does not yet support separate compilation or information hiding, e.g., to create ADTs. The import mechanism from separately defined modules is a simple include mechanism. At the top-level, for example the lines

```
include Base
include Electrical
```

will include the definitions in file `base.mkl` and `electrical.mkl` into the current module. However, the include mechanism ensures both that equivalent definitions are not included twice and reports an error if the include statements of the modules introduce circular dependencies. The approach taken here is preliminary and is likely to be changed in future revisions of the language.

7.2 Lambda Calculus and Operational Semantics

In the following section, we give a brief introduction to the lambda calculus, as well as the foundation of operational semantics.

7.2.1 Untyped Lambda Calculus

The lambda calculus was invented by Alonso Church in the 1930s. Today it forms the foundation of many programming languages in general, and for functional programming languages in particular. Several books have been written about the subject, where Barendregt [12] gives a comprehensive description of the foundations, Hindley and Seldin [67] a perhaps more accessible introduction, and Pierce [124] details how it can be extended with types and other language constructs to form programming languages.

The syntax of the lambda calculus is given with the following *Backus-Naur Form (BNF)* grammar:

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \tag{7.1}$$

where e is a *lambda expression*, x a variable, $\lambda x.e$ a *lambda abstraction*, and $e_1 e_2$ an application. The above grammar should be understood as defining the *abstract syntax* of

the calculus, i.e., that ambiguities of precedence and associativity have been handled by the earlier *parsing* stage from *concrete syntax*. When discussing calculi in this thesis, we mean the abstract syntax. However, in examples and proofs, we need to write out expressions concretely. We use parentheses to remove ambiguities, but to avoid too many parentheses, we use the following convention:

The body of a lambda abstraction is counted as far to the right as possible, only ending with a closing parenthesis or end of expression. For example, the expression $\lambda z.\ \lambda x.\ \lambda y.\ x\ y\ z$ is equivalent to $\lambda z.\ (\lambda x.\ (\lambda y.\ x\ y\ z))$, but not to $\lambda z.(\lambda x.\ \lambda y.\ x\ y)\ z$, where the latter states that z is supplied as an argument to the lambda abstraction binding the variable x. The second convention is that function application is left associative, i.e., $x\ y\ z$ is equivalent to $((x\ y)\ z)$.

Grammar (7.1) is a convenient short way of stating the abstract syntax. A more formal definition can be giving inductively:

Definition 7.2.1 (Lambda expressions). Let $\mathbb{X}$ be the countable set of variable names. The set of lambda expressions $\mathcal{E}$ is the smallest set such that

- if $x \in \mathbb{X}$ then $x \in \mathcal{E}$
- if $x \in \mathbb{X}$ and $e \in \mathcal{E}$ then $\lambda x.e \in \mathcal{E}$
- if $e_1 \in \mathcal{E}$ and $e_2 \in \mathcal{E}$ then $e_1 e_2 \in \mathcal{E}$

In the coming calculi we use the simpler form defining the syntax using BNF.

A variable x in a lambda abstraction $\lambda x.e$ is said to be *bound* by the abstraction if it occurs *free* in the abstraction's body e. The set of free variables of a lambda expression e can be defined as a recursive function $FV(e)$:

Definition 7.2.2 (Free variable).

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x.\ e) & = & FV(e) - \{x\} \\
FV(e_1 e_2) & = & FV(e_1) \cup FV(e_2)
\end{array}
$$

For example, in $\lambda x.\ x\ y$ the variable x is bound by the lambda abstraction and y is a free variable of the expression. In the expression $\lambda x.\ \lambda y.\ \lambda x.\ z\ y\ x$, variable z appears free and x and y are bound. Variable x is bound by the innermost lambda. The outermost lambda does not bind any variables because x is not free in its body.

We write $[x \mapsto e]e_1$ meaning that all free occurrences of the variable x in e_1 are replaced/substituted by the expression e_1. We define substitution as follows:

Definition 7.2.3 (Substitution).

$$
\begin{array}{lcll}
[x \mapsto e]x & = & e & \\
[x \mapsto e]y & = & y & \text{if } y \neq x \\
[x \mapsto e](\lambda y.\ e_1) & = & \lambda y.\ [x \mapsto e]e_1 & \text{if } x \neq y \text{ and } y \notin FV(e) \\
[x \mapsto e](e_1\ e_2) & = & ([x \mapsto e]e_1)([x \mapsto e]e_2) &
\end{array}
$$

The computation of a lambda expression is performed by reducing expressions, so called *redexes*. The rewriting rule

$$(\lambda x.e_1)\ e \ \longrightarrow\ [x \mapsto e]e_1$$

is called the β-reduction. There are different *evaluation strategies* for the λ-calculus. They defines which order the redexes are reduced.

In *normal order* evaluation, the outermost and leftmost redex is is reduced first. E.g.,

$$\begin{aligned}(\lambda x.\ \lambda w.\ x)((\lambda y.\ y)(\lambda z.\ z)) &\longrightarrow \\ (\lambda w.(\lambda y.\ y)(\lambda z.\ z)) &\longrightarrow \\ (\lambda w.(\lambda z.\ z)) &\end{aligned}$$

Note that evaluation occurs under lambda, i.e., evaluation proceeds in the body of the lambda abstraction.

If we reconsider Definition 7.2.3, we see that the guards for the equations make the substitution a partial function. We say that Definition 7.2.3 is *capture avoiding*, meaning that a free variable cannot be accidentally captured by another lambda abstraction. For example the expression $\lambda z.\ (\lambda x.\ \lambda z.\ x)\ z$ is reduced to $\lambda z.\ ([x \mapsto z](\lambda z.\ x))$. However, the partial definition of substitution does not apply because the binding variable z is free in the substituting expression, i.e., the guard of the second equation in Definition 7.2.3. Hence, we say that we are working on expressions up to α-conversion (i.e., name conversion), meaning that the names can be consistently renamed in an expression. We make use of the following convention:

Convention 1. All expressions that differ only with names of binding variables are exchangeable in all contexts.

Hence, the binding variable z for the innermost lambda can be renamed to e.g., w, and the expression $\lambda z.\ ([x \mapsto z](\lambda w.\ x))$ can be reduced to $\lambda z.\ \lambda w.\ z$.

In the *call-by-name* evaluation strategy, also the leftmost outermost redex is reduced first, but evaluation is not allowed under lambda. E.g,

$$\begin{aligned}(\lambda x.\ \lambda w.\ x)((\lambda y.\ y)(\lambda z.\ z)) &\longrightarrow \\ (\lambda w.(\lambda y.\ y)(\lambda z.\ z)) &\end{aligned}$$

Hence, the evaluation stops after the first step because the only available redex is under a lambda abstraction.

Finally, in the *call-by-value* evaluation strategy, the argument is first reduced to a value, followed by a beta reduction of the outermost redex. In the pure lambda calculus, only lambda abstractions are values. However, we will later in this thesis enrich the language with more constructs and where other expressions are also values. The same example as above with the call-by-value strategy evaluates as follows:

$$\begin{aligned}(\lambda x.\ \lambda w.\ x)((\lambda y.\ y)(\lambda z.\ z)) &\longrightarrow \\ (\lambda x.\ \lambda w.\ x)(\lambda z.\ z) &\longrightarrow \\ (\lambda w.\ \lambda z.\ z) &\end{aligned}$$

In this thesis, we are only concerned with enriched variants of call-by-value calculi.

We now formally define the semantics of the lambda calculus with the call-by-value evaluation strategy using *small-step operational semantics*. The rules for the operational semantics are given below:

Definition 7.2.4 (Call-by-value operational semantics).

$$\frac{e_1 \longrightarrow e_1'}{e_1 e_2 \longrightarrow e_1' e_2} \text{ (E-APP1)} \qquad \frac{e_2 \longrightarrow e_2'}{v_1 e_2 \longrightarrow v_1 e_2'} \text{ (E-APP2)}$$

$$(\lambda x.e)\; v \;\longrightarrow\; [x \mapsto v]e \text{ (E-BETA)}$$

We must also give the definition of a value

$$v ::= \lambda x.e \tag{7.2}$$

i.e., a value is a lambda abstraction. The first two rules are called *congruence* rules and the last one *computation* rule. The former rules are used for "going inside" an expression, and the last one, the β-rule, reduces (computes) a redex. Note also how the meta-variables for values make the calculus deterministic. Hence, at most one rule can apply.

7.3 Chapter Summary and Conclusions

We have in this chapter introduced fundamental concepts in functional programming, by giving examples in our research language MKL. Language constructs presented so far is available in most standard functional languages, that is, new language constructs in MKL are presented in the next chapters.

We have also given a short introduction to the lambda calculus, which will be used in Chapter 10 where we present the core of MKL's semantics.

8

Modeling in MKL

A fundamental construct in an EOO languages is an acausal model (also called non-causal model). Such models can encapsulate and compose both continuous-time behavior in the form of DAEs or other interconnected components, where the direction of information flow between the components is not specified.

In functional programming languages, such as Haskell [134] and Standard ML [98], standard libraries have for a long time been highly reusable, due to the basic property of having functions as first-class values. This property, also called *higher-order functions*, means that functions can be passed around in the language as any other value.

In this chapter, we investigate the combination of acausal models with higher-order functions. We call this concept *higher-order acausal models (HOAMs)*.

A similar idea called *first-class relations on signals* has been outlined in the context of functional hybrid modeling (FHM) [109]. Giorgidze and Nilsson's later developments of this work have focused on efficient JIT compilation and aspects of embedding the constructs in Haskell [61, 62, 63]. However, that work is still at an early stage regarding formalizing the semantics. In contrast, our previous work's main objective has been to define a formal operational semantics for a subset of a typical EOO language [17]. From the technical results of our earlier work, we have extracted the more general ideas of HOAM, which was first published by Broman and Fritzson in 2008 [25]. This chapter consists of a refinement of that work, where examples have been updated to conform to the MKL syntax.

In the chapter, we show examples of using HOAMs that in principle[1] subsume three different constructs in Modelica:

[1] Because the semantics of MKL and Modelica are fundamentally different we cannot say that the constructs can directly replace each other.

- The Modelica `redeclare` construct used for replacing sub-models and components in a Modelica model with other models/components (subsumed by passing models as arguments to other models).
- The Modelica `for`-equation construct used for creating equations and connecting several components together (subsumed by using recursive HOAMs).
- The Modelica construct of conditional components used for inclusion/exclusion of a component depending on a conditional expression (subsumed by using `if`-expressions and HOAMs).

An objective of this chapter is to make the basic ideas of higher-order acausal models accessible both to engineers with little functional language programming background, as well as to computer scientists with minimal knowledge of physical acausal modeling. Hence, the chapter is structured in the following way to both reflect the broad intended audience, as well as presenting the contributions of the concept of HOAMs:

- We give an informal introduction to physical modeling in our research language MKL (Section 8.1).
- We state a definition of higher-order acausal models (HOAMs) and outline motivating examples. Surprisingly, this concept has not previously been widely explored in the context of EOO-languages (Section 8.2).
- Finally, we give examples using dynamic data structures together with higher-order models and discuss polymorphism. (Section 8.3).

8.1 Basic Physical Modeling in MKL

In conventional object-oriented programming languages, such as Java or C++, the behavior of classes is described using methods. However, in equation-based object-oriented languages, the continuous-time behavior is typically described using differential algebraic equations and the discrete-time behavior using e.g., conditional equations. This behavior is grouped into abstractions called classes or models (Modelica) or entities and architectures (VHDL-AMS). From now on we refer to such abstractions simply as *models*.

Models are blue-prints for creating *model instances* (in Modelica called components). The models typically have well-defined interfaces consisting of ports (also called connectors), which can be connected together using *connections*. A typical property of EOO-languages is that these connections usually are *acausal*, meaning that the direction of information flow between model instances is not defined at modeling time.

(I)

```
let Circuit =
  let e1:Electrical in
  let e2:Electrical in
  let e3:Electrical in
  let e4:Electrical in
  Resistor 10. e1 e2;
  Capacitor 0.01 e2 e4;
  Resistor 100. e1 e3;
  Inductor 0.1 e3 e4;
  SineVoltage 220. 50. e1 e4;
  Ground e4
```

(II)

```
model Circuit
  Resistor R1(R=10);
  Capacitor C(C=0.01);
  Resistor R2(R=100);
  Inductor L(L=0.1);
  SineVoltage AC(VA=220);
  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 Circuit;
```

Figure 8.1: *Figure (I) lists the MKL model definition of a simple electrical circuit, and (II) shows a Modelica model of the same circuit.*

In the context of EOO languages, we define acausal (also called non-causal) models using the following definition:

Definition 8.1.1 (Acausal Model).
An acausal model is an abstraction that encapsulates and composes

1. continuous-time behavior in the form of differential algebraic equations (DAEs), and/or

2. interconnected components, where the direction of information flow between components is not specified.

Sometimes, a model has both causal and acausal ports. In such a case we say that the model is *partially acausal*.

In many EOO languages, acausal models also contain conditional constructs for handling discrete events. Moreover, connections between model instances can typically both express potential connections (across) generating direct equality equations and flow (also called through) connections generating sum-to-zero equations.

8.1.1 A Simple Electrical Circuit

To illustrate the basic modeling capabilities of MKL, the source code of a simple electrical circuit is listed in Figure 8.1. Part (I) shows the corresponding textual model given in MKL. For clarity to the readers familiar with the Modelica language, we also compare to the same model given as Modelica textual code (II).

In the example `Circuit`, the model is given a name using the **`let`**-construct. The expression defining the model lists four local **`let`**-expressions. These expressions define

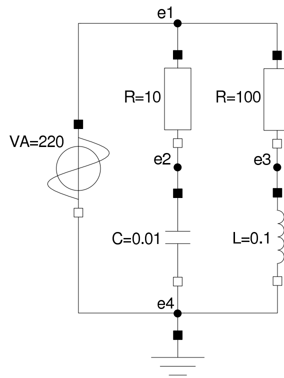


Figure 8.2: *Graphical model of a simple electrical circuit.*

four *nodes*[2] of type `Electrical`, i.e., the nodes are defined for the electrical domain. Below the definitions of nodes, six *components* are defined. Figure 8.2 shows a graphical representation of the circuit, including the six components and the nodes `e1`, `e2`, `e3`, and `e4`.

Consider the first resistor component using the following line, which contains a call to the `Resistor` by passing three arguments using the functional call notation without parenthesis.

```
Resistor 10. e1 e2;
```

The two last arguments state that nodes `e1` and `e2` are connected to this resistor instance (compare Figure 8.2 and Figure 8.1). We say that the nodes are connected to the components' *ports*. In the same manner, we can see that, e.g., node `e4` is connected to the ports of the `VoltageSource`, `Inductor`, and `Capacitor` instances. Hence, the intuition is that nodes in a model *connect* components together. The type `Electrical` of the node indicates that we are modeling in the electrical domain and that only ports of type `Electrical` can be connected to nodes of the same type. The first argument expresses that the resistance for this instance is 10 Ohm. In the same way, the first argument states that the `Inductor` has inductance 0.1.

Modeling using MKL differs in several ways compared to Modelica (Figure 8.1, part II). First, models are not defined anonymously in Modelica and are not treated as first-class citizens. Second, the way in which acausal connections are defined between model instances differs. In MKL, the connection (in this case the electrical nodes), is created and then connected to the model instances by giving it as an argument to the creation of sub-model instances. In Modelica, a special `connect`-equation construct is defined in the language. This construct is used to define binary connections between connectors of sub-model instances. From a user point of view, both approaches can be used to express acausal connections between model instance. Hence, we let it be up to the reader to judge what is the most natural way of defining interconnections. However, from a formal semantics point of view, with regards to HOAMs, we have found it easier to encode

[2]Nodes are similar to what is called *terminals* in VHDL-AMS.

connections using the ordinary parameter passing style, exemplified here using MKL.

8.1.2 Models and Equation Systems

The main model in this example is the `Circuit` model. This model contains instances of other models, such as the `Resistor` model:

```
let Resistor R:Real -> p:Electrical -> n:Electrical -> Equations=
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  R *. i = v
```

In the same way as for `Circuit`, this model is defined using the **`let`**-construct followed by formal parameters with type annotations. In this `Resistor` model, the first formal parameter `R` of type `Real` states the resistance of a component. The second and third formal parameters are the ports of the component, i.e., the connection points. The type for both formal parameters is `Electrical`, which is the type of a node in the electrical domain. Recall that the nodes in model `Circuit` had type `Electrical`. Hence, acausal connections in MKL are simply parameter passing of nodes. Note that these connections do not state any causality and order of computation. The control flow and approach to solve the equation system is not defined at modeling time.

The return type given for the resistor model is of type `Equations`. This is the return type of any acausal model defined in the language. Hence, when an instance is created of the `Resistor` model, the returned value is a system of equations. Several systems of equations are composed together using the operator `;` (recall the use of this operator in the `Circuit` model in Figure 8.1). In the next chapter, we go into the details about what the `Equations` type actually mean. However, for the purpose of illustrating physical modeling capabilities, it is enough to view it as an abstract concept representing a system of equations.

The body of the `Resistor` model consists of four lines of code. The first two lines define two new unknown variables `i` and `v`. Note that compared to an ordinary `let`-expression, no value is bound to the variable. Hence, `let`-expressions without a binding value are treated as unknowns in the model. These unknowns are then later solved during simulation.

The third line states an `ElectricalBranch`. The purpose of the branch is twofold. First, it is used to bind equations to the unknowns `i` and `v` (first and second arguments). The unknown current `i` is the current flowing through the component. The unknown voltage `v` is the voltage drop over the component, i.e., the potential difference between the positive port `p` and negative port `n`. The third and fourth arguments are the nodes coming from the ports of the model. The second purpose of the branch equation is for generating equations conforming to Kirchhoff's current law. The details of this connection semantics are described and discussed in Section 11.2. However, the intuition of the `ElectricalBranch` is that it is path of flow through a component and between two nodes. From a graph theoretical point of view, we would say vertex instead of node, edge instead of branch, and graph instead of network. However, we will use the former terminology because it is commonly used in the electrical domain and also in related languages, such as VHDL-AMS [10].

The fourth line states an equation describing the continuous-time behavior of the model. For the `Resistor` model, it is simply an algebraic equation stating Ohm's law. The `Inductor` model is defined as follows:

```
let Inductor L:Real -> p:Electrical -> n:Electrical -> Equations=
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  L *. (der i) = v
```

and similarly, the `Capacitor` model:

```
let Capacitor C:Real -> p:Electrical -> n:Electrical -> Equations=
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  C *. (der v) = i
```

The main difference compared to the `Resistor` model is that the `Inductor` and the `Capacitor` models contain differential equations, where the `i` and `v` variables are differentiated with respect to time using the `der`-operator. The voltage source model is defined as follows:

```
let SineVoltage V:Real -> f:Real ->
                p:Electrical -> n:Electrical -> Equations =
  let PI = 3.1415 in
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  v = V *. sin(2. *. PI *. f *. time)
```

The first model `SineVoltage` specifies a time varying voltage source, by using a sine function `sin` with type `Real -> Real`, and the global time variable `time`, which gives the current time in seconds. Note also that we have defined a constant `PI` using an ordinary `let`-expression. The last model in this example `Circuit` is the `Ground` model:

```
let Ground p:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
  ElectricalRefBranch i v p;
  v = 0.
```

Similarly to what is done in Modelica, the `Ground` model is modeled by binding a potential voltage variable to zero. The main difference of the `Ground` model compared to the other models is that it is using a reference branch equation `ElectricalRefBranch` instead of an `ElectricalBranch`. The reference branch takes only one node as argument, in this case, the node coming from the positive port `p`. The unknown `v` gives the absolute potential value of the node coming from the parameter `p`. The unknown `i` represents the current flowing through the component, which in this case ought to be zero.

The careful reader might now ask what the difference is between the definition of an acausal model using a `let`-expression and a higher-order function defined using a `let`-expression? The answer is: none. Acausal models in MKL are defined and abstracted

using ordinary functions. The difference is that these models contain unknowns defined by using `let`-expressions that lack a binding to a value. Moreover, the actual computation of solving the equations are delayed to a later stage. We will postpone the discussion of the details of the exact meaning of these constructs to the next chapter, and instead give a bigger example of higher-order modeling.

Note also that we have as a convention to start ordinary function names with a lower-case letter (e.g., the `fact` factorial function in Chapter 7) and model names with an upper-case letter (e.g., `Circuit` or `Resistor`). Even if both are functions from the language point of view, we find that this convention clarifies the source code.

8.2 Higher-Order Acausal Modeling

In EOO languages models are typically treated as compile time entities, which are translated into hybrid DAEs during the compiler elaboration phase. We have previously seen how functions can be turned into first-class citizens, passed around, and dynamically created during evaluation (see Chapter 7). Can the same concept of higher-order functions semantics be generalized to also apply to acausal models in EOO languages? If so, does this give any improved expressive power in such a generalized EOO language?

In this section we describe concrete examples of acausal modeling using MKL. However, let us first define what we actually mean by higher-order acausal models.

Definition 8.2.1 (Higher-Order Acausal Model (HOAM)).
A higher-order acausal model is an acausal model, which can be

1. parameterized with other HOAMs.
2. recursively composed to generate new HOAMs.
3. passed as an argument to, or returned as a result from functions.

In the first case of the definition, models can be parameterized by other models. For example, the constructor of an automobile model can take as an argument another model representing a gearbox. Hence, different automobile instances can be created with different gearboxes, as long as the gearboxes respect the interface (i.e., the type) of the gearbox parameter[3] of the automobile model.

The second case of Definition 8.2.1 states that a model can reference itself; resulting in a recursive model definition. This capability can for example express models composed of many similar parts, e.g., discretization of flexible shafts in mechanical systems or pipes in fluid models.

Finally, the third case emphasizes the fact that HOAMs are first-class citizens, e.g., that models can be both passed as arguments to functions and created and returned as results from functions. Hence, in the same way as in the case of higher-order functions, generic reusable functions can be created that perform various tasks on arbitrary models, as long as they respect the defined types (interfaces) of the models' formal parameters. Consequently, this property enables *model composition* to be defined and executed within

[3]We refer here to formal parameters when the model is viewed as a function. It is not related to the concept of parameters in Modelica, which are constants during simulation time.

the modeling language itself. For example, certain discretizations of models can be implemented as a generic function and stored in a standard library, and then reused with different user defined models.

Some special and complex language constructs in currently available EOO languages express part of the described functionality (e.g., the redeclare and for-equation constructs in Modelica). However, in the sequential sections we show that the concept of higher-order acausal models is a small but very powerful and expressive language construct that subsumes and/or can be used to define several other, more complex language constructs. Whether the end user finds this more functional approach of modeling easy or hard depends of course on many factors, e.g., previous programming language experiences and syntax preferences. However, from a semantic point of view, we show that the approach is very expressive because few language constructs enable rich modeling capabilities in a relatively small kernel language.

We will now in the rest of this section exemplify three kinds of uses of HOAM by giving examples in MKL.

8.2.1 Parameterization of Models with Models

A common goal of model design is to make model libraries extensible and reusable. A natural requirement is to be able to parameterize models with other models, i.e., to reuse a model by replacing some of the sub-models with other models. To illustrate the main idea of parameterized acausal models, consider the following over-simplified example of an automobile model:

```
let Automobile Engine:(Rotational -> Equations) ->
               Tire:(Rotational -> Equations) ->
               Equations =
  let r1:Rotational in
  let r2:Rotational in
  Engine r1;
  Gearbox r1 r2;
  Tire r2; Tire r2; Tire r2; Tire r2
```

In the example, the automobile is defined to have two formal parameters; an `Engine` model and a `Tire` model. To create a model instance of the automobile, the model can be applied to a specific engine, e.g., a model `EngineV6` and some type of tire, e.g. `TireTypeA`:

```
Automobile EngineV6 TireTypeA;
```

If later on a new engine was developed, e.g., `EngineV8`, a new automobile model instance can be created by changing the arguments when the model instance is created, e.g.,

```
Automobile EngineV8 TireTypeA;
```

Hence, new model instances can be created without the need to modify the definition of the `Automobile` model. This is analogous to a higher-order function which takes a function as a parameter, and somewhat related to a Modelica model with replaceable formal parameters. In fact, because acausal models are abstracted by higher-order functions in MKL, the *same* fundamental semantics is used in both cases.

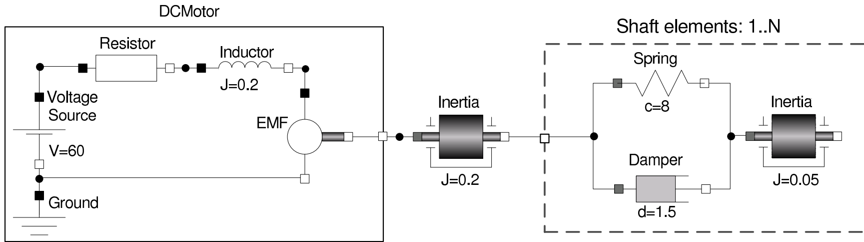


Figure 8.3: *A simple mechatronic system with a direct current (DC) motor to the left and a flexible shaft to the right. The flexible shaft consists of* 1 *to* N *elements, where each element includes an inertia, a spring, and a damper.*

In the example above, the definition of `Automobile` was not parametrized on the `Gearbox` model. Hence, the `Gearbox` definition must be given in the lexical scope of the `Automobile` definition. However, this model could of course also be defined as a parameter to `Automobile`.

This way of reusing acausal models has obvious strengths, and it is therefore not surprising that constructs with similar capabilities are available in some EOO languages, e.g., the special `redeclare` construct in Modelica. However, instead of creating a special language construct for this kind of reuse, we believe that HOAMs can give simpler and a more uniform semantics of an EOO language.

8.2.2 Recursively Defined Models

In many applications it is enough to hierarchically compose models by explicitly defining model instances within each other (e.g., the simple `Circuit` example). However, sometimes several hundred model instances of the same model should be connected to each other. This can of course be achieved manually by creating hundreds of explicit instances. However, this results in very large models that are hard to maintain and get an overview of.

One solution could be to add a loop-construct to the EOO language. This is the approach taken in Modelica, with the `for`-equation construct. However, such an extra language construct is actually not needed to model this behavior. Analogously to defining recursive functions, we can define *recursive models*. This gives the same modeling possibilities as adding the `for`-construct. However, we have also found it easier to define a compact formal semantics of the language using this construct.

Consider Figure 8.3 which shows a simple mechatronic model, i.e., a model containing components from both the electrical and mechanical domain. The left hand side of the model shows a direct current (DC) motor. The electromotoric force (EMF) component converts electrical energy to mechanical rotational energy.

In the middle of the model in Figure 8.3 a rotational body with Inertia $J = 0.2$ is defined. This body is connected to a flexible shaft, i.e., a shaft which is divided into a number of small bodies connected in series with a spring and a damper in parallel in between each pair of bodies. Variable N is the number of elements that the shaft consists of.

A model of the mechatronic system is described by the following MKL source code.

```
let MechSys =
  let r1:Rotational in
  let r2:Rotational in
  let r3:Rotational in
  DCMotor r1;
  Inertia 0.2 r1 r2;
  FlexibleShaft 120 r2 r3
```

If we recall from the previous section, the connection between electrical components was defined using `Electrical` nodes. However, in the rotational mechanical domain, the connection is instead defined by creating `Rotational` nodes. In the `MechSys` model, three such nodes are created. These nodes are used for connecting the `DCMotor`, the `Inertia` and the `FlexibleShaft` components together.

The most interesting part is the definition of the `FlexibleShaft` component. This shaft is connected to the Inertia to the left. To the right, it is connected to node `r3`, which is not connected to any other port. Hence, ports always need to be connected to a node, even if it is a singleton node. The first argument supplied to the `FlexibleShaft` model states the number of elements that the shaft should consist of; in this case 120 elements.

Can these 120 elements be described without the need of code duplication? Yes, by the simple but powerful mechanism of recursively defined models. Consider the following self-explanatory definitions of `ShaftElement`:

```
let ShaftElement flangeA:Rotational -> flangeB:Rotational ->
                 Equations =
  let r1:Rotational in
  Spring 8. flangeA r1;
  Damper 1.5 flangeA r1;
  Inertia 0.5 r1 flangeB
```

This model represents just one of the 120 elements connected in series in the flexible shaft. The actual flexible shaft model is recursively defined and makes use of the `ShaftElement` model:

```
let FlexibleShaft n:Int -> flangeA:Rotational ->
                  flangeB:Rotational -> Equations =
  if n == 1 then
    ShaftElement flangeA flangeB
  else
    let r1:Rotational in
    ShaftElement flangeA r1;
    FlexibleShaft (n-1) r1 flangeB
```

The recursive definition is a standard recursively defined function, where the `if`-expression evaluates to false, as long as the count parameter `n` is not equal to 1. For each recursive step, a new connection is created by defining `r1`, which connects the shaft elements in series. Note that the last element of the shaft is connected to the last argument supplied to the `FlexibleShaft` model because the shaft element created when the `if`-expression is evaluated to true takes parameter `flangeB` as an argument.

When the `MechSys` model is elaborated using our MKL prototype implementation, it results in a DAE consisting of 1586 equations and the same number of unknowns[4]. It is obviously beneficial to be able to define recursive models in cases such as the one above, instead of manually creating 120 instances of a shaft element.

However, it is still a bit annoying to be forced to write the recursive model definition each time one wants to serialize a number of model instances. Is it possible to capture and define this serialization behavior once and for all, and then reuse this functionality?

8.2.3 Higher-Order Functions for Generic Model Composition

In the previous section we have seen how models can be reused by applying models to other models, or to recursively define models. In this section we show that it is indeed possible to define several kinds of generic *model composition* strategies by using higher-order functions. These functions can in turn be part of a modeling language's standard library, enabling reuse of model composition functions.

Recall the examples from Chapter 7 of higher-order functions returning other anonymously defined functions. Assume that we want to create a general function, which can take any two models that have two ports defined (e.g., `Inertia` or `ShaftElement`), and then compose together by connecting them in parallel, and then return this new model:

```
let composeParallel
      model1:(Rotational -> Rotational -> Equations) ->
      model2:(Rotational -> Rotational -> Equations) ->
      (Rotational -> Rotational -> Equations) =
  (fun flangeA:Rotational -> fun flangeB:Rotational ->
     model1 flangeA flangeB;
     model2 flangeA flangeB)
```

However, for example, our `Spring` model does not take two arguments, but three, where the first one is the value for the particular component (e.g., spring constant for the `Spring` model and moment of inertia for the `Inertia` model). Because models are higher-order functions and multiple formal parameters are defined using currying, the solution is basic *partial application* where the first argument is supplied to the model. For example, a new function `comp` that composes a spring and a damper can be defined as follows:

```
let comp = composeParallel (Spring 8.) (Damper 1.5)
```

A standard library can then further be enhanced with other higher-order functions, e.g., a function that composes two models in series:

```
let composeSerial model1:(Rotational -> Rotational -> Equations)->
                  model2:(Rotational -> Rotational -> Equations)->
                  (Rotational -> Rotational -> Equations) =
  (fun flangeA:Rotational -> fun flangeB:Rotational ->
     let flangeM:Rotational in
```

[4]Using our elaboration approach, this is approximately half the number of equations generated by using the Modelica elaboration semantics (2922 equations for the corresponding Modelica model). The main reason is that our semantics do not need to generate equations for potential variables between connectors. Details of this elaboration semantics is given in Chapter 11.

```
    model1 flangeA flangeM;
    model2 flangeM flangeB)
```

Note that this time a new node is created between `model1` and `model2`. Consider now the following new definition of the ShaftElement:

```
let ShaftElement =
  let comp = composeParallel (Spring 8.) (Damper 1.5) in
  composeSerial comp (Inertia 0.5)
```

This results in the exact same model instance as the one giving before, where the components were connected using node connections. Hence, models can be composed by either direct node connection, or by utilizing higher-order functions for generic model composition.

We have now created two simple higher-order functions which compose models in parallel and in series. However, can we create a function that takes a model M and an integer n, and then returns a new model where n number of models M have been connected in series? If this is possible, we do not have to create a special recursive model for the `FlexibleShaft`, as shown in the previous section.

Fortunately, this is indeed possible by combining a recursive model (e.g., a recursive function) and a higher-order function. Consider the following definition of a serialization function for the rotational domain:

```
let serializeRotational
          n:Int -> model:(Rotational -> Rotational -> Equations) ->
          (Rotational -> Rotational -> Equations) =
  let recmodel n:Int -> flangeA:Rotational ->
               flangeB:Rotational -> Equations =
    if n <= 1 then
      model flangeA flangeB
    else
      let nodeNew:Rotational in
      model flangeA nodeNew;
      recmodel (n-1) nodeNew flangeB
    in
      (fun flangeA:Rotational -> fun flangeB:Rotational ->
       recmodel n flangeA flangeB)
```

The main function definition `serializeRotational` contains a local function definition `recmodel`. A closer look at the latter function shows that it is almost identical to the definition of model `ShaftElement`. The main difference is that in `recmodel`, instances are created from the variable `model`, which is the second parameter of the function `serializeRotational`.

The last part of the main function returns a curried function representing a rotational model of two ports (`flangeA` and `flangeB`). The body of this function is calling `recmodel`, resulting in a composed model with `n` elements in series.
Now, we can once again define the mechatronic system given in Figure 8.3, but this time by using function `serializeRotational`:

```
let MechSys2 =
  let r1:Rotational in
```

```
let r2:Rotational in
let r3:Rotational in
DCMotor r1;
Inertia 0.2 r1 r2;
(serializeRotational 120 ShaftElement) r2 r3
```

Even if the serialize function might seem a bit complicated to define, the good news is that such functions usually are created by library developers and not end-users. Fortunately, the end-user only has to call the serialize function and then use the newly created model.

8.3 Dynamic Data Structures and Polymorphism

We have in the previous section created a flexible function for connecting series of models in the rotational domain. However, how can we connect a list of models in series, which are not configured in the same way, i.e., do not have the same inertia or spring constant? Moreover, do we need to create a new version of e.g. the serialization functions for each physical domain? These questions are the topics for this section.

8.3.1 Model Composition over Lists of Models

Let us a assume that we want to model the flexible shaft again, but with a few hypothetical requirements:

- The shaft element should be modeled in the same way as in the previous section, with the difference that the inertia `J` can be different for each shaft element.
- The first element should have inertia 0.5 and then each element should have an increased intertia of 0.2.
- An inertia component of an element should only exist if the expected inertia is greater than 5.0, otherwise it should not be inserted in the model.
- There should be no shaft elements with inertia larger or equal to 10.0, i.e., this states the termination condition for the generation of the shaft elements.

To start with, we define a type synonym for a rotational model with two ports:

```
type RotModel = Rotational -> Rotational -> Equations
```

The idea would be to create a list of shaft element models which is valid for the above requirements and then to connect all these models in series. A general serialization function over a list of models in the rotational domain can be defined as follows:

```
let serializeList models:[RotModel] -> RotModel =
  let serialize flangeA:Rotational -> flangeB:Rotational ->
                models:[RotModel] -> Equations =
    match models with
    | [m] -> m flangeA flangeB
    | m::ms ->
       let flangeM:Rotational in
```

```
        m flangeA flangeM;
        serialize flangeM flangeB ms
     | [] -> error "illegal to serialize an empty list"
  in
   (fun flangeA:Rotational -> fun flangeB:Rotational ->
     serialize flangeA flangeB models)
```

A local function is performing the recursion over the list. The list is deconstructed using pattern matching, and returning a composed new model, where all model elements are connected in series.

To meet the requirements, one way could be to create a new shaft element, which is parameterized by the inertial `J`:

```
let ShaftElement2 J:Real =
  let comp = composeParallel (Spring 8.) (Damper 1.5) in
  if J >. 5. then composeSerial comp (Inertia J) else comp
```

Note that this variant of the shaft element is only adding an Inertia instance, if `J` is larger than 5. This selection is performed with a standard `if`-expression.

We also need to generate the list of shaft elements, where the inertia is increased with `0.2` for each element. This is straightforward using a recursive function:

```
let genElemList J:Real -> incrJ:Real -> maxJ:Real -> [RotModel] =
  if J <. maxJ then
    (ShaftElement2 J)::(genElemList (J +. incrJ) incrJ maxJ)
  else []
```

The first parameter is the inertial, the second the increment value and the third the max value, i.e., deciding when to finish the recursion.

Finally, we create an instance of this new flexible shaft inside the mechatronic system:

```
let MechSys3 =
  let r1:Rotational in
  let r2:Rotational in
  let r3:Rotational in
  DCMotor r1;
  Inertia 0.2 r1 r2;
  (serializeList (genElemList 0.5 0.2 10.)) r2 r3
```

8.3.2 Parametric Polymorphism

We have in the previous section illustrated some of the modeling flexibility and expressiveness of being able to use higher-order models together with data structures such as lists. However, so far we have modeled all components within a specific domain; in this case the mechanical rotational domain. For example, recall that the function `serializeList` was specially defined for the `RotModel` type

```
type RotModel = Rotational -> Rotational -> Equations
```

where the ports were `Rotational` nodes. Would it not be good to be able to create one function that could take models from *any* physical domain and connect such models

in series? The solution would be to incorporate *parametric polymorphism* in the language, i.e., to have type variables in the **let**-expression. For example, a generic variant of `serializeList` could look like

```
let serializeList models:['a -> 'a -> Equations] ->
                  ('a -> 'a -> Equations) =
  ...
```

where the type variables are denoted as identifiers prefixed by a single-quote `'`. However, the current version of our experimental platform is monomorphic with regards to types and therefore such a definition as above is not allowed. At first glance, this extension seems straightforward to incorporate into the language, but a more in depth study needs to be performed to see how it interacts with the other constructs of the model lambda calculus, which is presented in Chapter 10. We leave as future work the study of incorporation of parametric polymorphism into the language.

8.4 Chapter Summary and Conclusions

We have in this chapter given a basic introduction of the fundamentals of mathematical modeling in MKL. In particular, we study and give examples of how the concept of higher-order acausal model (HOAM) can be used for modeling.

We conclude that HOAMs can be used to express several of the language constructs available in Modelica, such as `for`-equations, `redeclare`-constructs, and conditional components. We have also shown how HOAMs can be used together with lists for flexible modeling. However, it is to early to draw any general conclusions that this modeling technique is easy to use from a user perspective.

9

Intensional Analysis of Models

IN metaprogramming environments and systems, *metaprograms* manipulate, transform and analyze *object programs*. Metaprograms can, for exampl,e be compilers, theorem provers, transformation systems, partial evaluators, and analyzers. The aim of metaprograms can be to increase the expressive power of a system and/or to increase the performance of systems.

Programming languages capable of *intensional analysis* of code are program analyzers, i.e., metaprograms that inspect and analyze an object program. *Extensional metaprogramming* languages are on the other hand languages that only generate new programs using information available in the metaprogram.

Designing and implementing model libraries for different domains often requires deep knowledge of the mechanisms of the underlying language, as well as implementation details of the used compiler or software tool. Mathematical models include equations of different forms. The system of equations in such models can be inspected and analyzed in several ways and for different needs. In this chapter, we introduce the idea of using *intensional analysis of mathematical models*, i.e., the possibility for a function to inspect and analyze the equation system of the model. The programming is *homogeneous*, meaning that the program manipulates the models created in the same language. The chapter is organized as follows:

- We explain the concept of models, unknowns, and model types of MKL (Section 9.1).

- We describe, by giving several examples, how pattern matching can be used for intensional analysis of models (Section 9.2).

9.1 Models and Unknowns

In the previous chapter, we introduced variables that are both algebraic and appear differentiated, e.g., recall the model definition of capacitor:

```
let Capacitor C:Real -> p:Electrical -> n:Electrical -> Equations=
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  C *. (der v) = i
```

In the model, we have defined two unknown variables: `i` of type `Current` and `v` of type `Voltage`. We say that these variables are binding *unknowns* because they are not bound to any explicit values by the **`let`**-expressions. The unknown `v` appears differentiated (expression `(der v)`), while `i` does not and is therefore an algebraic variable.

From a modeling point of view, one might expect that the `der` operator and the definitions of `Current` and `ElectricalBranch` are part of the MKL language. However, they are not. Instead these construct are defined in MKL standard libraries.

9.1.1 Unknowns

The fundamental construct for mathematical modeling in MKL is the concept of unknown. Unknowns are defined using the syntax

$$\texttt{\textbf{let}}\ x\texttt{:}T\ \texttt{\textbf{in}}\ e$$

where x is the variable bound to the unknown, T the type of the unknown and e the expression where x appears bound.

Unknowns are values and first-class citizen, i.e., they can be passed around as any other value. Note that this is why we say that the variable x is *bound* to an unknown, i.e., an unknown is created when the **`let`**-expression is evaluated.

9.1.2 Model Type

The concrete syntax of types in MKL is defined by the following grammar:

$$\begin{aligned} T ::= {} & \texttt{Int} \mid \texttt{Real} \mid \texttt{Bool} \mid \texttt{String} \mid T\texttt{->}T \mid \texttt{()} \mid \\ & \texttt{[}T\texttt{]} \mid \texttt{(}T_1\texttt{,}\ldots\texttt{,}T_n\texttt{)} \mid \texttt{Set}\ T \mid T\texttt{=>}T \mid \{T\} \mid \\ & \texttt{<}T\texttt{>} \mid \texttt{<>} \end{aligned}$$

Types $<T>$ and $<>$ are called *model types*. Type $<T>$ is a *specific model type* and said to be a T-model type. For example, <Int> is an integer model type and <Real -> Real> is a function model type. Type $<>$ is called the *any model type*. The intuition is that a value of a model type is a mathematical model, i.e., it includes unknowns and equations.

From a modeling point of view, the terms *unknown* and *variable* are often used with no difference in their meaning. However, from our language design point of view, they are different concepts.

Definition 9.1.1. An *unknown* is the value that is created by evaluating a let-expression of form `let` x`:`T `in` e. The new unknown will be bound by x and substituted in expression e.

Unknowns are here defined to be values (the unknown value) and can never be bound to another value, while variables are bound within function abstractions and can be substituted by another value.

An unknown is always of a model type. When binding an unknown to a variable using a **let**-expression, the type is syntactically checked. For example, expression

```
let x:<Real> in x
```

defines `x` as a `Real` model type. Now, consider the expression

```
let x:<Real> in
  x +. 1.
```

that is a model of type `<Real>`. The add operator `+.` cannot compute the result of the addition because `x` is unknown.

During type checking of the expression, the type checker checks if either of the operand expressions are of a model type. If this is the case, the operand that was not a model is embedded into a new expression, the *model value expression*[1]:

`val` e

where e is the expression that after evaluation will be a value. The value expression has two purposes. First, it turns the type of an expression e of type T into a model, where expression **val** e has type <T>. Secondly, the type T is internally tagged (not visible to the user) to the expression and can later be used to match the type of the value during intensional analysis.

This expression can be explicitly given by the user, for example in

```
let x:<Real> in
  x +. (val 1.)
```

where both `x` and expression **val** `1.` are explicit model types. In both examples, the operands are models, where in the former the `val` expression was inserted by the type checker, while in the latter case it was explicitly stated by the user.

The add operator is itself a function, which is defined in the standard library `base.mkl` (see Appendix D.1) as

```
let (+.) : Real -> Real -> Real = @@real_add
```

The line states that a prefix operator `(+.)` takes two arguments of type `Real` (curried form) and returns a `Real`. The actual operation is performed by the built-in function `@@real_add`, which has `@@` symbols prefixed to distinguish it from other user defined identifiers. The operator can be used as either an infix operator e_1 `+.` e_2 or as a prefix operator `(+.)` e_1 e_2.

[1] The explanation given in this chapter is informal and somewhat simplified. Details are given in the next chapter.

The type checker has a similar procedure for the function value itself, as for its arguments. If both arguments are of model type, the function value is embedded into a `val` expression. Hence, our expression will, after the type checking phase, have the form

```
let x:<Real> in
  (val (+.)) x (val 1.)
```

when we write the add operator in prefix form.

9.1.3 Models as Data Structures

The procedure informally described in the previous section, is related to, but different from *binding-time analysis (BTA)*, used in offline partial evaluators [66, 80]. During BTA, it is determined, given some static input data, if expressions in a program can be safely evaluated statically (at compile time) or if the computation of the expression must be delayed until runtime. However, for our translation steps performed during type checking, decisions are made regarding which expressions *cannot* be evaluated at runtime. No evaluation is performed statically at compile-time (in the context of model types). Consider the following slightly larger expression:

```
let y = 2. in
let x:<Real> in
  x +. 3. *. y
```

After the translation phase during type checking, the program is

```
let y = 2. in
let x:<Real> in
  (val (+.)) x (val ((*.) 3. y))
```

Hence, operator `(+.)` itself is embedded into a **`val`**-expression, but in the case of the multiplication, the whole expression `((*.) 3. y)` is embedded. Why the difference? The answer lies in the type returned by the type checker. For the operator `(+.)`, its second operand needs to be a model because the unknown `x` is a model by definition. However, in the case of `(*.)`, both arguments have type `Real`, i.e., they are not models. Hence, that expression can be safely evaluated during runtime.

The above example is what can be entered as an MKL program, but internally in the compiler, a different operator is used for application, called *model application*. The reason is that the application should be treated as data, i.e., the computation should not be carried out directly. The model application is here denoted with the infix @ symbol:

```
let y = 2. in
let x:<Real> in
  ((val (+.)) @ x) @ (val ((*.) 3. y))
```

An expression e_1 @ e_2, a model application of e_1 to e_2 can from a dynamic semantics point of view be considered as a tuple holding a pair of expressions. Hence, the model translation creates a data structure from an expression, so that operations are not unsafely performed on unknowns. The result of evaluating the above expression is

```
let x:<Real> in
  ((val (+.)) @ x) @ (val (6.))
```

Hence, sub-expressions that are safe to evaluate can be evaluated, but expressions containing unknowns are translated into data structures.

9.2 Intensional Analysis of Models

Intensional analysis, in the literature of metaprogramming languages, is referred to as the ability of the language to observe the structure of the code and its decomposition [129]. The information gathered during the analysis can then be used to synthesize a result. In the case of our mathematical modeling aims, the purpose would be to observe and inspect the equation system, symbolically manipulate it (e.g., symbolic differentiation), and solve the equation system. Alternatively, the system of equations can be analyzed and then translated into another form, e.g., compiled into another target language.

In the following section we show how intensional analysis can be performed on models in MKL, i.e., on expressions of a model type. In this section we explain the syntax and semantics informally from a programmers point of view. Hence, we will be using the high-level constructs (e.g., pattern matching) which are then translated into more primitive constructs for the core of the language. In Chapter 10 we explain the core language formally.

9.2.1 Pattern Matching on Models

In the previous section we stated that models are data structures which can include unknowns. To be able to inspect the model we need to deconstruct the data structure. This can be performed with *pattern matching* on models. A model can be built up of three kinds of elements:

- *Unknown.* An unknown is the value that is created by a **`let`**-expression of form `let` x`:`T `in` e.
- *Model Application.* A value v_1 applied to another value v_2, where the application is treated as data.
- *Model Value.* The expression `val` v, which embeds a value v.

The syntax of patterns for deconstructing models are as follows:

$$p \ ::= \ \mathbf{uk} : T \mid p_1 \, p_2 \mid \mathbf{val} \; x : T \mid x$$

where x is a pattern variable and T represent types. The pattern for unknowns **`uk`** $:T$ matches if the matching value is an unknown and was defined with a type that is equal to T. Note that you can only test if an unknown match with a specific type, you cannot extract the type. The pattern for model applications, written as an application $p_1 \; p_2$, consists of two patterns. Hence, it is possible to nest these model application patterns[2].

[2]Note that we did not use the @ symbol for model application in patterns. This symbol was only used in the previous section to emphasize the difference of applications and model applications when they are stored in an internal form in the program.

The pattern matches the matching value if it is a model application. The last pattern for model values **val** x:T matches if the matching value is a model value and the type of the value embedded in the model value is consistent with type T. In such case, pattern variable x is bound to the embedded value. The last pattern x is the pattern variable.

Consider now the following simple model of an expression with two unknowns:

```
let M =
  let k = 2. in
  let x:<Real> in
  let y:<Real> in
    x +. 3. *. y *. (x +. k *. 4.)
```

We would now like to analyze model `M` and create a function that returns all constants with type `Real`:

```
let constReal m:<> -> acc:[Real] -> [Real] =
  match m with
  | m1 m2 -> constReal m2 (constReal m1 acc)
  | val v:Real -> v::acc
  | _ -> acc
```

The function `constReal` takes as input a model `m`, an accumulator[3] list `acc`, and returns a list of constants found in the model. The expression to be matched is `m`. In the first case, a pattern for a model application is matched. In such a case recursive calls are made both with `m1` and `m2` as arguments. Parameter `m` has type any model `<>`, making it possible perform the recursive call regardless of the specific model type of the supplied argument. The accumulator list `acc` is used for storing/accumulating (through `v::acc`) all constants that are found, and is therefore threaded through the recursion. If the matching value is not a model application, but a model value, case two applies. Moreover, the matching case checks that the type of `v` is `Real`. In such a case, `v` is added to the accumulator list. Finally, if none of the other cases apply, the accumulator list is returned.

If the expression `constReal M []` is evaluated, we get the list `[8.,3.]`. We supply an empty list as the second argument to `constReal`. The first thing to notice is that the model application could be used to traverse the whole model. The reason is that all operators, including the primitive operators on basic types, are represented as curried functions. In the second case, model values are matched where the type of the embedded value should be `Real`. The second thing to notice is that because we know that the embedded value has type `Real`, we can use it in a type-safe way for further computation. The third observation to be made is that we only get a list of two elements, but the model contains three constants, `3.`, `4.`, and `2.`. The latter was bound by variable `k`. The reason is that the model translation during type checking only converts the whole expression `k *. 4.` into a model value. Hence, when the expression `M` is evaluated, `k` will be substituted by `2.` and the embedded model value will be computed to `8.`.

Consider now another function for pretty-printing the model, i.e., for creating a string representation of the model:

```
let pprint m:<Real> -> String =
  match m with
```

[3] An accumulator list is a formal parameter used to accumulate results during recursive calls of the function.

```
| e1 +. e2 -> "(" ++ (pprint e1) ++ " +. " ++ (pprint e2) ++ ")"
| e1 *. e2 -> (pprint e1) ++ " *. " ++ (pprint e2)
| val v:Real -> real2string v
| uk:<Real> -> "uk"
```

The function `pprint` takes a model `m` as input and returns a pretty-printed `String`. The first two cases match the infix versions of operators `+.` and `*.`. The infix representation of the pattern makes the rule more readable, but is actually just syntactic sugar for the prefix pattern rule used in combination with a **when** pattern guard. For example, the second case is equivalent to:

```
| op e e2 when op == (*.) -> (pprint e1) ++" *. "++ (pprint e2)
```

where the pattern is a nested model application pattern. The third case in the **match**-expression in function `pprint` matches a model value and converts the embedded value into a string. Note that we can safely use `v` in a context where expressions of type `Real` are expected because the type checker guarantees that the embedded type is of type `Real`. In the last line, we match that the unknown should have type `<Real>`. If none of the cases matches, a runtime error will be generated.

9.2.2 Analyzing Systems of Equations

We are primarily interested in mathematical models consisting of systems of equations. Let us consider the following classic example:

```
let LotkaVolterra =
  let growthRateRabbits = 0.04 in
  let deathRateRabbits = 0.0005 in
  let deathRateFoxes = 0.09 in
  let efficiencyGrowthFoxes = 0.1 in
  let rabbits:Population in
  let foxes:Population in
  Init rabbits 700.;
  Init foxes 10.;
  der(rabbits) = growthRateRabbits *. rabbits -.
                 deathRateRabbits *. rabbits *. foxes;
  der(foxes)   = efficiencyGrowthFoxes *. deathRateRabbits *.
                 rabbits *. foxes -. deathRateFoxes *. foxes
```

This is a simple form of biological model modeling the population dynamics of an ecological system. The model, called Lotka-Volterra after its inventors, models a predator-prey relationship, where the predators are foxes and the prey animals are rabbits. The model given in this example is a translation of the model described by Fritzson [51]. At the top of the model, a list of constants is given. The last two `let`-expressions define two unknowns `rabbits` and `foxes`, both of type `Population`, which is just a type alias

```
type Population = <Real>
```

for making the model more readable.

At the bottom of the model, we have two differential equations. Both of the variables `rabbit` and `foxes` appear differentiated, and the differentiated variables both appear

on the left hand side of the equations. Hence, we have an explicit ODE in state-space form. To solve the initial value problem, we need to define initial values for the state variables. This is expressed with the two `Init` declarations, where for example the variable `rabbit` is given the initial value `700.`.

The MKL language in itself does not know about the concepts of derivatives, initial values, and equations. These are all defined in a program or model. The following definitions are a selection from the definitions given in the MKL standard library file `modeling.mkl` (See Appendix D.2 for a full listing). The type definition

```
type Eqs
```

defines a new *pseudo* type called `Eqs`. We call it a pseudo type because there is nothing on the right hand side of the type definition, i.e., there are no constructor or alias defined for pseudo types. It is only used for type checking models. Because models have model type, we need to define a model type of `Eqs`.

```
type Equations = <Eqs>
```

The intuition is that `Equations` is the type of system of equations (one or more equations). This is also the standard return type from a model abstraction, i.e., the result of a model instantiation is an equation system (compare the electrical and mechanical models described in Chapter 8).

The next four lines lists the constructs for defining an equation system.

```
let Eq : <Real -> Real -> Eqs>
let EquationSysNode : <Eqs -> Eqs -> Eqs>
let (=) : <Real -> Real -> Eqs> = Eq
let (;) : <Eqs -> Eqs -> Eqs> = EquationSysNode
```

The first definition defines an unknown called `Eq` which is of function model type. The function takes two arguments of type `Real` as input and returns an equation system. The second unknown is a composition operator for the equation system. An equation system will internally be stored as a tree of equations, using the `EquationSysNode` as the nodes in the tree. The leaves will be the equations. Note how the types define the relationships.

The last two lines define prefix and infix operators for the equation = and composition `;` operators for creating systems of equations. These infix operators are also what is used when defining models.

The derivative operator `der` is defined as follows:

```
let der : <Real -> Real>
```

i.e., it is an unknown function taking a `Real` as input and return a `Real`. Sometimes it is convenient to use the independent variable `time` explicitly in the model. For example, in the previous chapter, we defined model `SineVoltage`, which uses the `time` variable. Time is also an unknown and defined as follows:

```
let time : <Real>
```

Finally, we define how a model can be given initial values:

```
let Init : <Real -> Real -> Eqs>
let InitGuess : <Real -> Real -> Eqs>
```

The first `Init` constructor is intended to define an initial condition, where the user states that it must be initialized to this value (i.e., `fixed = true` in Modelica). The second constructor `InitGuess` should be used when the initial value is only a guess value, i.e., as starting point for the solver to search for a consistent initial value.

The first thing to notice about all these definitions is that they define new unknowns. Hence, the concept of unknown has a very broad use in MKL. It is used both for modeling unknowns in the ordinary mathematical sense in equations, but also used as a constructor for expressions. Note also that the actual semantics of the meaning of these definitions are not defined here, but is defined by the analysis functions that inspects the model.

Now, let us analyze the model. Our first example task is to count the number of equations and the number of unknowns that an instance of a model contains.

Starting with counting the number of unknowns, we need to count the number of unique unknowns, not the number of times unknowns appear in equations. We create a type alias for the set of unknowns:

```
type UkSet = (Set <Real>)
```

Our task is now to define a function which takes a system of equations of type `Equations` as input and return a value of type `UkSet`:

```
let unknowns eqs:Equations -> UkSet =
  let get e:<> -> acc:UkSet -> UkSet =
     match e with
     | e1 e2 -> get e2 (get e1 acc)
     | 'time -> acc
     | uk:<Real> -> Set.add e acc
     | _ -> acc
  in get eqs (Set.empty)
```

The function `unknowns` traverses the model using model application patterns. Each time it finds an unknown, the matching value `e`, which in this case is the unknown of type `<Real>` is added to the set. Because it is a set, there are no duplications of elements. There are two new observations to be made in the example.

The first one is the pattern case with `'time`. The pattern syntax `'`e is a syntactic sugar for matching against an expression. Hence, the line with `'time` is equivalent to the following line

```
     | e when e == time -> acc
```

The rationale for having the `time` case is that `time` is also defined to be an unknown of type `<Real>`. Because we do not want to include the independent variable `time` in the set of unknowns, it is excluded by the match case.

The second observation in the example is that the first parameter `e` of the `get` function has type `<>`. Recall that this was defined to be the *any* model type, which has the meaning that it is type consistent with all other model types. For example, the `get` function, can be applied to any argument, as long as it is a model type. This introduces dynamic types of models within the statically typed language. The approach is inspired by Abadi *et. al.*'s [1] work on dynamic typing in statically typed languages, as well as the work on gradual typing by Siek and Taha [132, 133]. The intuition of the type consistency relation is that a type <> is type consistent with <> or <T>, for any type T. A type <T_1> is type

consistent with another type <T_2> iff T_1 and T_2 are type consistent. A formalization of this type consistency relation will be given in Chapter 10.

The number of unknowns can then be computed by taking the size of the unknown set:

```
let noUnknowns eqs:Equations -> Int =
  Set.size (unknowns eqs)
```

The computation of the number of equations is more direct:

```
let noEquations eqs:Equations -> Int =
  match eqs with
  | e1 ; e2 -> (noEquations e1) + (noEquations e2)
  | e1 = e2 -> 1
  | _ -> 0
```

In this function we are not traversing any model application, we are only moving down in the tree of equations. By also matching on the equation using infix notation, we exclude all other relations that have type `<Eqs>`, e.g., `Init`-equations.

By supplying the `LotkaVolterra` model as argument to both of the `noUnknowns` and `noEquations` functions we get the result 2 for both the number of unknowns and the number of equations.

Are these functions for calculating the number of unknowns and the number of equations only valid for such small example models as `LotkaVolterra`? No, they can actually handle arbitrary models, as long as the model is defined using the same primitives for equations etc. By calling these functions on the elaborated variant of the `MechSys` model described in the previous chapter, we get the result of 1586 equations and 1586 unknowns, which are the expected results.

The second and last example in this section for illustrating the idea of intensional analysis of models is to define a function that extracts a mapping between unknowns and their defined initial values. We start by defining a type alias for an initial value mapping between unknowns and initial values:

```
type InitValMap = (<Real> => Real)
```

The function for extracting the initial values are defined as follows:

```
let initValues eqs:Equations -> InitValMap =
  let get eqs:Equations -> acc:InitValMap -> InitValMap =
    match eqs with
    | e1 ; e2 -> get e2 (get e1 acc)
    | Init x (val v:Real) -> Map.add x v acc
    | _ -> acc
  in get eqs (Map.empty)
```

The local function `get` takes as its second argument an accumulator map, which is a mapping that will accumulate a binding each time the `Init` expression is matched. The function definition is straightforward and should be self describing. Also note that we are using a nested pattern for the `Init` case, from which the embedded value `v` of the model value is extracted.

9.3 Chapter Summary and Conclusions

In this chapter we gave a short introduction to the ideas of intensional analysis of models in MKL. We will in both Chapter 11 and 12 give larger examples for showing the approach of intensional analysis of models.

We showed in a few examples that the language is expressive for traversing the model structure. This expressiveness is mainly due to the flexibility of having dynamic types for models, i.e., that a type of a model does not have to be specific, it could be `<>`.

Regarding language safety, the model type concept in MKL requires that types must be used when modeling. Hence, the MKL interpreter can type check and report errors in models, even if the constructs are user defined. This enables early and explicit feedback of modeling errors. For example, if the user tries to insert an equation inside an expression, e.g. `(3. = 2.) +. 4.` the type checker will report an error. Hence, at the modeling level, the type requirements are strict. However, for analyzing the model, we have favored expressiveness.

10

Semantics of MKL

We have in previous chapters informally described the syntax and semantics of MKL. In this chapter we formalize both the static and the dynamic semantics for a core of the MKL language.

We present three different languages. The language $\lambda^{<>}$ is the source language corresponding to a core of MKL. An expression in $\lambda^{<>}$ can be lifted to an intermediate language $\lambda_L^{<>}$, meaning that selected expressions are lifted into models. The reason for model lifting is, as explained in the previous chapter, to create data structures of models that can later be inspected and analyzed. The language $\lambda_L^{<>}$ can be used for evaluation, but is not technically sufficient for proving type safety. Hence, we define a second intermediate language $\lambda_{LC}^{<>}$ where type casts are inserted. We prove soundness of the translation between the intermediate languages, the usual progress and preservation lemmas for $\lambda_{LC}^{<>}$, and finally type safety for $\lambda^{<>}$. The chapter is structured as follows:

- We describe the abstract syntax for $\lambda^{<>}$ and $\lambda_L^{<>}$ respectively (Section 10.1).
- We state the type system for $\lambda^{<>}$, rules for lifting expressions to models, as well as a type consistency relation used for comparing types. (Section 10.2).
- We prove soundness of model lifting and cast insertion, as well as giving a type system and operational dynamic small-step semantics for $\lambda_{LC}^{<>}$. (Section 10.3).
- We prove type safety for $\lambda^{<>}$ (Section 10.4).
- Finally, we discuss different extensions to the core language $\lambda^{<>}$ (Section 10.5).

10.1 Syntax

Consider the abstract syntax for $\lambda^{<>}$ that is summarized in Figure 10.1. The meta-variables x and y range over $\mathbb{X}$, a countable set of names. The meta-variable e ranges

Variables	$x, y \in \mathbb{X}$		
Unknowns	$u \in \mathbb{U}$		
Constants	$c \in \mathbb{C}$		
Expressions	e	$::=$	$x \mid \lambda x\!:\!\tau.e \mid e\ e \mid c \mid$
			$u\!:\!\tau \mid \nu(\tau) \mid e\,@\,e \mid \texttt{val}\ e\!:\!\tau \mid \texttt{decon}(e, d, e, e)$
Deconstruct patterns	d	$::=$	$\texttt{uk}\!:\!\tau \mid x\,@\,x \mid \texttt{val}\ x\!:\!\tau$
Values	v	$::=$	$\lambda x\!:\!\tau.e \mid c \mid u\!:\!\tau \mid v\,@\,v \mid \texttt{val}\ v\!:\!\tau$
Ground Types	$\gamma \in \mathbb{G}$		
Types	τ	$::=$	$\gamma \mid \tau \rightarrow \tau \mid \texttt{<}\tau\texttt{>} \mid \texttt{<>}$

Figure 10.1: *Abstract syntax for $\lambda^{\texttt{<>}}$ and $\lambda_L^{\texttt{<>}}$*

over the set of expressions $Expr$ and τ ranges over the set of types $Types$. We use subscripts for denoting different expressions or types, e.g., e_1 and e_2 represent two different expressions.

The first four expressions are standard. The expression x is a free variable and lambda abstraction $\lambda x\!:\!\tau.e$ binds variable x of type τ in e. We use the lambda notation here, which has the same meaning as MKL's concrete syntax for anonymous functions that is defined with the `fun` keyword. The expression $e_1\ e_2$ is application and $c \in \mathbb{C}$ a constant. The set of constants $\mathbb{C}$ is the union of the set of boolean values $\{\texttt{true}, \texttt{false}\}$, the set of integers, the set of reals (represented as floating-point values), the set of strings, and the set of primitive functions.

The next five expressions are new for $\lambda^{\texttt{<>}}$. The aim of the language is to be able to express mathematical models and to inspect a models' structure. Hence, the concept of *unknowns* is central. The first expression $u : \tau$ is an unknown u tagged with type τ. The set $\mathbb{U}$ is defined as the set of all unknowns. Unlike e.g., logic variables in logic programming languages, unknowns in MKL cannot be bound to values. Instead, they are used to build up the data structure of a model that can be deconstructed at a later stage.

To simplify $\lambda^{\texttt{<>}}$ compared to MKL, we define an expression $\nu(\tau)$ (pronounced "new"), which creates an unknown expression $u : \texttt{<}\tau\texttt{>}$ when evaluated. In MKL, unknowns are constructed using `let`-expressions of the form $\texttt{let}\ x\!:\!T\ \texttt{in}\ e$. Hence, we define the following derived form:

$$\texttt{let}\ x\!:\!\texttt{<}\tau\texttt{>}\ \texttt{in}\ e \equiv (\lambda x\!:\!\texttt{<}\tau\texttt{>}.e)\nu(\tau) \tag{10.1}$$

The expression $e\,@\,e$ is a model application, which is typed as a function, but is never applied. From a runtime perspective a model application can be seen as a tuple with two elements.

The expression $\texttt{val}\ e\!:\!\tau$ is a model value that embeds the expression e of type τ. We call this a model *value* because e must be closed and is evaluated to a value before it can be extracted from $\texttt{val}\ e\!:\!\tau$.

The expression $\texttt{decon}(e_1, d, e_2, e_3)$ is a deconstructor of models. The value after evaluating e_1 is the value to be deconstructed and matched against the deconstruction pattern d. We choose the letter d for patterns instead of p to avoid confusion with patterns

in `match` expressions. The expression e_2 is returned on a successful match and e_3 on a unsuccessful match.

Deconstructor patterns can have the following shapes: $\texttt{uk}:\tau$ for unknowns, $x \,@\, x$ for matching model application, and $\texttt{val}\ x:\tau$ for matching model values. The variable x is a pattern variable and τ a type tag[1]

The meta-variable v ranges over the set of *Values*, where $\mathit{Values} \subseteq \mathit{Expr}$[2]. Lambda abstractions, constants, and unknowns are always values. The language is using weak reduction with a call-by-value evaluation strategy and therefore variables are not values. A model application $v \,@\, v$ is a value if its sub-expressions are values. A model value expression $\texttt{val}\ v:\tau$ is also defined to be a value, if its sub-expression is a value.

10.2 Type System and Model Lifting

In this section we describe the type system for $\lambda^{\texttt{<>}}$ and how expressions are lifted to models, by translation to $\lambda_L^{\texttt{<>}}$. We start by considering types and how they are compared.

Recall the syntax definition of types in Figure 10.1. There are two standard types and two new types for this language. The meta-variable γ ranges over all ground types $\mathbb{G}$, which includes boolean, integer type etc. The type $\tau \to \tau$ is the standard function (arrow) type.

The two new types $\texttt{<}\tau\texttt{>}$ and $\texttt{<>}$ are both *model types*. The former is the *specific model type*, stating that this model is of type τ. The latter is called the *any model type* because a value with that type can be any model. The intuition of the type system is that it introduces dynamic typing *for models* in a statically typed functional language. To deconstruct a model, the explicit deconstruct expression `decon` must be used.

A first attempt of defining a type system for model types would be to introduce a subtyping relation between specific model types and any model types. In such a case, type $\texttt{<>}$ represents the top model type. The subtyping rules for such a system could be as follows:

Subtyping rules $\boxed{\tau <: \tau'}$

$$\frac{}{\gamma <: \gamma} \qquad \frac{}{\texttt{<}\tau\texttt{>} <: \texttt{<>}} \qquad \frac{\tau_1 <: \tau_2}{\texttt{<}\tau_1\texttt{>} <: \texttt{<}\tau_2\texttt{>}} \qquad \frac{\tau_3 <: \tau_1 \quad \tau_2 <: \tau_4}{\tau_1 \to \tau_2 <: \tau_3 \to \tau_4}$$

A type system would then include the usual rule of subsumption:

$$\frac{\Gamma \vdash e:\tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e:\tau_2}$$

[1]Note that deconstructor patterns are not nested for the reason of making the core language as simple as possible. However, `match`-expressions in full MKL can contain nested patterns. These `match`-expressions are then translated during pattern compilation to primitives for deconstructing values, where `decon` is used for deconstructing models.

[2]In this chapter, we only define an operational semantics for $\lambda_{LC}^{\texttt{<>}}$. The languages $\lambda^{\texttt{<>}}$ and $\lambda_L^{\texttt{<>}}$ are only used as translation steps for proving type safety. Consequently, values are strictly not necessary to define at this stage. However, we define a big-step semantics for $\lambda_L^{\texttt{<>}}$ in Appendix C that motivates the inclusion of values.

Now, assume we have a function $f : \texttt{<Int>} \rightarrow \texttt{<Int>}$ and a constant $c : \texttt{<Int>}$. Then, an expression

$$(\lambda x\!:\!\texttt{<>}.x)c$$

has type <> in such a subtyping system. However, an expression

$$(\lambda x\!:\!\texttt{<>}.f x)c$$

is not typeable because x has type <> which is a supertype and not a subtype of <Int>, which is the function f's argument's type. We argue that this typesystem is too restrictive and does not give the flexibility of dynamic typing of models. Instead, we choose to use a more flexible approach with *type consistency*.

10.2.1 Type Consistency

We adopt the idea of replacing type equality in type checking rules with the type consistency relation $\sim$. This idea was pioneered by Siek and Taha [132, 133] in the context of gradual typing. The intuition of the consistency relation in our language is that ground types shall be compared with equality, while the type <> is consistent with both $\texttt{<}\tau\texttt{>}$ and <> for some type τ. For example:

$$\texttt{<Int>} \sim \texttt{<>} \qquad \texttt{<Int->Real>} \sim \texttt{<Int->Real>}$$

$$\texttt{<>} \sim \texttt{<Bool->Real>} \qquad \texttt{<Real>->Real} \sim \texttt{<>->Real}$$

$$\texttt{Int} \not\sim \texttt{<>} \qquad \texttt{Real-><>} \not\sim \texttt{<>->Real} \qquad \texttt{Real} \not\sim \texttt{Bool}$$

Note the difference from gradual typing [133] that the type <> is only consistent with other *model types*, not with any other type.

Before we define type consistency, we define a *restriction* operator as follows:

Restriction $\boxed{\tau \parallel_\tau}$

$$\begin{array}{lcll}
\texttt{<}\tau\texttt{>} \parallel_{\texttt{<>}} & = & \texttt{<>} & \\
\texttt{<}\tau_1\texttt{>} \parallel_{\texttt{<}\tau_2\texttt{>}} & = & \texttt{<}\tau_1 \parallel_{\tau_2}\texttt{>} & \\
\tau_{11} \rightarrow \tau_{12} \parallel_{\tau_{21}\rightarrow\tau_{22}} & = & \tau_{11} \parallel_{\tau_{21}} \rightarrow \tau_{12} \parallel_{\tau_{22}} & \\
\tau_1 \parallel_{\tau_2} & = & \tau_1 & \text{if } \tau_1 \neq \texttt{<}\tau_3\texttt{>} \text{ and } \tau_1 \neq \tau_{11} \rightarrow \tau_{12} \\
 & & & \text{for some } \tau_3, \tau_{11}, \text{and } \tau_{12}
\end{array}$$

We can now define type consistency:

Definition 10.2.1 (Type Consistency).
Two types τ_1 and τ_2 are consistent, written $\tau_1 \sim \tau_2$ iff $\tau_1 \parallel_{\tau_2} = \tau_2 \parallel_{\tau_1}$.

Proposition 10.1 (Properties of $\sim$)
The restriction operator has the following basic properties:

(a) $\sim$ is reflexive.

(b) $\sim$ *is symmetric.*

(c) $\sim$ *is not transitive. For example*
$\texttt{<Int>} \sim \texttt{<>}$ *and* $\texttt{<>} \sim \texttt{<Real>}$ *but* $\texttt{<Int>} \not\sim \texttt{<Real>}$.

(d) $\tau_1 \sim \tau_2$ *iff* $\texttt{<}\tau_1\texttt{>} \sim \texttt{<}\tau_2\texttt{>}$.

(e) If $\texttt{<>} \sim \tau$ *then* $\tau = \texttt{<>}$ *or* $\exists \tau'.\tau = \texttt{<}\tau'\texttt{>}$.

(f) If $\texttt{<>} \sim \tau$ *and* $\tau \sim \tau'$ *then* $\texttt{<>} \sim \tau'$.

(g) If $\tau \sim \texttt{<}\tau'\texttt{>}$ *then* $\tau \sim \texttt{<>}$.

Note that property (e) is a compact way to test if a type is a model type (either a specific or an any model type).

10.2.2 Type System

Before we explain the type system for $\lambda^{<>}$ we define the typing environment as follows:

Definition 10.2.2 (Typing Environment). The typing environment is a partial function $\Gamma : \mathbb{X} \to Types$, where the domain is the set of variable names and the co-domain the set of types.

Syntactically, the typing environment will also be handled with set notations, e.g., $x : \tau \in \Gamma$ is equivalent to $\Gamma(x) = \tau$. However, Definition 10.2.2 states that variable names in the environment are always distinct.

We use the notation $\Gamma, x{:}\tau$ to extend environment Γ with a new binding $x{:}\tau$. If a binding of x exists in Γ, the new binding replaces the old one. We define the domain of a typing relation as follows:

Definition 10.2.3. $dom(\Gamma) \equiv \{x \mid x : \tau \in \Gamma\}$

We also define the subset relation between typing environments:

Definition 10.2.4. $\Gamma \subseteq \Gamma' \equiv \forall x \tau.\Gamma(x) = \tau \; implies \; \Gamma'(x) = \tau$

The type system for $\lambda^{<>}$ is defined by a four-place *model lifting relation*

$$\Gamma \vdash_L e \rightsquigarrow e' : \tau$$

where e is an expression in $\lambda^{<>}$, e' an expression in $\lambda_L^{<>}$, τ the resulting type, and Γ the typing environment. The model lifting relation is inductively defined using a set of inference rules given in Figure 10.2 on page 154[3].

Definition 10.2.5 (Well typed expression in $\lambda^{<>}$). An expression e of language $\lambda^{<>}$ is well typed (typable) in typing environment Γ if there exits e' and τ, such that $\Gamma \vdash_L e \rightsquigarrow e' : \tau$.

[3] To make it easier to compare the different intermediate languages' semantics, we list all the rules at the end of the chapter.

Language $\lambda^{\texttt{<>}}$ is a explicitly typed language and the model lifting can therefore be performed in a direct bottom up manner. Input to such a function would be an empty typing environment and expression e_1 and the output expression e_2 whose type is τ.

We now give an overview of the translation rules for the model lifting relation. We first consider the rules that are not lifting any expression, i.e., where the type of the expression is not changed during translation.

The rules (L-VAR) for variables and (L-ABS) for lambda abstractions are standard and similar to the simply-typed lambda calculus.

The rule (L-CONST) assumes a function $\Delta : \mathbb{C} \rightarrow Types$ that applied to a constant returns the constant's type. We assume that the Δ function cannot return a model type and therefore give the following assumption:

Assumption A1 (Δ-types).
If $\Delta(c) = \tau$ then $\tau \in \mathbb{G}$ or there exists τ_1 and τ_2 such that $\tau = \tau_1 \rightarrow \tau_2$.

Now, consider the following examples:

$$(u\texttt{:<Int} \rightarrow \texttt{Int>}) : \texttt{<Int} \rightarrow \texttt{Int>} \tag{10.2}$$

$$(\texttt{val}\ 5\texttt{:Int}) : \texttt{<Int>} \tag{10.3}$$

$$(u\texttt{:<Int} \rightarrow \texttt{Int>})@(\texttt{val}\ 5\texttt{:Int}) : \texttt{<Int>} \tag{10.4}$$

$$(\lambda x\texttt{:<>}.x)\ (\texttt{val}\ 5\texttt{:Int}) : \texttt{<>} \tag{10.5}$$

$$(\lambda x\texttt{:<Int>}.x)\ (\texttt{val}\ 5\texttt{:Int}) : \texttt{<Int>} \tag{10.6}$$

$$(u\texttt{:<>})@(\texttt{val}\ 5\texttt{:Int}) : \texttt{<>} \tag{10.7}$$

$$\nu(\texttt{Int}) : \texttt{<Int>} \tag{10.8}$$

$$\tag{10.9}$$

In the first example, we see that an unknown tagged with a specific model type is of the same type as the type tag (type rule (L-UK)). Also, the premise $\texttt{<>} \sim \tau_1$ of (L-UK) makes sure that unknowns are always tagged with only model types, i.e., unknown expressions are always models.

Example (10.3) shows the use of type rule (L-VAL). The type of the embedded value 5 has type Int and the val expression has type <Int>. The intuition is that a model value expressions embed arbitrary expression e with type τ with the result that $\texttt{val}\ e\texttt{:}\tau$ is of type $\texttt{<}\tau\texttt{>}$. We say that expression e is "lifted" to be of model type. The model value is similar to values of type Dynamic developed by Abadi *et. al.* [1]. That work inspired our design, but there are several differences, including the procedure for type checking.

In example (10.4), a model application applies an unknown to a model value. We can see that the resulting model type is specific, and follows normal conventions of type checking of applications, with the exception that all expressions are models. The example demonstrates how the rule (L-MODAPP2) is used for deriving the type of the expression. Note also that expression (10.4) is a value, i.e., the model application will never actually be applied because there is no beta reduction for model applications.

So far all model types have been specific. However, in example (10.5), we see an ordinary function application, where the lambda binding variable of the lambda expression is given any model type <>. Even though the argument has specific model type <Int>,

the resulting type of the expression is <>. However, if the lambda's binding variable has a specific model type (example (10.6)), the specific type is preserved. In both cases, rule

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e_1' : \tau_{11} \to \tau_{12} \quad \Gamma \vdash_L e_2 \rightsquigarrow e_2' : \tau_2 \quad \tau_{11} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau_{12}} \text{ (L-APP)}$$

was used. Note how the type consistency premise $\tau_{11} \sim \tau_2$ is used instead of type equivalence between the type of the argument and the type of the abstraction's formal parameter.

In the example (10.7), we see how rule (L-MODAPP1) is used when the type of the left hand side's expression is <>. Finally, in (10.8) we see how the $\nu(\texttt{Int})$ constructor of unknowns has a type Int and creates unknown expressions of model type <Int>. That is, when unknowns are created, they must be of a specific model type, which conform to the presence of premise $\texttt{<>} \sim \tau_1$ of rule (L-UK).

The last three type rules are applicable for type checking a decon expression, i.e., the expression for deconstructing models.

The rule (L-DECON-UK) checks using premise $\texttt{<>} \sim \tau_1$ that the type of e_1 is a model. Premise $\tau_2 \sim \tau_3$ makes sure that the two alternative return expressions e_2 and e_3 are consistent with each other. The last premise $\texttt{<>} \sim \tau_4$ checks that the type of the pattern tag type is a model. The restriction operator $\tau_2 \parallel_{\tau_3}$ is used to define the result type. Note that the restriction operator itself is not symmetric, but because of premise $\tau_2 \sim \tau_3$ we have directly from the definition of $\sim$ that $\tau_2 \parallel_{\tau_3} = \tau_3 \parallel_{\tau_2}$.

The rule (L-DECON-APP) is applicable when the deconstructor pattern is a model application. Note that bindings for x_1 and x_2 are added to the typing environment when deriving the type for e_2. Note also that the type that x_1 and x_2 are bound to is <> because the deconstruction only knows that it was a model application that was deconstructed. We know nothing about the specific types of the sub-expressions, just that they must be of model type.

The last rule (L-DECON-VAL) is applicable when the deconstructor pattern is a model value. The pattern's tagged type τ_4 is bound to x in the typing environment when deriving the result for e_2. Note that τ_4 is not checked for being a model type because the embedded expression in a model value can be any type (typically not a model type).

10.2.3 Model Lifting

We saw in example (10.5) that a function can be applied to a value of type <Int>. The function's formal parameter can be of both the specific model type <Int> or of the any model type <>. However, in the application case, if there is a mismatch between expressions of model type and expressions that are not models, there is an essential mechanism for lifting non-models into models.

The purpose of this term translation is to lift expressions to model types when needed. For example, consider an expression x + 3 where x is of a model type. During evaluation x can potentially be replaced by an unknown, which will make the computation fail. The purpose is to lift such terms, so that the + operator and the second operand 3 are lifted to become models. This is done by inserting val expressions. The actual model lifting is performed by the five rules (L-APPM1) to (L-APPM2)

The rule (L-APPM1) type checks an application expression. The rule is applicable when τ_2 is a model type and τ_{11} is not. This is checked by the two premises $<\tau_{11}> \sim \tau_2$ and $\tau_{11} \not\sim \tau_2$. By encapsulating τ_{11} into a model type, the consistency check verifies that τ_2 is either $<>$ or a specific model that is consistent with $<\tau_{11}>$. The second premise $\tau_{11} \not\sim \tau_2$ makes sure that the rule is exclusive with respect to rule (L-APP). If the premises hold, the resulting expression e'_1 is lifted using a `val`-expression. Note also that the application is translated to a model application.

The rules (L-APPM2) and (L-APPM3) consider the application cases when e'_1 is derived to be of type $<>$. The rule (L-APPM2) is applicable if τ_2 is a model type (either any model type or specific). In such a case, the application is transformed to a model application. The rule (L-APPM3) is applicable if τ_2 is not a model type. In such a case e'_2 is lifted to become a model. In both of these rules, the resulting type is $<>$.

The rules (L-APPM4) and (L-APPM5) are applicable when e'_2 is derived to be of type $<\tau_{11} \rightarrow \tau_{12}>$. (L-APP4) is applicable if τ_2 is a model type. If so, the resulting term is transformed to a model application. Note also that the consistency checking makes sure that the specific types are consistent, if τ_2 was a specific model type. Rule (L-APPM5) is applicable if $\tau_{11} \sim \tau_2$ and $<\tau_{11}> \not\sim \tau_2$. In such a case we lift e'_2.

10.3 Cast Insertion and Dynamic Semantics

For an interpreter or compiler implementation, the language $\lambda_L^{<>}$ is typically the language that should be used for execution. In our prototype implementation (explained in Chapter 12) an extended version of this core language is used for evaluation. The big-step semantics of $\lambda_L^{<>}$ for implementing such an interpreter is given in Appendix C. However, in this section we are interested in reasoning about properties of the language, and in particular to prove type safety of the language. However, proving type safety directly on $\lambda_L^{<>}$ is technically not feasible, since the consistency relation in e.g., the rule (L-APP) makes it not possible to prove Preservation (see Chapter 10.4 for the Preservation lemma). Instead, we define a *cast insertion relation*, which translates expressions in $\lambda_L^{<>}$ into a language called $\lambda_{LC}^{<>}$. In the latter language, the type consistency relation between types are replaced with equality, making the proof possible.

In contrast to $\lambda^{<>}$ and $\lambda_L^{<>}$, which have the same syntax, $\lambda_{LC}^{<>}$ is updated as follows:

Intermediate Language $\lambda_{LC}^{<>}$ $\quad$ $e \in \lambda_{LC}^{<>} \supset \lambda_L^{<>}$

Expressions	e	$+=$	$\langle\tau \Leftarrow \tau\rangle e$
Values of models	w	$::=$	$u\!:\!\tau \mid v \,@\, v \mid \texttt{val}\; v\!:\!\tau$
Values with casts	ξ	$::=$	$w \mid \langle\tau \Leftarrow \tau\rangle\xi$
Values	v	$::=$	$\lambda x\!:\!\tau.e \mid c \mid \xi$

One new expression $\langle\tau_2 \Leftarrow \tau_1\rangle e$ for casts is defined, where the expression e is cast from source type τ_1 to target type τ_2. The intuition is that expression e is of type τ_1 and the whole cast expression $\langle\tau_2 \Leftarrow \tau_1\rangle e$ is of type τ_2.

We define new syntax for values of different categories. Let the meta-variable w ranges over $\mathit{ModValues}$, i.e., values of model types. Moreover, we define a meta-variable

ξ that ranges over $CastValues$. This separation of values into different syntactic categories is necessary for making the language deterministic, i.e., that not more than one rule of the runtime semantics is applicable at the same time.

10.3.1 Cast Insertion

Cast insertion is defined by a four-place *cast insertion relation*

$$\Gamma \vdash_C e \rightsquigarrow e' : \tau$$

where e is an expression in $\lambda_L^{<>}$, e' an expression in $\lambda_{LC}^{<>}$, τ the resulting type, and Γ the typing environment. The cast insertion relation is inductively defined using a set of inference rules given in Figure 10.3 on page 154.

When there is a model lifting translation for an expression e, the expression e is well-typed with regards to a type system for $\lambda_L^{<>}$. Because we do not make use of a specific type system of $\lambda_L^{<>}$, we omit its definition and instead state the soundness of translation with regards to the cast insertion relation.

Lemma 10.1 (Model Lifting is Sound)
If $\Gamma \vdash_L e \rightsquigarrow e' : \tau$ *then there exists an* e'' *such that* $\Gamma \vdash_C e' \rightsquigarrow e'' : \tau$.

Proof: By induction on a derivation of $\Gamma \vdash_L e \rightsquigarrow e' : \tau$. All cases are straightforward using the definition of type consistency. □

Let us now define the type system for $\lambda_{LC}^{<>}$ by a three-place *typing relation*

$$\Gamma \vdash e : \tau$$

where e is an expression in $\lambda_{LC}^{<>}$, τ its type, and Γ the typing environment. The typing relation is inductively defined in Figure 10.3 on page 155.
The aim of performing the cast insertion is to make it possible to prove type safety of the language. There are three separate cases where we need to remove the consistency relation to be able to prove the preservation lemma.

The first case is the existence of $\tau_{11} \sim \tau_2$ in rule (L-APP). Trying to prove preservation of $\lambda^{<>}$ by induction on a derivation of $\Gamma \vdash_L e \rightsquigarrow e' : \tau$ will fail on the (L-APP) case. Hence, the trick in the cast insertion is shown in the conclusion of rule:

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \tau_{11} \sim \tau_2}{\Gamma \vdash_C e_1\ e_2 \rightsquigarrow e_1'\ (\langle \tau_{11} \Leftarrow \tau_2 \rangle e_2') : \tau_{12}} \ \text{(C-APP)}$$

By casting e_2' from τ_2 to τ_{11}, the expression $\langle \tau_{11} \Leftarrow \tau_2 \rangle e_2'$ has type τ_{11}, eliminating the need for the premise $\tau_{11} \sim \tau_2$.

The second case where casts are needed is for the rules (L-DECON-UK), (L-DECON-APP), and (L-DECON-VAL). All these three rules are treated in the same way in this case. Therefore we will concentrate on the (L-DECON-UK) rule. The objective is to remove the premise $\tau_2 \sim \tau_3$ and instead have $\tau_2 = \tau_3$. Consider rule (T-DECON-UK) were both e_2 and e_3 derive the same type τ_2. Because $\tau_2 \sim \tau_3$ in (L-DECON-UK), we can in (C-DECON-UK) cast both τ_2 and τ_3 to τ_5 because $\tau_5 = \tau_2 \parallel_{\tau_3}$.

The third case involving casts is the result of the problem that model applications can be of both specific model types $<\tau>$ and $<>$ for some τ. If both model types exist, we cannot prove progress for the (T-DECON-APP) case, because we cannot derive the specific types for the sub-values of e_1 in (T-DECON-APP). The solution we use is to assume that the types of the sub-expressions of a model application are always $<>$. In such a case, we must have an inversion lemma, stating that the sub-expressions are of type $<>$. Note that we therefore have only one rule for model application in $\lambda_{LC}^{<>}$: (T-MODAPP). One problem with proving soundness of cast insertion is that the type τ must be the same. Note that the resulting type of (C-MODAPP2) is $<\tau_{12}>$ and not $<>$. The trick in (C-MODAPP2) is that we first cast each sub-expression of the model application to $<>$, and then cast the whole model application to $<\tau_{12}>$.

Finally, we prove the soundness of cast insertion:

Lemma 10.2 (Cast Insertion is Sound)
If $\Gamma \vdash_C e \rightsquigarrow e' : \tau$ *then* $\Gamma \vdash e' : \tau$.

Proof: By induction on a derivation of $\Gamma \vdash_C e \rightsquigarrow e' : \tau$. The proof is straightforward, where the case (C-MODAPP2) uses Proposition 10.1 (g). □

10.3.2 Dynamic Semantics

We define the dynamic semantics of $\lambda_{LC}^{<>}$ using operational semantics with *small-step* style, pioneered by Plotkin [126]. The shape of the single-step relation is

$$e \mid U \longrightarrow e' \mid U'$$

where expression e is reduced to e' in one step, and U and U' are starting and ending states for the store of unknowns. The meta-variable $U \subseteq \mathbb{U}$ ranges over a (potentially empty) set of unknowns. Hence, the operational semantics includes *computational effects* in terms of new unknowns that are created during evaluation. However, the unknowns cannot be assigned values and could be considered as symbols that can only be compared using equality.

Consider now the small-step semantics, defined as a set of inference rules in Figure 10.5 on page 157. The first eight rules of relation $e \mid U \longrightarrow e' \mid U'$ are *computation rules* that reduce an expression one step. The next seven rules are *congruence rules* which determine the used evaluation strategy; in this case call-by-value.

Application

The rule (E-APPABS) is the standard application rule (β-reduction). The value v_1 is substituted for x in e_1. The notation $[x \mapsto v_1]e_1$ stands for standard capture-avoiding substitution, where v_1 is substituted for all x that appear free in e_1. For completeness the definitions of substitution and the free variable function $FV(e)$ are given in Figure 10.6 on page 158.

This leads us to the following convention:

Convention 2. All expressions that differ only with respect to names of binding variables are exchangeable in all contexts.

Hence, for example, expressions $\lambda x{:}\tau.x$ and $\lambda y{:}\tau.y$ are exchangeable in all contexts. Our language only evaluates *closed* expressions meaning that the substituted expression will never contain free variables and therefore renaming is not needed to avoid capturing.

The second rule for application is (E-DELTA) where e_1 is evaluated to a constant c instead of a lambda. We use the standard notation of a δ function for abstracting the computation of built-in operators, i.e., $\delta(c, v)$ returns the result of applying c to v.

The two congruence rules (E-APP1) and (E-APP2) are used for evaluating application expressions. For an expression $e_1\ e_2$, the rule (E-APP1) is first used for evaluating e_1 to a value. Then, the rule (E-APP2) evaluates e_2 to a value. Finally, either rule (E-APPABS) or (E-DELTA) apply.

Unknowns, Model Applications, and Model Values

There are three kinds of model expressions: unknowns $u:\tau$, model values $\texttt{val}\ v:\tau$, and model application values $v_1 @ v_1$.

The computation rule (E-NEWUK) creates new unknowns when evaluated. The premise $u \notin U$ means that we pick a fresh unknown u that is not in the set U. The returned state is augmented with the new unknown. Note that the resulting unknown expression $u:\texttt{<}\tau_1\texttt{>}$ is "tagged" with the type $\texttt{<}\tau_1\texttt{>}$ from the ν-expression.

Using the unknown binder for creating new unknowns intentionally introduces the side effect that two expressions containing `let` unknown binders do not evaluated to the same value. This effect is by design, for example recall the `Circuit` model, where two `Resistor` components are created. The `Resistor` model is a function that contains let unknown binders for e.g., the current through the component. These unknowns must be distinct, which is performed by the effect in the rule (E-NEWUK).

Unknowns can be supplied as arguments to functions. However, because unknowns are symbols that cannot be bound to a value, there is no application rule $e_1\ e_2$ that evaluates e_1 to unknown u and then performs the application. Instead a *model application* expression $e_1 @ e_2$ is used to create a data structure. A model application expression can from a untyped point of view be seen as a tuple holding two expressions, which are by rules (E-MODAPP1) and (E-MODAPP2) evaluated to a value.

The last kind of model expression is *model values*, written $\texttt{val}\ e_1:\tau$. A model value embeds an expression e_1 and stores the type τ of e_1. Its evaluation rule (E-MODVAL) evaluates e_1 until it becomes a value.

Deconstructing Models

The previous subsection describes how model expressions are created and evaluated to values. The last two computation rules (E-DECON-T) and (E-DECON-F) are used for *deconstructing* models, i.e., unknowns $u:\tau$, model values $\texttt{val}\ v:\tau$, or model application values $v_1 @ v_1$.

Consider now the expression

$$(u:\texttt{<Int} \to \texttt{Int>}) @ (\texttt{val}\ 5:\texttt{Int}) \qquad (10.10)$$

which has been evaluated to a value. Because u is an unknown of function type, the expression cannot be evaluated further. However, it can be deconstructed using the expression $\texttt{decon}(v, d, e_2, e_3)$, which can be read as "match value v with pattern d. If it

matches, evaluate and return e_2 after substitution of values for pattern variables in d. If it does not match, evaluate and return the value for e_3".

Now, consider the congruence rule (E-DECON) in Figure 10.5. The rule evaluates the first expression e_1. When e_1 is a value, either reduction rule (E-DECON-T) or (E-DECON-F) apply. The value v_1, the deconstructor pattern d, and the expression e_2 are in both rules given to the *match* relation. If the match is true, the rule (E-DECON-T) applies and e_2' is returned. If the match is false, the rule (E-DECON-F) applies and e_3 is returned. Note that the axioms (M-UK), (M-MAPP), and (M-MVAL) of the match relation checks the shape of the expression (that it is an unknown, model application, or a model value). Moreover, for (M-UK) and (M-MVAL) it is also checked that the type tag τ_1 is equal to the type tag of the pattern.

Let $e_{example}$ denote the example expression given in (10.10). The following examples show the basic idea of the model deconstructor.

$$
\begin{aligned}
\texttt{decon}(e_{example}, x\,@\,y, x, \texttt{val}\ 1.1\texttt{:Real}) \mid U &\longrightarrow u\texttt{:<Int} \rightarrow \texttt{Int>} \mid U \\
\texttt{decon}(e_{example}, \texttt{uk:Real}, x, \texttt{val}\ 1.1\texttt{:Real}) \mid U &\longrightarrow \texttt{val}\ 1.1\texttt{:Real} \mid U \\
\texttt{decon}(e_{example}, x\,@\,y, y, \texttt{val}\ 1.1\texttt{:Real}) \mid U &\longrightarrow \texttt{val}\ 5\texttt{:Int} \mid U \\
\texttt{decon}(\texttt{val}\ 5\texttt{:Int}, \texttt{val}\ x\texttt{:Int}, x, 20) \mid U &\longrightarrow 5 \mid U \\
\texttt{decon}(u\texttt{:<Int} \rightarrow \texttt{Int>}, \texttt{uk:<Int} \rightarrow \texttt{Int>}, 1, 2) \mid U &\longrightarrow 1 \mid U \\
\texttt{decon}(u\texttt{:<Int} \rightarrow \texttt{Int>}, \texttt{uk:<Real>}, 1, 2) \mid U &\longrightarrow 2 \mid U
\end{aligned}
$$

10.3.3 Casts

The previous rules describe the fundamental semantics of $\lambda^{<>}_{LC}$. There are also three computation rules (E-CAST-ARROW), (E-CAST-GAMMA), and (E-CAST-MODEL), as well as one congruence rule (E-CAST) that handle casts. The casts semantics shall not be seen as a property of the language, but only an approach for enabling a type safety proof. The rationale for this statement is that a cast expression is first evaluated to an expression $\langle \tau_2 \Leftarrow \tau_1 \rangle v$ using (E-CAST), and then handled by the following rules:

- (E-CAST-ARROW) - the case is broken up into two separate casts, by introducing a new lambda abstraction.
- (E-CAST-GAMMA) - the cast is thrown away.
- (E-CAST-MODEL) - casts surrounding a model value gets thrown away before deconstructing a model.

Note that the resulting value can include casts, which is also defined as a value.

10.4 Type Safety

In this section we prove type safety by first proving the usual progress and preservation lemmas for the intermediate language $\lambda^{<>}_{LC}$. Type safety for $\lambda^{<>}$ is then established using the soundness of model lifting (Lemma 10.1) and soundness of cast insertion

(Lemma 10.2). The proof strategy for type safety that we use has its origins in the syntactic soundness approach by Wright and Felleisen [151], but is now typically organized in a different way. We are using an approach similar to Pierce [124].

We start by proving basic lemmas about the typing relation. First, we prove the inversion of typing relation.

Lemma 10.3 (Inversion of Typing Relation)

1. *If* $\Gamma \vdash x : \tau$ *then* $\Gamma(x) = \tau$.
2. *If* $\Gamma \vdash (u : \tau_1) : \tau$ *then* $\tau = \tau_1$ *and* $\mathtt{<>} \sim \tau_1$.
3. *If* $\Gamma \vdash \lambda x : \tau_1 . e_2 : \tau$ *then there exists a* τ_2 *such that* $\tau = \tau_1 \to \tau_2$ *and* $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$.
4. *If* $\Gamma \vdash c : \tau$ *then* $\Delta(c) = \tau$.
5. *If* $\Gamma \vdash \nu(\tau_1) : \tau$ *then* $\tau = \mathtt{<}\tau_1\mathtt{>}$.
6. *If* $\Gamma \vdash (\mathtt{val}\ e_1 : \tau_1) : \tau$ *then* $\tau = \mathtt{<}\tau_1\mathtt{>}$ *and* $\Gamma \vdash e_1 : \tau_1$.
7. *If* $\Gamma \vdash e_1\ e_2 : \tau$ *then there exists a* τ_{11} *such that* $\Gamma \vdash e_1 : \tau_{11} \to \tau$ *and* $\Gamma \vdash e_2 : \tau_{11}$.
8. *If* $\Gamma \vdash e_1 @ e_2 : \tau$ *then* $\tau = \mathtt{<>}$ *and* $\Gamma \vdash e_1 : \mathtt{<>}$ *and* $\Gamma \vdash e_2 : \mathtt{<>}$.
9. *If* $\Gamma \vdash \langle \tau_2 \Leftarrow \tau_1 \rangle e_1 : \tau$ *then* $\tau = \tau_2$ *and* $\Gamma \vdash e_1 : \tau_1$ *and* $\tau_1 \sim \tau_2$.
10. *If* $\Gamma \vdash \mathtt{decon}(e_1, \mathtt{uk} : \tau_4, e_2, e_3) : \tau$ *then there exists a* τ_1 *such that* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma \vdash e_2 : \tau$ *and* $\Gamma \vdash e_3 : \tau$ *and* $\mathtt{<>} \sim \tau_1$ *and* $\mathtt{<>} \sim \tau_4$.
11. *If* $\Gamma \vdash \mathtt{decon}(e_1, x_1 @ x_2, e_2, e_3) : \tau$ *then there exists a* τ_1 *such that* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x_1 : \mathtt{<>}, x_2 : \mathtt{<>} \vdash e_2 : \tau$ *and* $\Gamma \vdash e_3 : \tau$ *and* $\mathtt{<>} \sim \tau_1$.
12. *If* $\Gamma \vdash \mathtt{decon}(e_1, \mathtt{val}\ x : \tau_4, e_2, e_3) : \tau$ *then there exists a* τ_1 *such that* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x : \tau_4 \vdash e_2 : \tau$ *and* $\Gamma \vdash e_3 : \tau$ *and* $\mathtt{<>} \sim \tau_1$.

Proof: Immediate from the definition of $\Gamma \vdash e' : \tau$. □

The next lemmas tell us the shape of a value, given its type:

Lemma 10.4 (Canonical Forms)

1. *If* $\Gamma \vdash v : \gamma$ *then* $\exists c \in \mathbb{C}.\ c = v$.
2. *If* $\Gamma \vdash v : \tau_1 \to \tau_2$ *then* $(\exists x\ e.\ (\lambda x : \tau_1 . e) = v)$ *or* $(\exists c. c = v)$.
3. *If* $\Gamma \vdash v : \tau$ *and* $\mathtt{<>} \sim \tau$ *then* $(\exists u.\ u : \tau = v)$ *or* $(\exists \tau_1\ v_1.\ \tau = \mathtt{<}\tau_1\mathtt{>} \wedge \mathtt{val}\ v_1 : \tau_1 = v)$ *or* $(\exists v_1\ v_2.\ \tau = \mathtt{<>}\ \wedge\ v_1 @ v_2 = v)$ *or* $\exists \tau_1\ \xi. \langle \tau \Leftarrow \tau_1 \rangle \xi = v$.

Proof:

1. By induction on a derivation of the statement $\Gamma \vdash v : \gamma$.
2. By induction on a derivation of the statement $\Gamma \vdash v : \tau_1 \to \tau_2$.
3. By induction on a derivation of the statement $\Gamma \vdash v : \tau$. Case (T-CONST) uses Assumption A1. □

We are now ready to state one of the main lemmas of the proof, that a well-typed expression is either a value or we can take a step:

Lemma 10.5 (Progress)
If $\vdash e : \tau$ *then* $e \in$ *Values or for all* U *there exists* U' *and* e' *such that* $e \mid U \longrightarrow e' \mid U'$.

Proof: By induction on a derivation of $\vdash e : \tau$. Case (T-VAR) cannot occur, since e is closed. In cases (T-UK), (T-ABS), and (T-CONST) $e \in \mathit{Values}$. Case (T-APP) uses the Canonical Form Lemma number 2. Cases (T-VAL) and (T-MODAPP) are straightforward.

Case (T-CAST): By induction hypothesis, e_1 can either take a step or it is a value. If it can take a step, rule (E-CAST) apply. If it is a value, we perform case analysis on values. If $e_1 \in \mathit{CastValues}$, then e is a value. If e_1 is a lambda, Inversion Lemma number 3 is used and (E-CAST-ARROW) apply. If e_1 is s constant, Inversion Lemma number 4 together with Assumption A1 gives two subcases where (E-CAST-ARROW) and (E-CAST-GAMMA) applies respectively.

Cases (T-DECON-UK), (T-DECON-APP), and (T-DECON-VAL) are proven in the same manner. By induction hypothesis e_1 can either take a step or is a value. If it can take a step, (E-DECON) applies. In case of a value, the Canonical Form Lemma number 3 gives four cases. If $e_1 \in \mathit{CastValues}$ (E-CAST-MODEL) applies. In the other cases (E-DECON-T) or (E-DECON-F) apply. □

Assumption A2 (δ-typability).
If $\Delta(c) = \tau_1 \rightarrow \tau_2$ and $\Gamma \vdash v : \tau_1$ then $\Gamma \vdash \delta(c, v) : \tau_2$.

Towards proving the Preservation Lemma, we need for the case (T-APP) a Substitution Lemma that in turn needs an Environment Weakening Lemma.

Lemma 10.6 (Environment Weakening)
If $\Gamma \vdash e : \tau$ *and* $\Gamma \subseteq \Gamma'$ *then* $\Gamma' \vdash e : \tau$.

Proof: Straightforward induction on a derivation of $\Gamma \vdash e : \tau$. □

Lemma 10.7 (Substitution)
If $\Gamma, y : \tau' \vdash e : \tau$ *and* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash [y \mapsto e']e : \tau$.

Proof: Straightforward induction on a derivation of $\Gamma \vdash e : \tau$, where case (T-ABS) uses the Environment Weakening Lemma. □

Lemma 10.8 (Preservation)
If $\Gamma \vdash e : \tau$ *and* $e \mid U \longrightarrow e' \mid U'$ *then* $\Gamma \vdash e' : \tau$.

Proof: By induction on a derivation of $\Gamma \vdash e : \tau$. Cases (T-VAR), (T-UK), (T-ABS), and (T-CONST) are vacuously true. Case (T-NEWUK) is straightforward. Case (T-APP) uses the inversion lemma, substitution lemma, and Assumption A2. Case (T-CAST) has three cases, where case (E-CAST-ARROW) uses the Environment Weakening Lemma. Cases (T-DECON-UK), (T-DECON-APP), and (T-DECON-VAL) use the Inversion Lemma. □

Theorem 10.1 (Type Safety of $\lambda^{<>}$)
If $\vdash_L e_1 \rightsquigarrow e_2 : \tau$ *then there exists an* e_3 *such that* $\vdash_C e_2 \rightsquigarrow e_3 : \tau$ *and (if* $e_3 \mid U_3 \longrightarrow^* e_4 \mid U_4$ *then* $\vdash e_4 : \tau$ *and (*$e_4 \in$ *Values or there exists* e_5 *and* U_5 *such that* $e_4 \mid U_4 \longrightarrow e_5 \mid U_5$*)).*

Proof: By applying Lemma 10.1, soundness of model lifting, to $\vdash_L e_1 \rightsquigarrow e_2 : \tau$ we have $\vdash_C e_2 \rightsquigarrow e_3 : \tau$ for some e_3. Also, by soundness of cast insertion (Lemma 10.2), we have $\vdash e_3 : \tau$. By induction on a derivation of $e \mid U \ \longrightarrow^* \ e' \mid U'$ we have two cases: In the base case (RTC-REFL) $e = e'$ and we directly have $\vdash e' : \tau$. By applying Progress to $\vdash e' : \tau$ we show that e' is a value or there exists e'' and U'' such that $e' \mid U' \ \longrightarrow e'' \mid U''$. For case (RTC-STEP) we have by induction hypothesis $\vdash e' : \tau$. Also, by applying Preservation to assumption $e' \mid U' \ \longrightarrow e'' \mid U''$, we obtain $\vdash e'' : \tau$. By applying Progress to $\vdash e'' : \tau$ we reach the conclusion. □

10.5 Extending the Core

In this section, we discuss some essential parts when extending the core of MKL with other constructs.

10.5.1 Other Expressions and the Bot Type

We do not see that other expressions, such as `if`-expressions, the list constructor, tuples, and built-in ADTs for Map and Set, give any extra concern regarding the extra complexity of model types. No other expressions are lifted to models than the expressions explained in the previous section.

One detail that can be elegantly handled using type consistency is the type for the `error` expression, i.e., an expression that should terminate the program. We give an `error` the type `Bot`. The problem is however how to type this when the type system does not have subtyping. One solution is to use the consistency relation. If we extend the restriction operator $\tau_1 \parallel_{\tau_2}$ with a rule for `Bot` we get the following extended definition:

Restriction (extended) $\boxed{\tau \parallel_{\tau}}$

$<\tau> \parallel_{<>}$	$=$	$<>$	
$<\tau_1> \parallel_{<\tau_2>}$	$=$	$<\tau_1 \parallel_{\tau_2}>$	
$\tau_{11} \rightarrow \tau_{12} \parallel_{\tau_{21} \rightarrow \tau_{22}}$	$=$	$\tau_{11} \parallel_{\tau_{21}} \rightarrow \tau_{12} \parallel_{\tau_{22}}$	
$\tau_1 \parallel_{\tau_2}$	$=$	τ_1	if $\tau_1 \neq <\tau_3>$ and $\tau_1 \neq \tau_{11} \rightarrow \tau_{12}$ and $\tau_1 \neq \texttt{Bot}$ for some τ_3, τ_{11}, and τ_{12}
$\texttt{Bot} \parallel_{\tau}$	$=$	τ	

Lemma 10.9 (Consistency of Bot)

$\forall \tau.(\texttt{Bot} \sim \tau \textit{ and } \tau \sim \texttt{Bot})$

Proof: By rule 5 of the definition of restriction (extended) we have $\texttt{Bot} \parallel_{\tau} = \tau$ and by rule 4 we have $\tau \parallel_{\texttt{Bot}} = \tau$. Hence, $\texttt{Bot} \sim \tau$ by using Definition 10.2.1. Because $\sim$ is symmetric, we have $\tau \sim \texttt{Bot}$. □

Hence, all types are consistent with `Bot`. It turns out that this approach also solves the problem of specifying types explicitly for the empty list in a language without type vari-

ables. We simply give the empty list type `[Bot]`. Type `Bot` cannot be defined explicitly by the user because this would break the type system.

10.5.2 Pattern Matching

We have in previous chapters showed that pattern matching is an essential part of the language. However, pattern matching using `match`-expressions are not part of the core language presented in this chapter. Instead `match`-expressions are defined using *derived forms*, meaning that there is a translation step between the concrete syntactic form of a `match`-expression and an expression in an intermediate language based on the semantics presented in this chapter. This translation has not yet been formally defined for MKL.

10.5.3 Lifting and Binary Operators

One problem of symmetry appears when lifting binary operators that can be partially applied, e.g., `(+)` of type `Int -> Int -> Int`. For example, in expression

```
let x:<Int> in
  ((+) x)  3
```

sub-expression `(+) x` is lifted to a model application because `x` is of model type. Hence, `((+) @ x)` applied to `3` is also lifted. However, if the order of operands to `(+)` is reversed

```
let x:<Int> in
  ((+) 3)  x
```

expression `(+) 3` will not be lifted because the argument `3` is not of model type. Because operator `(+)` is in curried form, `(+)` can be partially applied to `3`. Hence, the value of `(+) 3` is embedded into a model value instead of being translated into a model application.

Our first attempt to solve this problem of non-symmetry was to include special rules for binary built-in operators in the model lifting relation, i.e., as part of the type system. However, this resulted in a very complicated type system. Informally, our solution in the implementation is instead to add an extra match rule for `decon` where model values with embedded partially applied binary operators can be deconstructed.

10.5.4 Equality

In the full MKL language, we have a built-in polymorphic equality operator. In the current version of the language, we define equivalence of values with binary relation $\equiv_\alpha$, meaning syntactic equality up to renaming of bound variables (α-conversion). This is satisfying for constant terms, tuples, unknowns etc. in an interpreted setting. The approach works for lambda expressions, but the current prototype implementation gets very slow because the environments of the closures are compared. Alpha-equivalence is handled by nameless representation of the environment, i.e., we use de Bruijn-indices [46].
Type and translation rules for equality can be defined as follows:

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e_1':\tau_1 \quad \Gamma \vdash_L e_2 \rightsquigarrow e_2':\tau_2 \quad \texttt{<>} \not\sim \tau_1 \quad \texttt{<>} \sim \tau_2 \quad \texttt{<}\tau_1\texttt{>} \sim \tau_2}{\Gamma \vdash_L e_1 \texttt{ == } e_2 \rightsquigarrow (\texttt{val}\ e_1':\tau_1 \texttt{ == } e_2'):\texttt{Bool}} \text{ (L-EQUAL1)}$$

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e_1':\tau_1 \quad \Gamma \vdash_L e_2 \rightsquigarrow e_2':\tau_2 \quad \texttt{<>} \sim \tau_1 \quad \texttt{<>} \not\sim \tau_2 \quad \tau_1 \sim \texttt{<}\tau_2\texttt{>}}{\Gamma \vdash_L e_1 \texttt{ == } e_2 \rightsquigarrow (e_1' \texttt{ == } \texttt{val}\ e_2':\tau_2):\texttt{Bool}} \text{ (L-EQUAL2)}$$

$$\frac{\Gamma \vdash_L e_1 \rightsquigarrow e_1':\tau_1 \qquad \Gamma \vdash_L e_2 \rightsquigarrow e_2':\tau_2 \qquad \tau_1 \sim \tau_2}{\Gamma \vdash_L e_1 \texttt{ == } e_2 \rightsquigarrow e_1' \texttt{ == } e_2':\texttt{Bool}} \text{ (L-EQUAL3)}$$

Hence, if one of the operands is of model type and the other one is not, the expression that is not a model is lifted (rules (L-EQUAL1) and (L-EQUAL2)). If both operands are models or none of them, no lifting occurs (L-EQUAL3).

Note that in contrast to an application expression, the equality operator is never lifted to become a model. If that would be the case, it could not be used to compare e.g., unknowns. An alternative would be to have more than one equality operator, one that is lifted and one that is not. However, at the current stage, we do not see any reason to introduce this.

10.6 Chapter Summary and Conclusions

We have in this chapter presented a formal semantics of an essential core of the MKL language. We have tried to formulate our semantics with rigor by making all definitions as clear as possible.

The choice of small-step semantics, to have several intermediate languages, and to insert casts are all choices due to the type safety proof. Our type safety proof increases our confidence of the language, but we would at the same time stress that it does not guarantee correctness between the formal semantics and an implementation.

$$\boxed{\Gamma \vdash_L e \leadsto e' : \tau}$$

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash_L x \leadsto x : \tau_1}\ \text{(L-VAR)} \qquad \frac{\texttt{<>} \sim \tau_1}{\Gamma \vdash_L u : \tau_1 \leadsto (u : \tau_1) : \tau_1}\ \text{(L-UK)}$$

$$\frac{\Gamma, x : \tau_1 \vdash_L e_2 \leadsto e'_2 : \tau_2}{\Gamma \vdash_L \lambda x : \tau_1 . e_2 \leadsto \lambda x : \tau_1 . e'_2 : \tau_1 \to \tau_2}\ \text{(L-ABS)} \qquad \frac{\Delta(c) = \tau_1}{\Gamma \vdash_L c \leadsto c : \tau_1}\ \text{(L-CONST)}$$

$$\frac{}{\Gamma \vdash_L \nu(\tau_1) \leadsto \nu(\tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(L-NEWUK)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \tau_1}{\Gamma \vdash_L \texttt{val}\ e_1 : \tau_1 \leadsto (\texttt{val}\ e'_1 : \tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(L-VAL)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \tau_{11} \to \tau_{12} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \tau_{11} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto e'_1\ e'_2 : \tau_{12}}\ \text{(L-APP)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \tau_{11} \to \tau_{12} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \texttt{<}\tau_{11}\texttt{>} \sim \tau_2 \quad \tau_{11} \not\sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto (\texttt{val}\ e'_1 : \tau_{11} \to \tau_{12}) @ e'_2 : \texttt{<}\tau_{12}\texttt{>}}\ \text{(L-APPM1)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \texttt{<>} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \texttt{<>} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto e'_1 @ e'_2 : \texttt{<>}}\ \text{(L-APPM2)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \texttt{<>} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \texttt{<>} \not\sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto e'_1 @ (\texttt{val}\ e'_2 : \tau_2) : \texttt{<>}}\ \text{(L-APPM3)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \texttt{<}\tau_{11} \to \tau_{12}\texttt{>} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \texttt{<}\tau_{11}\texttt{>} \sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto e'_1 @ e'_2 : \texttt{<}\tau_{12}\texttt{>}}\ \text{(L-APPM4)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \texttt{<}\tau_{11} \to \tau_{12}\texttt{>} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \tau_{11} \sim \tau_2 \quad \texttt{<}\tau_{11}\texttt{>} \not\sim \tau_2}{\Gamma \vdash_L e_1\ e_2 \leadsto e'_1 @ (\texttt{val}\ e'_2 : \tau_2) : \texttt{<}\tau_{12}\texttt{>}}\ \text{(L-APPM5)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \texttt{<>} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \texttt{<>} \sim \tau_2}{\Gamma \vdash_L e_1 @ e_2 \leadsto e'_1 @ e'_2 : \texttt{<>}}\ \text{(L-MODAPP1)}$$

$$\frac{\Gamma \vdash_L e_1 \leadsto e'_1 : \texttt{<}\tau_{11} \to \tau_{12}\texttt{>} \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \texttt{<}\tau_{11}\texttt{>} \sim \tau_2}{\Gamma \vdash_L e_1 @ e_2 \leadsto e'_1 @ e'_2 : \texttt{<}\tau_{12}\texttt{>}}\ \text{(L-MODAPP2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e'_1 : \tau_1 \quad \Gamma \vdash_L e_2 \leadsto e'_2 : \tau_2 \quad \Gamma \vdash_L e_3 \leadsto e'_3 : \tau_3 \\ \texttt{<>} \sim \tau_1 \qquad \tau_2 \sim \tau_3 \qquad \texttt{<>} \sim \tau_4\end{array}}{\Gamma \vdash_L \texttt{decon}(e_1, \texttt{uk} : \tau_4, e_2, e_3) \leadsto \texttt{decon}(e'_1, \texttt{uk} : \tau_4, e'_2, e'_3) : (\tau_2 \parallel_{\tau_3})}\ \text{(L-DECON-UK)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e'_1 : \tau_1 \qquad \Gamma, x_1 : \texttt{<>}, x_2 : \texttt{<>} \vdash_L e_2 \leadsto e'_2 : \tau_2 \\ \Gamma \vdash_L e_3 \leadsto e'_3 : \tau_3 \qquad \texttt{<>} \sim \tau_1 \qquad \tau_2 \sim \tau_3\end{array}}{\Gamma \vdash_L \texttt{decon}(e_1, x_1 @ x_2, e_2, e_3) \leadsto \texttt{decon}(e'_1, x_1 @ x_2, e'_2, e'_3) : (\tau_2 \parallel_{\tau_3})}\ \text{(L-DECON-APP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_L e_1 \leadsto e'_1 : \tau_1 \qquad \Gamma, x : \tau_4 \vdash_L e_2 \leadsto e'_2 : \tau_2 \\ \Gamma \vdash_L e_3 \leadsto e'_3 : \tau_3 \qquad \texttt{<>} \sim \tau_1 \qquad \tau_2 \sim \tau_3\end{array}}{\Gamma \vdash_L \texttt{decon}(e_1, \texttt{val}\ x : \tau_4, e_2, e_3) \leadsto \texttt{decon}(e'_1, \texttt{val}\ x : \tau_4, e'_2, e'_3) : (\tau_2 \parallel_{\tau_3})}\ \text{(L-DECON-VAL)}$$

Figure 10.2: *Model lifting*

$$\boxed{\Gamma \vdash_C e \rightsquigarrow e' : \tau}$$

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash_C x \rightsquigarrow x : \tau_1} \text{(C-VAR)} \qquad \frac{\texttt{<>} \sim \tau_1}{\Gamma \vdash_C (u : \tau_1) \rightsquigarrow (u : \tau_1) : \tau_1} \text{(C-UK)}$$

$$\frac{\Gamma, x : \tau_1 \vdash_C e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash_C \lambda x : \tau_1 . e_2 \rightsquigarrow \lambda x : \tau_1 . e_2' : \tau_1 \to \tau_2} \text{(C-ABS)} \qquad \frac{\Delta(c) = \tau_1}{\Gamma \vdash_C c \rightsquigarrow c : \tau_1} \text{(C-CONST)}$$

$$\frac{}{\Gamma \vdash_C \nu(\tau_1) \rightsquigarrow \nu(\tau_1) : \texttt{<}\tau_1\texttt{>}} \text{(C-NEWUK)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1}{\Gamma \vdash_C \texttt{val}\, e_1 : \tau_1 \rightsquigarrow (\texttt{val}\, e_1' : \tau_1) : \texttt{<}\tau_1\texttt{>}} \text{(C-VAL)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_{11} \to \tau_{12} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \tau_{11} \sim \tau_2}{\Gamma \vdash_C e_1\ e_2 \rightsquigarrow e_1'\ (\langle \tau_{11} \Leftarrow \tau_2 \rangle e_2') : \tau_{12}} \text{(C-APP)}$$

$$\frac{\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \texttt{<>} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \texttt{<>} \sim \tau_2}{\Gamma \vdash_C e_1 @ e_2 \rightsquigarrow e_1' @ \langle \texttt{<>} \Leftarrow \tau_2 \rangle e_2' : \texttt{<>}} \text{(C-MODAPP1)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \texttt{<}\tau_{11} \to \tau_{12}\texttt{>} \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \texttt{<}\tau_{11}\texttt{>} \sim \tau_2 \\ e_1'' = (\langle \texttt{<>} \Leftarrow \texttt{<}\tau_{11} \to \tau_{12}\texttt{>} \rangle e_1') \qquad e_2'' = \langle \texttt{<>} \Leftarrow \tau_2 \rangle e_2'\end{array}}{\Gamma \vdash_C e_1 @ e_2 \rightsquigarrow \langle \texttt{<}\tau_{12}\texttt{>} \Leftarrow \texttt{<>} \rangle (e_1'' @ e_2'') : \texttt{<}\tau_{12}\texttt{>}} \text{(C-MODAPP2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \quad \Gamma \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \quad \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \\ \texttt{<>} \sim \tau_1 \qquad \tau_2 \sim \tau_3 \qquad \texttt{<>} \sim \tau_4 \qquad \tau_5 = \tau_2 \parallel_{\tau_3} \\ e_2'' = \langle \tau_5 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_5 \Leftarrow \tau_3 \rangle e_3'\end{array}}{\Gamma \vdash_C \texttt{decon}(e_1, \texttt{uk} : \tau_4, e_2, e_3) \rightsquigarrow \texttt{decon}(e_1', \texttt{uk} : \tau_4, e_2'', e_3'') : \tau_5} \text{(C-DECON-UK)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma, x_1 : \texttt{<>}, x_2 : \texttt{<>} \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \qquad \texttt{<>} \sim \tau_1 \qquad \tau_2 \sim \tau_3 \qquad \tau_4 = \tau_2 \parallel_{\tau_3} \\ e_2'' = \langle \tau_4 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_4 \Leftarrow \tau_3 \rangle e_3'\end{array}}{\Gamma \vdash_C \texttt{decon}(e_1, x_1 @ x_2, e_2, e_3) \rightsquigarrow \texttt{decon}(e_1', x_1 @ x_2, e_2'', e_3'') : \tau_4} \text{(C-DECON-APP)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_C e_1 \rightsquigarrow e_1' : \tau_1 \qquad \Gamma, x : \tau_4 \vdash_C e_2 \rightsquigarrow e_2' : \tau_2 \\ \Gamma \vdash_C e_3 \rightsquigarrow e_3' : \tau_3 \qquad \texttt{<>} \sim \tau_1 \qquad \tau_2 \sim \tau_3 \qquad \tau_5 = \tau_2 \parallel_{\tau_3} \\ e_2'' = \langle \tau_5 \Leftarrow \tau_2 \rangle e_2' \qquad e_3'' = \langle \tau_5 \Leftarrow \tau_3 \rangle e_3'\end{array}}{\Gamma \vdash_C \texttt{decon}(e_1, \texttt{val}\ x : \tau_4, e_2, e_3) \rightsquigarrow \texttt{decon}(e_1', \texttt{val}\ x : \tau_4, e_2'', e_3'') : \tau_5} \text{(C-DECON-VAL)}$$

Figure 10.3: *Cast insertion*

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\Gamma(x) = \tau_1}{\Gamma \vdash x : \tau_1}\ \text{(T-VAR)} \qquad \frac{\texttt{<>} \sim \tau_1}{\Gamma \vdash (u : \tau_1) : \tau_1}\ \text{(T-UK)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e_2 : \tau_1 \rightarrow \tau_2}\ \text{(T-ABS)} \qquad \frac{\Delta(c) = \tau_1}{\Gamma \vdash c : \tau_1}\ \text{(T-CONST)}$$

$$\frac{}{\Gamma \vdash \nu(\tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(T-NEWUK)} \qquad \frac{\Gamma \vdash e_1 : \tau_1}{\Gamma \vdash (\texttt{val}\ e_1 : \tau_1) : \texttt{<}\tau_1\texttt{>}}\ \text{(T-VAL)}$$

$$\frac{\Gamma \vdash e_1 : \tau_{11} \rightarrow \tau_{12} \quad \Gamma \vdash e_2 : \tau_{11}}{\Gamma \vdash e_1\ e_2 : \tau_{12}}\ \text{(T-APP)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{<>} \qquad \Gamma \vdash e_2 : \texttt{<>}}{\Gamma \vdash e_1 @ e_2 : \texttt{<>}}\ \text{(T-MODAPP)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \tau_1 \sim \tau_2}{\Gamma \vdash \langle \tau_2 \Leftarrow \tau_1 \rangle e_1 : \tau_2}\ \text{(T-CAST)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2 \qquad \texttt{<>} \sim \tau_1 \qquad \texttt{<>} \sim \tau_4}{\Gamma \vdash \texttt{decon}(e_1, \texttt{uk} : \tau_4, e_2, e_3) : \tau_2}\ \text{(T-DECON-UK)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x_1 : \texttt{<>}, x_2 : \texttt{<>} \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2 \qquad \texttt{<>} \sim \tau_1}{\Gamma \vdash \texttt{decon}(e_1, x_1 @ x_2, e_2, e_3) : \tau_2}\ \text{(T-DECON-APP)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_4 \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_2 \qquad \texttt{<>} \sim \tau_1}{\Gamma \vdash \texttt{decon}(e_1, \texttt{val}\ x : \tau_4, e_2, e_3) : \tau_2}\ \text{(T-DECON-VAL)}$$

Figure 10.4: *Type System for* $\lambda_{LC}^{\texttt{<>}}$

Computation Rules $\boxed{e \mid U \longrightarrow e' \mid U'}$

$$(\lambda x\!:\!\tau_1.e_1)v_1 \mid U \longrightarrow [x \mapsto v_1]e_1 \mid U \quad \text{(E-APPABS)} \qquad c_1\ v_1 \mid U \longrightarrow \delta(c_1, v_1) \mid U \quad \text{(E-DELTA)}$$

$$\frac{u \notin U}{\nu(\tau_1) \mid U \longrightarrow u\!:\!\texttt{<}\tau_1\texttt{>} \mid U \cup \{u\}} \text{(E-NEWUK)}$$

$$\frac{\mathit{match}(w_1, d, e_2, e_2')}{\texttt{decon}(w_1, d, e_2, e_3) \mid U \longrightarrow e_2' \mid U} \text{(E-DECON-T)}$$

$$\frac{\neg\mathit{match}(w_1, d, e_2, e_2')}{\texttt{decon}(w_1, d, e_2, e_3) \mid U \longrightarrow e_3 \mid U} \text{(E-DECON-F)}$$

$$\langle \tau_1 \rightarrow \tau_2 \Leftarrow \tau_3 \rightarrow \tau_4 \rangle v_1 \mid U \longrightarrow \lambda x\!:\!\tau_1.\langle \tau_2 \Leftarrow \tau_4 \rangle (v_1 \langle \tau_3 \Leftarrow \tau_1 \rangle x) \mid U \quad \text{(E-CAST-ARROW)}$$

$$\langle \gamma \Leftarrow \gamma \rangle v_1 \mid U \longrightarrow v_1 \mid U \quad \text{(E-CAST-GAMMA)}$$

$$\texttt{decon}(\langle \tau_2 \Leftarrow \tau_1 \rangle \xi_1, d, e_1, e_2) \mid U \longrightarrow \texttt{decon}(\xi_1, d, e_1, e_2) \mid U \quad \text{(E-CAST-MODEL)}$$

Congruence Rules $\boxed{e \mid U \longrightarrow e' \mid U'}$

$$\frac{e_1 \mid U \longrightarrow e_1' \mid U'}{e_1\ e_2 \mid U \longrightarrow e_1'\ e_2 \mid U'} \text{(E-APP1)} \qquad \frac{e_2 \mid U \longrightarrow e_2' \mid U'}{v_1\ e_2 \mid U \longrightarrow v_1\ e_2' \mid U'} \text{(E-APP2)}$$

$$\frac{e_1 \mid U \longrightarrow e_1' \mid U'}{e_1 @ e_2 \mid U \longrightarrow e_1' @ e_2 \mid U'} \text{(E-MODAPP1)} \qquad \frac{e_2 \mid U \longrightarrow e_2' \mid U'}{v_1 @ e_2 \mid U \longrightarrow v_1 @ e_2' \mid U'} \text{(E-MODAPP2)}$$

$$\frac{e_1 \mid U \longrightarrow e_1' \mid U'}{\texttt{val}\ e_1\!:\!\tau_1 \mid U \longrightarrow \texttt{val}\ e_1'\!:\!\tau_1 \mid U'} \text{(E-MODVAL)}$$

$$\frac{e_1 \mid U \longrightarrow e_1' \mid U'}{\texttt{decon}(e_1, d, e_2, e_3) \mid U \longrightarrow \texttt{decon}(e_1', d, e_2, e_3) \mid U'} \text{(E-DECON)}$$

$$\frac{e_1 \mid U \longrightarrow e_1' \mid U'}{\langle \tau_2 \Leftarrow \tau_1 \rangle e_1 \mid U \longrightarrow \langle \tau_2 \Leftarrow \tau_1 \rangle e_1' \mid U'} \text{(E-CAST)}$$

Match Rules $\boxed{\mathit{match}(e_1, d, e_2, e_3)}$

$$\mathit{match}(u\!:\!\tau_1, \texttt{uk}\!:\!\tau_1, e_1, e_1) \quad \text{(M-UK)}$$

$$\mathit{match}(v_1 @ v_2, x_1 @ x_2, e_1, (\lambda x_1\!:\!\texttt{<>}.\lambda x_2\!:\!\texttt{<>}.e_1)\ v_1\ v_2) \quad \text{(M-MAPP)}$$

$$\mathit{match}(\texttt{val}\ v_1\!:\!\tau_1, \texttt{val}\ x\!:\!\tau_1, e_1, (\lambda x\!:\!\tau_1.e_1)\ v_1) \quad \text{(M-MVAL)}$$

Reflexive Transitive Closure $\boxed{e \mid U \ \longrightarrow^* \ e' \mid U'}$

$$e \mid U \ \longrightarrow^* \ e' \mid U' \quad \text{(RTC-REFL)} \qquad \frac{e \mid U \ \longrightarrow^* \ e' \mid U' \quad e' \mid U' \ \longrightarrow \ e'' \mid U''}{e \mid U \ \longrightarrow^* \ e'' \mid U''} \text{(RTC-STEP)}$$

Figure 10.5: *Small-step operational semantics for* $\lambda_{LC}^{\texttt{<>}}$.

Free variables $\boxed{FV(e)}$

$$
\begin{array}{lcl}
FV(x) & = & \{x\} \\
FV(\lambda x{:}\tau.e) & = & FV(e) \setminus \{x\} \\
FV(e_1\ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(c) & = & \emptyset \\
FV(u{:}\tau) & = & \emptyset \\
FV(\nu(\tau)) & = & \emptyset \\
FV(e_1 @ e_2) & = & FV(e_1) \cup FV(e_2) \\
FV(\texttt{val}\ e{:}\tau) & = & FV(e) \\
FV(\texttt{decon}(e_1, d, e_2, e_3)) & = & FV(e_1) \cup FV(e_2) \cup FV(e_3)
\end{array}
$$

Substitiution $\boxed{[x \mapsto e]e}$

$$
\begin{array}{lcll}
[x \mapsto e]x & = & e & \\
[x \mapsto e]y & = & y & \text{if } x \neq y \\
[x \mapsto e]\lambda y{:}\tau.e_1 & = & \lambda y{:}\tau.[x \mapsto e]e_1 & \text{if } x \neq y \text{ and } y \notin FV(e) \\
[x \mapsto e]e_1\ e_2 & = & [x \mapsto e]e_1\ [x \mapsto e]e_2 & \\
[x \mapsto e]c & = & c & \\
[x \mapsto e]u{:}\tau & = & u{:}\tau & \\
[x \mapsto e]\nu(\tau) & = & \nu(\tau) & \\
[x \mapsto e]e_1 @ e_2 & = & [x \mapsto e]e_1 @ [x \mapsto e]e_2 & \\
[x \mapsto e]\texttt{val}\ e_1{:}\tau & = & \texttt{val}\ [x \mapsto e]e_1{:}\tau & \\
[x \mapsto e]\texttt{decon}(e_1, d, e_2, e_3) & = & \texttt{decon}([x \mapsto e]e_1, d, [x \mapsto e]e_2, [x \mapsto e]e_3) &
\end{array}
$$

Figure 10.6: *Free variables and substitution.*

11

Elaboration Semantics

In Chapter 2, we defined the *elaboration* phase as the translation from a model to a hybrid DAE. In this chapter, we discuss the elaboration phase with focus on two main areas: *connection semantics* and *extracting model information*. This chapter is organized as follows:

- We give an overview of the different activities that are typically involved during elaboration in an compiler for an EOO language. We explain briefly when type checking is performed on MKL models and how the instance hierarchy of the model is collapsed (Section 11.1).

- We explain the connection semantics for handling acausal connections in MKL, as well as discuss how the semantics relate to Modelica's informal connection semantics. The semantics are formally defined using a recursive functional definition. This is followed by an executable specification where the semantics are also specified as MKL functions (Section 11.2).

- We discuss the problem of extracting simulation results from a model, i.e., how to specify which variables should be presented to a user. We suggest a solution to the problem using the construct of *probes*, and discuss pros and cons compared to using *hierarchy names* (Section 11.3).

11.1 Overview of Elaboration

The process of elaboration, where a EOO model is translated into an equation system can informally be described to perform at least the following three main activities:

- *Type checking of models.* Check that parameterized models conform to the type rules of the language and that basic operations and function calls are type correct.

For example, a function having one argument cannot be applied to two arguments and a plus operator cannot have a string as its left operand and an integer as its right operand, etc.

- *Collapsing the instance hierarchy*. During this activity, new unknowns and equations are created for sub-components of a model. For example, if a model contains two resistors `R1` and `R2`, where `R1` is parameterized with 10 ohm, and `R2` with 50 ohm, two equations are created `u1 = 10 * i1` and `u2 = 50 * i2`. Moreover, unknowns, such as the voltage drop over the components must be different for the components. Hence, `u1` and `u2` must be different unknowns.
- *Connection semantics*. Acausal ports contain flow and potential variables, where the former must sum-to-zero at connection points and the latter must have the same potential at the connection point. This activity generates equations and unknowns to enable acausal modeling.

In Modelica, all these three activities are typically performed at compile time in a Modelica compiler. In the Modelica specification, it is not specified in which order the activities should be performed. Commonly, the two first activities are performed together, while the last one could be performed in a separate phase.

MKL separates these activities into distinct phases. These activities are performed at different point in time during the process:

1. *Type checking* is performed at compile time by the MKL compiler (or before evaluation in an interpretive setting).
2. *Collapsing the instance hierarchy* is performed at runtime when executing the MKL program.
3. *Connection equation generation* is generated at runtime by a user defined function that performs intensional analysis (inspects) the equation system of the model.

We will now briefly discuss the first two activities and then give a detailed description of the connection semantics in Section 11.2.

11.1.1 Type Checking of Models

Of the described activities in the elaboration process, only type checking is performed at compile time (or before evaluation in the interpretive setting). Because EOO models in MKL are ordinary functions defined in an MKL program, type checking of a model is the same as type checking a program.

For example, recall the `Mechsys` model described in Chapter 8:

```
let MechSys =
  let r1:Rotational in
  let r2:Rotational in
  let r3:Electrical in
  DCMotor r1;
  Inertia 0.2 r1 r2;
  FlexibleShaft 120 r2 r3
```

The type checker will report a type error for argument `r3` on the last line saying that a rotational node was expected, but an electrical node supplied. Note that we have in this example changed the type of the definition of node `r3` to be `Electrical`.

This kind of type errors is caught by the MKL compiler, even if the DSL for the mechatronic domain was defined as a library in MKL itself. We have specified and discussed the type system for performing these checks in Chapter 10.

Note that certain kinds of checks are currently not performed directly by the type checker. One such example is constraint delta checking, as presented in Chapter 6. This checking can of course be performed at the equation level using intensional analysis on the model, but in such a case we loose the property of isolating and locating the source of the error - one of the main benefits of the constraint delta approach. We see it as a challenge and future work to incorporate such a checking at the kernel language level, without being dependent on properties of the DSL.

11.1.2 Collapsing the Instance Hierarchy

A model abstraction in MKL is created using ordinary higher-order functions. Creating an instance of a model means passing arguments to the model so that an equation system can be generated. For example, the type signature for the `Resistor` model is

```
Real -> Electrical -> Electrical -> Equations
```

Hence, by applying `Resistor` to three arguments (the resistance instance and two electrical nodes), the function will be computed and an equation system will be returned. Consequently, the formal semantics for collapsing the instance hierarchy is defined by the operational dynamic semantics described in Chapter 10.

11.2 Connection Semantics

The connection semantics, i.e., the ability of the EOO language to handle acausal connections, is handled differently in different languages. We will in this section formalize the semantics of generating correct equations and unknowns for MKL. This semantics is different from the Modelica semantics in several aspects. However, we will see that they both give the same semantic behaviour, i.e., that we can define the same kind of components and that the generated equation systems give the same solution after simulation.

11.2.1 A Minimal Circuit in Modelica

Let us first give an intuition for the Modelica approach of describing connection semantics. Figure 11.1 shows a graphical view of the circuit. The circuit contains a direct current constant voltage source `DC`, a resistor `R`, and a ground `G`. The figure shows a visualization of unknowns and their location after elaborating the components with a Modelica compiler. Both the voltage source and the resistor have a positive connector[1] `p` (filled square) and a negative connector `n` (unfilled). The ground only has a positive connector. Each connector creates two variables: `i` for current and `v` for voltage. These are defined in the connector class:

[1] Remember that in Modelica ports are called connectors

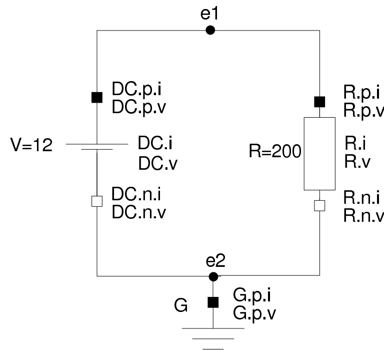


Figure 11.1: *A minimal electrical circuit modeled using Modelica. To the right of each component the unknowns are stated using dot notation, e.g.,* `DC.p.v` *is the potential variable for the positive connector in the* `DC` *component.*

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

We use the prefix notation for defining variables, as customary in Modelica. For example unknowns `R.p.v` and `R.p.i` define the two unknowns in the positive pin of resistor `R`.

The four unknowns `DC.v`, `DC.i`, `R.v`, and `R.v` are defined by the base class `TwoPin`

```
model TwoPin "Superclass of components"
  Pin p, n;
  Voltage v;
  Current i;
equation
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end TwoPin;
```

This base class is also contributing 3 equations. The first one defines the voltage drop over the component. The second one states that the current `p.i` in the positive connector should be equal but with opposite sign compared to the current in the negative connector `p.n`. The reason for the opposite sign is Modelica's convention that connectors always describe the positive direction of flow into the component. The last equation `i = p.i` states that the current in the circuit is the same as the current flowing into the circuit. The `TwoPin` class generates the following equations for the voltage source

```
DC.v = DC.p.v-DC.n.v;
0 = DC.p.i+DC.n.i;
DC.i = DC.p.i;
```

and for the resistor

```
R.v = R.p.v-R.n.v;
```

```
0 = R.p.i+R.n.i;
R.i = R.p.i;
```

Each of the models also contribute one specific equation describing the behavior of the component. For example, in the resistor case it is Ohm law

```
model Resistor "Ideal electrical resistor"
  extends TwoPin;
  parameter Real R "Resistance";
equation
  R*i = v;
end Resistor;
```

generating equation

```
200*R.i = R.v;
```

Similarly the constant voltage source

```
model ConstantVoltage "Source for constant voltage"
  parameter Real V "Value of constant voltage";
  extends Interfaces.TwoPin;
equation
  v = V;
end ConstantVoltage;
```

generates the equation

```
DC.v = 12;
```

Finally, the ground model

```
model Ground "Ground node"
  Interfaces.Pin p;
equation
  p.v = 0;
end Ground;
```

contributes with one equation after elaboration

```
G.p.v = 0;
```

We have so far 14 unknowns but only 9 equations. There are obviously 5 equations missing because the number of equations and unknowns must match. In the top circuit definition

```
model MiniCircuit
  Resistor R(R=200);
  Ground G;
  ConstantVoltage DC(V=12);
equation
  connect(DC.p, R.p);
  connect(G.p, DC.n);
  connect(DC.n, R.n);
end MiniCircuit;
```

we have three `connect`-equations, from which we shall generate new equations. According to Kirchhoff's current law, the sum of current flowing into a node is equal to the sum of current flowing out, i.e., the current should sum-to-zero. Because the connection relationships between components are in Modelica given by binary `connect`-equations, the compiler should first form *connection*[2] sets. Also, for each connector part of the set, information should be provided if the connector is connected from the outside or the inside, called *outside* connectors and *inside* connectors. In this simple case, all connectors are inside connectors because all the component that are connected are located inside circuit `MiniCircuit`.

We now have two connection sets, corresponding to connection nodes `e1` and `e2`. The first `connect`-equation `connect(DC.p, R.p)` generates the set {`DC.p`, `R.p`}, and the second two `connect`-equations the set {`G.p`, `DC.n`, `R.n`}. Note that the first set are all connected to node `e1` and the second set to node `e2`. Note also that if there would have been a fourth `connect`-equation **`connect`**`(G.p, R.n)`, the same connection set should have been generated.

According to the language specification, equations shall be generated so that potential variables (voltage) are equal. Hence, for a set with cardinality n, $n-1$ equations need to be generated. In the first set {`DC.p`, `R.p`}, corresponding to node `e1`, the equation

```
R.p.v = DC.p.v;
```

is generated, and for the second set {`G.p`, `DC.n`, `R.n`}, the equations

```
G.p.v = DC.n.v;
R.n.v = DC.n.v;
```

are generated. Following Kirchhoff's current law, the sum-to-zero equation for node `e1` is

```
DC.p.i+R.p.i = 0;
```

and for node `e2`

```
DC.n.i+G.p.i+R.n.i = 0;
```

Note that all unknowns for the currents can have positive sign because Modelica defines the direction of flow to always be positive *into* the component.

11.2.2 A Minimal Circuit in MKL

Let us first summarize a number of observations from the Modelica example:

- Equations that are describing the behavior of a component, for example Ohm's law `200*R.i = R.v` or the definition of the constant voltage `DC.v = 12` relate to the current flowing *through* the component, or the voltage drop *over* the component. The exception is the ground component, where the unknown of the connector was explicitly accessed, i.e., the equation `G.p.v = 0`.

[2] It is called *connection sets* in the Modelica specification, but should probably be *connector sets*. However, to be compliant with the Modelica specification, we still use the term connection set.

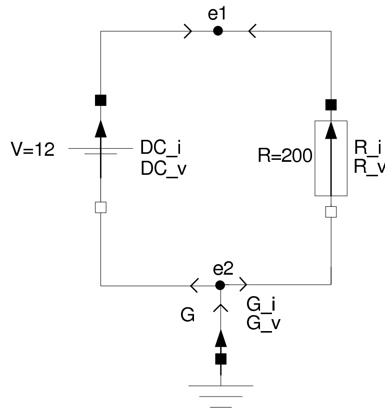


Figure 11.2: *A minimal electrical circuit visualized graphically. To the right of each component the unknowns for a corresponding MKL model are stated. The arrows surrounding the nodes indicate the direction of how the sum-to-zero equation is computed.*

- The unknown for voltage in connectors that are connected, e.g., `R.p.v` and `DC.p.v` always denote the same value and this value always correspond to a connection node (e.g. `e1`), which in turn corresponds to a connection set.
- The number of sum-to-zero equations that are generated is equal to the number of connection nodes in the circuit, assuming that all connectors are connected.

Let us now walk through the same mini circuit example using the MKL approach for connections. Consider the following top level code for model `MiniModel`.

```
let MiniCircuit =
  let e1:Electrical in
  let e2:Electrical in
  Resistor 200. e1 e2;
  ConstantVoltage 12. e1 e2;
  Ground e2
```

The first observation to be made is that we do not have any connect-equations. Instead, we connect the components by supplying nodes to the component. A node, just like most other constructs in MKL, is simply an unknown with a specific type. In the electrical domain, nodes are defined with type `Electrical`, which is defined in the standard library file `electrical.mkl`.

```
type ElectricalNode
type Electrical = <ElectricalNode>
```

Hence, the type `Electrical` is a type alias for the node model type. Because `ElectricalNode` is a user defined type, it can be recognized using pattern matching during intensional analysis of the model.

Models are, as previously discussed, defined as functions. For example, the model

```
let Resistor R:Real -> p:Electrical -> n:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  R *. i = v
```

takes the resistance, as well as two electrical nodes as input. The goal of the elaboration process is to get a set of equations, and because the return type of these models is the type `Equations`, applying the model to its arguments, including the nodes, is the same as model instantiation. Note that this is possible because unknowns are first class and nodes are unknowns.

Now, consider Figure 11.2. In this case, we have only 6 unknowns, each corresponding to the current flow through the connector (e.g., `DC_i`, `R_i`, and `G_i`) and the *voltage drop* over the component (and `DC_v`, `R_v`, and `G_v`). We use the underscore notation to make a distinction from the Modelica example. These unknowns are the unknowns explicitly defined in the `Resistor` model, the `ConstantVoltage` model

```
let ConstantVoltage V:Real -> p:Electrical -> n:Electrical ->
                    Equations =
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  v = V
```

and the `Ground` model

```
let Ground p:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
  ElectricalRefBranch i v p;
  v = 0.
```

Contrary to the Modelica semantics, we have no connectors that define unknowns. The ports, by contrast to Modelica's connectors, are just normal formal parameters to which unknowns can be passed, which are here representing nodes.

Recall now our earlier observation that all potential variables in a connection set corresponds to the same unknown and that each connection set corresponds to one node. Hence, we give our first informal rule for the node elaboration

> **Rule 1 - Unknown potentials:** for each node in the circuit, create an unknown representing the potential in the node.

Following this informal rule on the nodes `e1` and `e2` in Figure 11.2, we generate two more unknowns, denoted

```
e1_v
e2_v
```

The earlier observation also stated that we shall generate a sum-to-zero equation for each node due to Kirchhoff's current law . However, we must with great care choose signs for the currents. This is our first use of the definitions of *branches*. In both the voltage source and in the resistor, we have a definition of `ElectricalBranch`. The intuition

of a branch is that it states the path of flow through a component between two nodes. It is defined in the standard library file `electrical.mkl`:

```
let ElectricalBranch :
  <Real -> Real -> ElectricalNode -> ElectricalNode -> Eqs>
```

Hence, an electrical branch is also an unknown defining a function model type. Consider again the definition inside `Resistor`. The first argument to `ElectricalBranch` is the unknown current, the second argument the unknown voltage. The third argument is the positive node and the fourth argument the negative node. Hence, by matching on the branch, we can decide if the branch is pointing with its positive or negative side towards a node. From this follows the third rule:

> **Rule 2 - Sum-to-zero equations:** For each node n in the circuit, create a sum-to-zero equation, such that the unknown flow variables for the branches connected to node n get a positive sign if the branch is pointing in the positive direction to the node, and a negative sign if it is pointing in the negative direction. For reference branches, the positive sign is always used.

In Figure 11.2, the direction of the branches are marked as arrows in the components. Using this approach, we generate two sum-to-zero equations for the example:

```
DC_i +. R_i = 0.
-.DC_i -. R_i +. G_i = 0.
```

In the ground model, we have an `ElectricalRefBranch`, defined as follows

```
let ElectricalRefBranch : <Real -> Real -> ElectricalNode -> Eqs>
```

The ref branch takes an unknown current and an unknown voltage as inputs, but only *one* node. Such a node can be seen as branch for directly accessing the reference values of node. In the case of the ground component, this unknown is reference in the sum-to-zero equation, but not in any other equations.

We have now two potential unknowns for the nodes `e1` and `e2` and two sum-to-zero equations. However, each of the components has two unknowns resulting in six unknowns that do not have any related equation. The first three of the missing equations are already defined explicitly in the model defining the components behavior. These are:

```
200. *. R_i = R_v
G_v = 0.
DC_v = 12.
```

Hence, we now have in total 8 unknowns and 5 equations. What equations are missing? The equations defining the voltage drop over the component:

> **Rule 3 - Relative potential equations:** For each branch in the circuit, create an equation stating the voltage drop between the unknowns defined for the connected nodes. For reference branches, state that the voltage drop is equal to the potential of the connected node.

Following Rule 3, we get

```
R_v = e1_v -. e2_v
DC_v = e1_v -. e2_v
G_v = e2_v
```

We now have 8 equation and 8 unknowns. Why did we get 8 equations while the Modelica compiler generated 14 equations? First, in Modelica, several unknowns are generated for the same potential variable, representing the same node. In this case, 4 unknowns were generated, instead of two that are needed in the MKL semantics, i.e., 2 more unknowns. Secondly, the current unknowns are both represented for the flow inside the component as well as through the connectors in Modelica. In total, this is 4 more unknowns and corresponding equations in Modelica. Note that the ground component includes the two unknowns in both cases.

After this informal introduction, we are now ready to formalize the connection semantics.

11.2.3 Formalization of the Connection Semantics

Let N be a finite set of nodes and $n \in N$ denote a node element. Let U be a finite set of unknowns and $u \in U$ an unknown. A branch is a quadruple $(u_f, u_{rp}, n_1, n_2) \in B_{bin}$, where u_f is a flow unknown, u_{rp} a relative potential unknown, n_1 a first and n_2 a second node connected to the branch. Let $(u_f, u_{rp}, n_1) \in B_{ref}$ be a reference branch, where u_f is the flow unknown, u_{rp} a relative potential unknown, n_1 a connected node. Set B_{bin} is the set of binary branches and B_{ref} is a set of unary reference branches. Let $B = B_{bin} \cup B_{ref}$ denote a set of all branches. Let the pair $(n, u_p) \in P$ denote the potential unknown associated with node n, and let P be the set of such pairs. We define an expression e and a list of equations E with the grammar rules

$$
\begin{aligned}
e &::= e + e \mid e - e \mid 0 \mid u \\
E &::= (e = e) \cdot E \mid \varepsilon
\end{aligned}
$$

where + and - are plus and minus operators, 0 the value zero, and u an unknown. The term $(e = e) \cdot E$ is the cons of an equation onto an equation list, ε is empty list.

Figure 11.3 defines the connection elaboration semantics as a set of recursive function definitions. Each of the functions is categorized according to the informal rules described in the previous section.

In Rule 1. unknown potentials are generated for each node. A function *ukpot* is defined, which takes a tuple as input, where the first element N is the set of nodes and the second element U a set of unknowns. The set U is used for generating unique unknowns. The function returns a set P, i.e., a mapping (n, u_p) between the nodes and the new unknowns representing potential values. The function is total when the guard $u \notin U$ is interpreted as generating a new fresh u that is not in the set U.

In Rule 2, a list of sum-to-zero equations are generated. It consist of one main function *sumzero* and one help function *sumexpr*. The function *sumzero* takes a tuple as input, where the first element N is the set of nodes and the second element B the set of branches. For each $n \in N$, the function creates the sum-to-zero expression using the help function *sumexpr*. The first three cases of the body consider binary branches by matching on the quadruple (u_f, u_{rp}, n_1, n_2). Only nodes that are directly connected to

Rule 1 - Unknown potentials $\boxed{\mathit{ukpot}(N,U)}$

$$
\begin{array}{lcll}
\mathit{ukpot}(\emptyset, U) & = & \emptyset & \\
\mathit{ukpot}(N, U) & = & \mathit{ukpot}(N - \{n\}, U \cup \{u\}) \cup \{(n,u)\} & \text{if } n \in N \text{ and } u \notin U
\end{array}
$$

Rule 2 - Sum-to-zero equations $\boxed{\mathit{sumzero}(N,B)}$

$$
\begin{array}{lll}
\mathit{sumzero}(\emptyset, B) & = \varepsilon & \\
\mathit{sumzero}(N, B) & = (\mathit{sumexpr}(n, B) = 0) \cdot \mathit{sumzero}(N - \{n\}, B) & \text{if } n \in N
\end{array}
$$

$\boxed{\mathit{sumexpr}(n,B)}$

$$
\begin{array}{ll}
\mathit{sumexpr}(n, \emptyset) & = 0 \\
\mathit{sumexpr}(n, B \cup \{b\}) = & \left\{
\begin{array}{ll}
\mathit{sumexpr}(n, B - \{b\}) + u_f & \text{if } (u_f, u_{rp}, n_1, n_2) = b \text{ and} \\
 & \quad n = n_1 \text{ and } n \neq n_2 \\
\mathit{sumexpr}(n, B - \{b\}) - u_f & \text{if } (u_f, u_{rp}, n_1, n_2) = b \text{ and} \\
 & \quad n \neq n_1 \text{ and } n = n_2 \\
\mathit{sumexpr}(n, B - \{b\}) & \text{if } (u_f, u_{rp}, n_1, n_2) = b \text{ and} \\
 & \quad ((n \neq n_1 \text{ and } n \neq n_2) \text{ or} \\
 & \quad (n = n_1 \text{ and } n = n_2)) \\
\mathit{sumexpr}(n, B - \{b\}) + u_f & \text{if } (u_f, u_{rp}, n_1) = b \text{ and } n = n_1 \\
\mathit{sumexpr}(n, B - \{b\}) & \text{if } (u_f, u_{rp}, n_1) = b \text{ and } n \neq n_1
\end{array}
\right.
\end{array}
$$

Rule 3 - Relative potential equations $\boxed{\mathit{relpot}(P,B)}$

$$
\begin{array}{ll}
\mathit{relpot}(P, \emptyset) & = \varepsilon \\
\mathit{relpot}(P, B \cup \{b\}) = & \left\{
\begin{array}{ll}
(u_{rp} = u_{p1} - u_{p2}) \cdot \mathit{relpot}(P, B - \{b\}) & \text{if } (u_f, u_{rp}, n_1, n_2) = b \\
 & \quad \text{and } (n_1, u_{p1}) \in P \\
 & \quad \text{and } (n_2, u_{p2}) \in P \\
(u_{rp} = u_{p1}) \cdot \mathit{relpot}(P, B - \{b\}) & \text{if } (u_f, u_{rp}, n_1) = b \\
 & \quad \text{and} (n_1, u_{p1}) \in P
\end{array}
\right.
\end{array}
$$

Connection elaboration $\boxed{\mathit{conelab}(N,B,E,U)}$

$$
\begin{array}{lcll}
\mathit{conelab}(N, B, E, U) & = & (E', U') & \text{if } \; P = \mathit{ukpot}(N, U) \\
 & & & \quad E' = E \oplus \mathit{sumzero}(N, B) \oplus \mathit{relpot}(P, B) \\
 & & & \quad U' = U \cup \{u \mid (n, u) \in P\}
\end{array}
$$

Figure 11.3: *Formalization of MKL's connection semantics.*

the considered branch are added to the expression. The last two cases handles reference branches in the same manner. Note that a 0 expression is inserted at the end of the recur-

sion. This zero expression can easily be eliminated by also introducing unary minus in the expression. However, it makes the definition less readable and is therefore avoided in this formalization.

In Rule 3, we generate the relative potential equations. In the electrical domain this corresponds to the voltage drop over a component. The function *relpot* takes a tuple as argument where the first element is a set P consisting of the tuple $(n, u_p) \in P$, where n is a node and u_p a potential unknown associated with the node. For each branch, we pick out the associated potential unknowns for the connected nodes. The list of equations is generated recursively.

Finally, the last function definition $conelab$ takes a quadruple as argument. The first element N is the nodes of the model and the second element B the branches. The third element E is a list of equations that already exist in the model, e.g., equations such as Ohm's law. The fourth element are the unknowns defined in the circuit, e.g., the current flowing through a component or the voltage drop over a component. The function returns a tuple consisting of the elaborated list of equations and all unknowns of the model. Note that we are using the symbol $\oplus$ for an infix left associated list append operator.

We define the following limitations on what can be considered valid input to the *conelab* function.

Proposition 11.1 (Valid input for connection elaboration)
For a valid input quadruple (N,B,E,U) holds

1. *If* $(u_f, u_{rp}, n_1, n_2) \in B$ *then* $\{u_f, u_{rp}\} \subseteq U$ *and* $\{n_1, n_2\} \subseteq N$
2. *If* $(u_f, u_{rp}, n_1) \in B$ *then* $\{u_f, u_{rp}\} \subseteq U$ *and* $n_1 \in N$

Note that nodes that are not connected to a branch are allowed as input, but will result in equations of form 0 = 0, which should be discarded.

11.2.4 Composition, and Multiple States

We will now discuss the connection semantics for when parts of a model are used to construct a new model. Consider Figure 11.4, which consists of three circuits. Figure 11.4a shows a simple circuit called `CircuitA`, where an inductor and a capacitor is connected in parallel, which are in turn connected in series with a resistor. These three components are composed into another model, illustrated in Figure 11.4c. The new model has two ports (or connectors) to which the internal components are connected. Finally, in Figure 11.4b, we show a new model named `CircuitB` where an instance of the new model is created and then connected. Hence, `CircuitA` and `CircuitB` models the exact same circuit, but abstracted in different ways.

Elaborating with Modelica

When elaborating `CircuitA` in Modelica, we get an equation-system containing 27 equations and 27 unknowns. However, when elaborating circuit `CircuitB`, we get 32 equations and 32 unknown. When we simulate the two circuits, we see that they get the exact same simulation result. Why are then 5 more equations generated for `CircuitB`

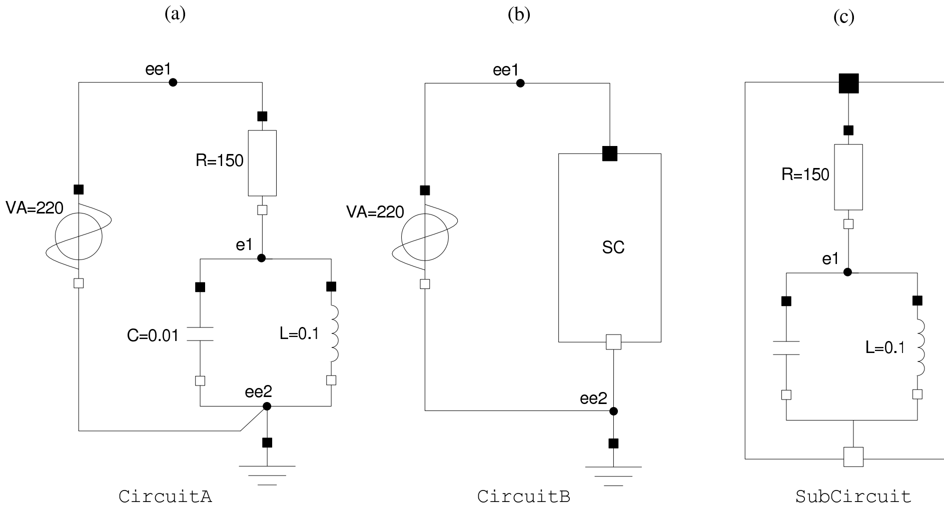


Figure 11.4: *Illustration how parts of a circuit can be composed into a new model abstraction. Figure (a) shows the full circuit and Figure (c) shows how three of the components are composed into a new model. Figure (b) shows how an instance of the model in (c) is used. At the end, the models in Figure (a) and Figure (b) model the same circuit.*

than `CircuitA`? The reason is that the abstracted model now contains two new connectors, each contributing with two unknowns. The last new unknown is the voltage drop over component M.

In `CircuitB`, the Modelica connect semantics need to take into consideration if connectors are *inside* or *outside* connectors. The same connector can both be an inside and outside connector, depending on if the current connection set are elaborated from the inside or the outside of a component. When `CircuitB` is elaborated, it starts by first elaborating its sub-components. When the `SC` component is elaborated, the connections between the components `R`, `C`, and `L` are elaborated. In the case of equations for potential variables, there is no difference, e.g., the equation between `SC`'s positive connector and the resistor is

```
SC.p.v = SC.R.p.v;
```

Similarly, between the resistor `R`, the inductor `L` and the capacitor `C`, two equations are generated to make the potential equal

```
SC.L.p.v = SC.C.p.v;
SC.R.n.v = SC.C.p.v;
```

For the sum-to-zero equations, it is a bit different. First, consider equation

```
SC.C.p.i+SC.L.p.i+SC.R.n.i = 0;
```

which is the sum-to-zero equation for the connection set between components `SC.R`, `SC.L`, and `SC.C`. All the connected connectors are here *inside* connectors because the connectors belong to components that are inside the model that is currently elaborated.

The opposite, an *outside* connector, is an connector that is connecting to the outside of the model. Consider the following equation:

```
(-SC.p.i) + SC.R.p.i = 0;
```

This is the sum-to-zero equation between component `SC`'s positive connector (an outside connector) and resistor `SC.R`'s positive connector (an inside connector). According to the Modelica specification, outside connectors shall have negative sign. The reason for this is to make the signs right when connecting the new component.

Elaborating with MKL

Now, let us consider `CircuitA` modeled in MKL:

```
let CircuitA =
  let ee1:Electrical in
  let ee2:Electrical in
  let e1:Electrical in
  Resistor 150. ee1 e1;
  Inductor 0.1 e1 ee2;
  Capacitor 0.01 e1 ee2;
  SineVoltage 220. 50. ee1 ee2;
  Ground ee2
```

When elaborating this model, we get 13 equations and 13 unknowns. This is approximately half of the 27 equations generated for `CircuitA` in Modelica. We simulate and check that the both circuits give the same behavior. The reason for the differences is again that MKL does not have unknowns in the ports.

Now, if we compose the three sub-components Resistor, Inductor, Capacitor into another model

```
let SubComponent p:Electrical -> n:Electrical =
  let e1:Electrical in
  Resistor 150. p e1;
  Inductor 0.1 e1 n;
  Capacitor 0.01 e1 n
```

and then use this in `CircuitB` we have the following model:

```
let CircuitB =
  let ee1:Electrical in
  let ee2:Electrical in
  SineVoltage 220. 50. ee1 ee2;
  SubComponent ee1 ee2;
  Ground ee2
```

When elaborating `CircuitB` we get 13 equations and 13 unknowns. Hence, for both model `CircuitA` and `CircuitB`, MKL gives 13 equations, but in the Modelica case, `CircuitA` gave 27 equations and `CircuitB` 32 equations. Why is not `CircuitB` generating more equations in the MKL case? Because the elaboration semantics of MKL does not generate new equations when models are composed. The reason for this is as follows:

Consider again model the `SubComponent`. The body of the model is almost exactly the same as line 4-7 in model `CircuitA`. The difference is only the name of the nodes. When the model `SubComponent` is instantiated in `CircuitA`, the nodes `ee1` and `ee2` are supplied to the function and then substituted into the body of `SubComponent`. Hence, after collapsing the instance hierarchy of the model, and before connection elaboration is performed, the models `CircuitA` and `CircuitB` are equivalent.

Models with Several States

In certain domains and for certain models the model should have different flow values on different sides of the component. In Modelica this is implicitly solved because the connectors have their own flow variables. However, because the corresponding MKL model only has one flow variable for each branch, which is given by the `ElectricalBranch`, it would perhaps be hard to model such domain models. Fortunately, this not the case.

A typical example when this occurs is when modeling rotational 1D mechanics. Example models are inertia and ideal gear. Let us study the latter a bit closer. An ideal gear can be modeled as follows, by following the conventions in the Modelica standard library:

```
let IdealGear ratio:Real -> flangeA:Rotational ->
              flangeB:Rotational -> Equations =
  let tauA:Torque in
  let tauB:Torque in
  let phiA:Angle in
  let phiB:Angle in
  RotationalRefBranch tauA phiA flangeA;
  RotationalRefBranch tauB phiB flangeB;
  phiA = ratio *. phiB;
  0. = ratio *. tauA +. tauB
```

In an ideal gear, both the torque and the angle at each side should be different. The linear relationship is given by the parameter `ratio` and the relation is modeled by the two last equations. Instead of having one branch over the component, we define two reference branches and thus generate unknowns for both `flangeA` and `flangeB`. Recall the connection semantics in Section 11.2.3. In the case of a reference branch in the mechanical domain, the angles `phiA` and `phiB`, which are the potential variables, will be equal to the connected node. Hence, `phiA` and `phiB` will be the absolute angles for `flangeA` and `flangeB`. The flow variable in the reference case, i.e., the torques `tauA` and `tauB` represent the flow at each side of the component.

11.2.5 Executable Specification

In MKL, intensional analysis of models can be used to inspect the equation system. Also, new models can be created by synthesizing new models. Hence, we can transform a model A into a model B according to some transformation rules.

We will now demonstrate this principle by creating an executable specification of the connection semantics. The elaboration semantics is defined in the standard library file `elaboration.mkl` and can be found in Appendix D.7. The full executable specification corresponding to the definition in Section 11.2.3 fits approximately into two and a

half page of source code. Hence, we would also like to demonstrate that the functional style of transforming a model can be considered as an expressive alternative.

The implementation is pure functional and declarative with the exception of the generation of unknowns, which is part of the MKL semantics. There are also some differences in this executable specification compared to the formal definition in Section 11.2.3. The most important one is performance. Implementing the formal definition as defined in the previous section is possible, but generates a very slow solution that does not scale. The main reason is that for each node $n \in N$ in the function *sumzero*, the function *sumexpr* is called and then recursively applied for each $b \in B$. Hence, we can directly say that the asymptotic lower bound is $\Omega(|N||B|)$ which is not scalable for an executable solution.

We will now walk through the source code for the connection semantics function by function. First, consider the header of the main function:

```
let elaborateConnections isBranch:(<> -> Bool) ->
                         isRefBranch:(<> -> Bool) ->
                         model:Equations ->
                         Equations =
```

The function `elaborateConnections` takes as its first two arguments two predicate functions, used to check if a certain branch is a branch or not. For example, when used in the mechatronic domain, the first function that could be supplied is

```
let isMechatronicBranch b:<> =
  match b with
  | 'ElectricalBranch -> true
  | 'RotationalBranch -> true
  | _ -> false
```

By separating out this predicate, the function `elaborateConnections` can easily be used for different domains. The third formal parameter is a `model` of type `Equations`. We know that a system of equations is a MKL model and that we therefore can inspect this data. The return type is also `Equations`, hence we define a model transformation, where both the source and target models are of the same type.

The function `elaborateConnections` contains several local functions. Consider the first helper function:

```
  let addNode node:Node -> nodemap:NodeMap -> NodeMap =
    if Map.mem node nodemap then nodemap
    else let u:<Real> in Map.add node u nodemap
  in
```

which uses type aliases[3]

```
type Node = <>
type Unknown = <>
type NodeMap = (Node => Unknown)
```

The function `addNode` takes a node as input and if it does not exist as a key in the node map, it is added with a new fresh unknown of type `<Real>`. This function will be used when looking up nodes in the model and then to create unknown potentials for that node

[3]Type aliases does not give any improved type checking, but makes the code more readable.

(recall Rule 1). The function also makes sure that if an unknown potential is already created for the node, it will not create a new one.

The next function defines the generation of sum-to-zero expressions

```
let sumexpr branches:BranchSet -> ExprMap =
  let worker branches:BranchList -> emap:ExprMap -> ExprMap =
    match branches with
    | (b i v p n)::bs when isBranch b ->
        let emap1 = if Map.mem p emap
                    then Map.add p ((Map.find p emap) +. i) emap
                    else Map.add p i emap in
        let emap2 = if Map.mem n emap1
                    then Map.add n ((Map.find n emap1) -. i) emap1
                    else Map.add n (-. i) emap1 in
        worker bs emap2
    | (b i v p)::bs when isRefBranch b ->
        let emap1 = if Map.mem p emap
                    then Map.add p ((Map.find p emap) +. i) emap
                    else Map.add p i emap in
        worker bs emap1
    | [] -> emap
  in worker (Set.toList branches) (Map.empty)
in
```

Five new type aliases are used in this function

```
type Expr = <>
type Branch = <>
type ExprMap = (Node => Expr)
type BranchSet = (Set Branch)
type BranchList = [Branch]
```

The function `sumexpr` takes a set of branches as input and returns a map where the key is a node and the value is an expression. The local function `worker` takes a list of branches and the expression map (node to expression) and creates a new expression map. The function implements *sumexpr* in Rule 2 for sum-expressions, i.e., it matches either a binary branch (first match case) or a reference branch (second match case). Note that the model function `b` is here a pattern variable and the check that it is a branch is done in the **`when`** guard, by calling the predicate function `isBranch`. The function `isBranch` is a formal parameter to the main function `elaborateConnections`. Note that this variable is accessible because `sumexpr` is a local function to `elaborateConnections`.

The pattern variable `i` is the flow unknown and `v` is the relative potential unknown. We use the naming convention for the electrical domain here, but the elaboration function is not specific to that domain. The last pattern variables `p` and `n` are the positive and negative nodes connected to the branch.

The main activity that the match cases do is to first check if an expression exists for the node. If it exists, a new expression is created, where the current is added

```
Map.find p emap) +. i
```

in the case when the node is connected to the positive side of the branch, and subtracted

```
(Map.find n emap1) -. i
```

in the case when it is connected to the negative side of the branch. Note that because both the value returned from `(Map.find n emap1)` and variable `i` are model types, this whole expression will be a model type.

Note also that the insertion of new expressions is purely functional, i.e., we use the ADT `Map` which implements a declarative map container. Insertion and lookup of values for the `Map` ADT has $O(logn)$ complexity, where n is the number of elements in the map.

The actual generation of sum-to-zero equations is performed by the `sumzero` function:

```
let sumzero m:Equations -> branches:BranchSet -> Equations =
  let worker elist:ExprList -> Equations =
    match elist with
    | (_,e)::es -> e = 0.; worker es
    | [] -> m
  in worker (Map.toList (sumexpr branches))
in
```

The function `sumzero` calls `sumexpr`, generates a list of expressions and then creates sum-to-zero equations directly. An informal analysis gives the asymptotic upper bound complexity for `sumzero` to be $O(|B|\ log|B|)$, where $|B|$ is the number of branches. The improvement, compared to the formal definition in Section 11.2.3 is that the function `sumzero` only performs one pass over the branches and during this pass inserts and adds elements to a map at $O(log\ |B|)$ time. We know that the size of the expression map returned from `sumexpr` is smaller or equal to the size of the branch set called `branches`, because each step in the recursion in `sumexpr` is at most adding one element. Hence, the iteration over the expression list in `sumzero` does not make the complexity worse.

The last local function for the connection elaboration is the function `potentials`:

```
let potentials model:Equations -> (Equations,BranchSet) =
  let worker m:Equations -> nodemap:NodeMap ->
             branchset:BranchSet -> (<>,NodeMap,BranchSet) =
    match m with
    | b i v p n when isBranch b ->
      let nodemap2 = addNode n (addNode p nodemap) in
      let eq = (v = (Map.find p nodemap2) -.
                (Map.find n nodemap2)) in
      (eq,nodemap2,Set.add m branchset)
    | b i v p when isRefBranch b ->
      let nodemap2 = addNode p nodemap in
      let eq = (v = (Map.find p nodemap2)) in
      (eq,nodemap2,Set.add m branchset)
    | e1 ; e2 ->
       let (e1b,nodemap1,branchset1) =
             worker e1 nodemap branchset in
       let (e2b,nodemap2,branchset2) =
             worker e2 nodemap1 branchset1 in
       (e1b ; e2b, nodemap2, branchset2)
    | _ ->  (m,nodemap,branchset)
  in
```

```
      let (model,_,branchset) =
        worker model (Map.empty) (Set.empty)
      in (model,branchset)
  in
```

The function `potentials` generates both new potential unknowns for the nodes (Rule 1) and adds relative potential equations (Rule 3). The local function `worker` recursively traverses the equation system (third case with pattern `e1 ; e2`). The first two match cases match a binary branches and reference branches respectively. The function `addNode` is now used to add both the positive and the negative nodes to the node map (described earlier). The values of this map are the new potential unknowns. The only equations that reference the potential unknowns are the relative potential equations generated in the first two match cases, e.g., the line

```
let eq = (v = (Map.find p nodemap2) -.
          (Map.find n nodemap2)) in
```

from the first match case. We create a new equation bound to the variable `eq`. The new equation includes the unknowns looked up in `nodemap2`, which are associated to the corresponding nodes.

The last part of function `elaborateConnections` makes use of the locally defined functions described above:

```
let (model2,branchset2) = potentials model in
sumzero model2 branchset2
```

First, function `potential` is applied to the model supplied to the main elaboration function. A transformed model is returned, which includes new relative potential equations together with unknown potentials for the nodes. Function `potentials` also returns the branchset for the model. This set is then later used in function `sumzero` for generating the sum-to-zero equations. Hence, only function `potential` needs to traverse the equation system. The function `sumzero` adds the new sum-to-zero equations on top of the equation system of `model2` and returns the final system of equations.

11.3 Extracting Model Information

A mathematical model can be used for several different things. The application area that we have discussed mostly in this thesis is for simulation. However, if a model after elaboration consists of thousands of equations, there must be a strategy for how to extract for example the simulation results of a particular variable, e.g., for the purpose of plotting. We call this the *model information extraction problem*.

11.3.1 Hierarchy Naming vs. Probing

There are several approaches for solving the model information extraction problem, which all have their pros and cons. We see two main alternatives for extracting information as the result of using a model:

1. *Hierarchy naming* where all unknowns in the model have a unique textual name that can be used for identifying the corresponding unknown.

2. *Probing* with a language construct for "probing" information from particular unknowns in the model. The probes are added into the model.

The first approach, hierarchy naming, is used in, e.g., Modelica. For example, assume that a main model `Circuit` has as a sub-component a resistor component called `R`. A resistor model has a variable called `v`, which defines the voltage drop over the component. Hence, when the model is elaborated into an equation system, the unknown for the voltage drop has the name `Circuit.R.v`. Hence, Modelica tools, such as Dymola [45] or MathModelica [91] support a tree browser view after simulation, where the user can click on different variable names to inspect the simulation result.

The alternative approach, which we call *probing* is when an element, a "probe", is added to the model, which indicates which unknown should be extracted. This could be done both visually using a GUI or textually, by inserting a language statement or expression. For example, MapleSim [90] is using this approach.

We see both pros and cons with both alternatives. With the hierarchy naming approach, each unknown has always a unique name that can be understood by a user. However, for large models with thousands of equations and several levels in the hierarchy, it can become fairly hard to get an orientation by looking at variable names in a tree view. The main problem with this approach is that the tree view for inspecting the variable names is not the same view as the one for the source code because it is created according to the instance hierarchy. However, the pros with this approach is that the model does not need to be altered before simulation. Moreover, the decision of which variables that should be inspected can be postponed to after the simulation.

The alternative of using probes forces the user to modify the model before simulation. However, with good software tool support, this does not necessarily mean that it is cumbersome for the user. A benefit with this solution is that the user has the same view for both modeling and the variables that should be inspected. In the simplest case, probes need to be inserted before simulation. However, if the model is compiled in a kind of debug mode, it might be possible that these probes can be added after simulation, and thus making use of the simulation result already available.

A major difference of the two approaches is how *open* the language's internal form should be to the user. To enable hierarchy naming, the names used in the model must be preserved during simulation. Comparing to ordinary program languages and compilers, this means that the model must be compiled in debug mode where symbol names are stored. Moreover, when making use of anonymous functions, it is less clear how this can be achieved because no names are available at modeling time. An analogy for probes and programming is that probes can be seen as a way of creating traces in the model, i.e., to decide in the model what information that should be printed out. Another difference is their ability to handle *variable structured systems*, i.e., when the number of equations and variables change over time. Obviously, this poses new challenges for model information extraction because the set of unknowns cannot be decided upon compile time.

11.3.2 Modeling with Probes in MKL

The choice of how to extract information from a model is not encoded into the language. Instead, there are standard libraries which define the way information can be extracted.

Direct hierarchy naming is difficult to accomplish in MKL because in the same way as in other functional languages, the actual identifier name is not important for the computation result. Internally, the compiler and runtime system can be using indexes or pointers for variable lookup.

Let us consider the `LotkaVolterra` model again:

```
let LotkaVolterra =
  let growthRateRabbits = 0.04 in
  let deathRateRabbits = 0.0005 in
  let deathRateFoxes = 0.09 in
  let efficiencyGrowthFoxes = 0.1 in
  let rabbits:Population in
  let foxes:Population in
  Init rabbits 700.;
  Init foxes 10.;
  der(rabbits) = growthRateRabbits *. rabbits -.
                 deathRateRabbits *. rabbits *. foxes;
  der(foxes)   = efficiencyGrowthFoxes *. deathRateRabbits *.
                 rabbits *. foxes -. deathRateFoxes *. foxes
  probe "foxes" = foxes;
  probe "animals" = foxes +. rabbits
```

We have now added two more equations, both with one probe each. The probe construct is not built into the language. It is defined in the standard library `modeling.mkl`:

```
let probe : <String -> Real>
```

Hence, a probe is an unknown of a function model type. The function takes as input a `String`, and returns as result a `Real`. The string is the name that then later can be used to identify the unknown, e.g., when plotting the simulation result. Note that we in the probe `"animals"` also are probing the value that is the result of adding foxes and rabbits.

In the `LotkaVolterra` example, the probes were explicitly referencing an expression using an equation. However, for probing the information from components within a certain domain, is should be easy to access the information relevant for that domain. Now, let us consider the `MechSys` model again:

```
let MechSys =
  let r1:Rotational in
  let r2:Rotational in
  let r3:Rotational in
  DCMotor r1;
  Inertia 0.2 r1 r2;
  FlexibleShaft 120 r2 r3;
  SpeedSensor (probe "omega") r3
```

In the example, we are using a *sensor* for extracting the wanted information. These sensors are themselves models and can be used for example to extract speed information, and then use this information in a feedback loop for a control system. Such sensor models are also available in the Modelica standard library, and we have modeled here the same behavior. The definition of the speed sensor is

```
let SpeedSensor w:Signal -> flangeB:Rotational -> Equations =
  let phi:Angle in
  RotationalRefBranch 0. (-.phi) flangeB;
  w = der(phi)
```

Recall the definition of the `Signal` type

```
type Signal = <Real>
```

i.e, the `SpeedSensor` gives an output signal of the angular velocity for the node, by differentiating the signal `phi`, i.e., the angel. Recall the line

```
  SpeedSensor (probe "omega") r3
```

of the `MechSys` model. Here we measure the speed of node `r3`, which will be the last node of the 120 elements shaft. We supply a probe called `"omega"` to the speed sensor, which results in that we can measure the angular velocity and associate it with the name `"omega"`.

11.3.3 Elaboration Semantics of Probes

The semantics for how probe information is extracted from a model is defined in the standard library file `elaboration.mkl` in Appendix D.7. The source code is listed below:

```
type ProbeMap =  (String => [Signal])

let addProbe s:String -> u:Signal -> ps:ProbeMap =
    if Map.mem s ps then Map.add s (u::(Map.find s ps)) ps
    else Map.add s [u] ps

let elaborateProbes model:Equations -> (Equations,ProbeMap) =
  let elab e:<> -> ps:ProbeMap -> (<>,ProbeMap) =
    match e with
    | 'probe (val s:String) -> let u:Signal in (u,addProbe s u ps)
    | e1 e2 ->
       let (e1b,ps1) = elab e1 ps in
        let (e2b,ps2) = elab e2 ps1 in
       (e1b e2b,ps2)
    | _ -> (e,ps)
  in elab model (Map.empty)
```

The function `elaborateProbes` takes an equation system as input and returns a tuple with the updated equation systems and a mapping of probes. The type `ProbeMap` is a mapping between the string values and a list of signals. The strings are the names given for the probes in the model, and the signals are the unknowns that are associated with the corresponding probes. Because a model can contain probes with the same names (e.g., several instances of a model containing a probe with a specific name), we need to associate a probe name with a list of signals and not just one single signal.

The local function `elab` recursively traverses the model, both equation systems and expressions. When a probe is matched (the first match case), we extract the string value

`s`. Then, we create a new unknown `u`, which is replacing the probe, i.e., the transformed model will have unknowns where the probes were located.

Recall that in the `LotkaVolterra` example, we introduced new equations, but no new explicit unknowns. The unknown is therefore introduced by the probe itself. Similarly, for the `SpeedSensor`, the probe will introduce a new unknown and supply it as an argument to `SpeedSensor`. If we are counting equations and unknowns for the `SpeedSensor`, we will see that it is over-determined, i.e., it has one equation too much. The unknown from the probe restores the balancing.

The actual generation of the probe mapping is handled by function `addProbe`. This probe mapping, which is one of the outputs from `elaborateProbes`, can then be used by other transformations. For example, if we want to code generate/export the equation system into another target language, e.g., a flat Modelica form, this probe map can be used for giving names to variables that should be easily accessible. On the other hand, if the simulation should be conducted by a library in MKL, the probe mapping can be used for giving names of the output data of the simulation. Hence, the aim of making the probe functionality as a separate function is to enable reuse and extensibility of the library.

11.4 Chapter Summary and Conclusions

We have in this chapter given both an informal overview of the elaboration process in MKL and also formally specified both the connection semantics and how to extract information out of models using probes. We have also given an informal explanation of the connection semantics in Modelica.

We can see that the connection semantics in Modelica and MKL are different, but give similar modeling capabilities. Semantically, we argue that the MKL approach is simpler because it separates the phases of type checking, collapsing the model hierarchy, and generation of connect equations into distinct phases.

From a modeling point of view, it is of course very subjective which approach is preferable, e.g., if `connect`-equations should be used, or if connections are defined by using nodes. The benefits that we see from a theoretical language point of view is that the nodes semantics are very natural and fit well into the framework of a typed functional language based on an effectful extension of the lambda calculus.

12

Implementation, Verification, and Evaluation

In Part II we illustrated different aspects of MKL, including several application examples, metaprogramming examples, and formal semantics. In this chapter, we discuss, verify, and evaluate our solution in relation to the problem area presented in Section 1.3. The chapter is structured as follows:

- We give a brief overview of the *prototype implementation* of MKL and discuss its current capabilities and limitations (Section 12.1).
- We briefly explain the implementation of two ways of using the model (Section 12.2).
- We explain how the *verification* of our solution has been performed (Section 12.3).
- We *evaluate*, discuss, and analyze our approach according to the areas presented in the problem area: safety aspects, expressiveness aspects and extensibility aspects. We also briefly discuss some performance aspects of the prototype implementation (Section 12.4).

12.1 Implementation

The MKL prototype is implemented as an interpreter in OCaml 3.11.2 [74]. The aim of the prototype at this stage is not to be a full fledged simulation environment that can be used directly in industry. Instead it is a research prototype used for exploring different ways of using metaprogramming for EOO languages.

Consider Figure 12.1, which shows a box-and-line diagram for the execution view for the architecture of the implementation.

A `.mkl` file is given as input to the left in the figure showing the translation process. It is translated in a number of sub-phases, followed by evaluation (execution) of the program. If there were no errors during translation and execution, the program outputs its

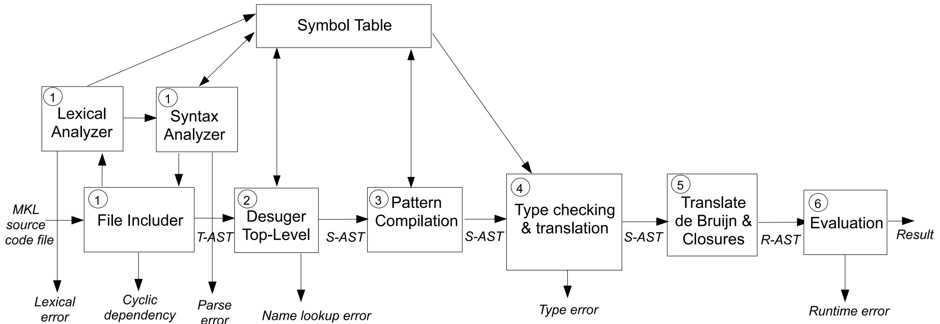


Figure 12.1: *Outline of the architecture for the MKL prototype implementation.*

results and terminates. We call the phases (1) - (5) the *static semantics*. We say that the last step (6), evaluation, is described by the *dynamic semantics*.

12.1.1 File Includer and Symbol Table

The first phase, the *file includer*, is a simple module system for handling `include` statements. The file includer first performs lexical and syntax analysis on the input file. The top-level statements in the file are inspected, one of which is the include statement. For each include statement, this procedure is recursively performed. The file included by the include statement is read and lexical and syntax analysis is performed. The process detects and reports cyclic dependencies between definitions as well as eliminates all duplications of the top level definitions.

During the lexical analysis new identifiers are detected and added to the global *symbol table* (illustrated at the top of Figure 12.1). A unique relationship is created between the Unicode string of the identifier and an integer value. Two hash tables are defined, one mapping strings to integers, the other one mapping integer values to strings. During the static semantics phases, the integer value is used for fast access and comparison. If an error is detected (the arrows at the bottom of the Figure 12.1) a user message is written to the standard output, where the string representation of identifiers are looked up in the symbol table.

12.1.2 Desugaring

The output from the file includer phase is a top-level AST, denoted T-AST. The T-AST contains the following four top elements:

- `let`-bindings where a value is bound to a value. This includes top-level function definitions. For example,

```
let double x:Int = x * 2
```

- New definition of unknowns. These are constructed in the syntax analyzer when parsing `let`-expressions of form `let` x`:`T `in`, where x is the new name and T its type. For example, the definition of an equation with an unknown equation value

  ```
  let Eq : <Real -> Real -> Eqs>
  ```

- New type declarations of the form `type` x. For example, the definition of electrical nodes:

  ```
  type ElectricalNode
  ```

- Type aliases of the form `type` x`=` T, where x is the new alias for type T. For example, the type alias for signals

  ```
  type Signal = <Real>
  ```

Note that `include` elements are also top-level elements, but these are removed and processed in the previous phase by the file includer.

Phase (2), called *desugar top-level*, translates the T-AST into a *static analysis AST*, denoted S-AST. Top-level **`let`**-style bindings are translated into local definitions, i.e., using the `in` form. Also, new types and type aliases are substituted in all terms following the definition, so that the all top-level elements are eliminated and an S-AST is returned as the result.

During this phase, name environments for both types and terms are used when traversing the AST. If a name is not found, a compile time error is reported to the user. The symbol table and information annotations on the terms are used for good error reporting, i.e., that the original names are used and line and row number of the error is reported.

The third phase, called *pattern compilation*, are **`match`**-expressions translated into primitive constructs of the S-AST. The procedure for the pattern compilation is standard and is implemented according to the techniques explained by Peyton Jones and Wadler [122].

12.1.3 Type Checking and Model Translation

During phase (4), S-AST terms are type checked according to the type systems presented in Chapter 10. Because the types of formal parameters are explicitly stated in the program, the type checking can be directly performed in a bottom-up fashion on the AST. During type checking, the AST is also translated so that unknowns cannot be accidentally evaluated, i.e., terms are lifted to become models, as explained in Chapters 9 and 10.

12.1.4 Program Evaluation after Translation

In phase (5) of the static semantic analysis the S-AST is translated to a runtime AST, denoted R-AST. In the S-AST, name bindings have been handled by using names and named environments. Before evaluation, the AST is translated into a nameless representation of terms using a technique invented by *de Bruijn* in 1972 [46]. Also, the R-AST also includes closure terms, i.e., terms that contains both a term and its environment. The evaluation (interpretation) of the program is now performed using a recursive function defined in OCaml.

12.2 Uses of Models

One of the main objectives with the MKL approach is to make it possible to use models in different ways, and that the semantics for the uses is implemented in libraries. We will now briefly explain two cases of using the models:

- Exporting the DAE to flat Modelica
- Simulate the DAE

12.2.1 Exporting the DAE to Flat Modelica

After elaborating a model to a DAE, it could sometimes be useful to export the model into another format. Specific library functions could be written for different target languages, such as XML or flat Modelica. In this case, we will exemplify the latter.

The whole export source code can be found in the library file `exportmodelica.mkl`, available in Appendix D.10.

The aim of the export function is to generate a flat Modelica file, i.e., a Modelica source code file that only contains equations and variables. For example, if the export function is executed on the `LotkaVolterra` model presented in previous sections, the output is as follows:

```
model LotkaVolterra
  output Real animals = uk3;
  output Real foxes = uk2;
protected
  Real uk3;
  Real uk2;
  Real uk1(start=10.,fixed=true);
  Real uk0(start=700.,fixed=true);
equation
  der(uk0) = (0.04 * uk0 - 0.0005 * uk0 * uk1);
  der(uk1) = (5e-05 * uk0 * uk1 - 0.09 * uk1);
  uk2 = uk1;
  uk3 = (uk1 + uk0);
end LotkaVolterra;
```

In the protected part of the generated model the four unknowns are named `uk0`,`uk1`,`uk2`, and `uk3`. These names are generated by the export functions because unknowns can only be compared for equality and match the type. The first two equations are the equations stating the LotkaVolterra dynamics, while the last two equations were given due to probing. At the top, two `output` variables are given. In this exportModelica function, we use the probe names to name these output variables. Note also that we use the `Init` equations for generating Modelica's syntax for start attributes.

Consider the following listing of the function for prettyprinting an expression.

```
  let pprintExpr expr:<> -> String =
    match expr with
    | e1 +. e2 -> "(" ++ pprintExpr e1 ++ " + "
                        ++ pprintExpr e2 ++ ")"
```

```
| e1 -. e2 -> "(" ++ pprintExpr e1 ++ " - "
                    ++ pprintExpr e2 ++ ")"
| e1 *. e2 -> pprintExpr e1 ++ " * " ++ pprintExpr e2
| e1 /. e2 -> pprintExpr e1 ++ " / " ++ pprintExpr e2
| -. e ->     "(-" ++ pprintExpr e ++ ")"
| e1 ^. e2 -> pprintExpr e1 ++ " ^ " ++ "("
                     ++ pprintExpr e2 ++ ")"
| 'der e ->   "der(" ++ pprintExpr e ++ ")"
| 'sin e ->   "sin(" ++ pprintExpr e ++ ")"
| 'cos e ->   "cos(" ++ pprintExpr e ++ ")"
...
| 'log e ->   "log(" ++ pprintExpr e ++ ")"
| 'log10 e -> "log10(" ++ pprintExpr e ++ ")"
| 'time -> "time"
| uk:<Real> -> mkvar uks expr
| val r:Real -> real2modelicaString r
```

Each case matches a certain form of expression. For example, we use the infix notation of matching binary operators. For matching application to a function, we use the pattern expression starting with an apostrophe. For example, the line:

```
| 'log e ->   "log(" ++ pprintExpr e ++ ")"
```

is equivalent to

```
| x e when x == log ->  "log(" ++ pprintExpr e ++ ")"
```

The line matching an unknown is calling a function `mkvar` which takes a data structure representing all unknowns as its first formal parameter, and this particular unknown as its second. All unknowns in the circuit have been given a unique integer value. Hence, `mkvar` looks up the integer value for `expr` and prints out the string.

The rest of the source code in Appendix D.10 should be self explaining.

12.2.2 Simulating the DAE

The second example of using a model is to simulate the model using an external DAE solver. However, the semantics for how to extract the equations, iterate over the simulation interval, store data, etc., is carried out in a MKL library. The standard library file `simulation.mkl` in Appendix D.9 shows a listing of the source code.

The main simulation function is called `simulate`:

```
let simulate model:Equations -> steptime:Real -> endtime:Real ->
             SimulationResult =
  let (model2,probes) = elaborate model in
  let probelist = Map.toList probes in
  let ukmap = makeUkMap model2 in
  let residual = makeResidual model2 ukmap in
  let (yy,yp) = makeInitValues model2 ukmap in
  let id = initConditionCorrection ukmap in
  let state = DAESolver.make yy yp id residual in
  let simloop currtime:Real -> acc:[StepVal] -> [StepVal] =
    if currtime >=. endtime then acc
```

```
    else
      let stepval = makeStepVal currtime yy probelist ukmap in
      let newtime = DAESolver.step steptime state in
      if newtime == 0. then acc
      else simloop (newtime +. steptime) (stepval::acc)
  in
    (makeProbeNames probelist 1,revResult (simloop 0. []) [])
```

This function takes as input a model, the step size, and the end time of the simulation. First, it elaborates the model and gets a new `model2` together with the `probes` mapping. The sixth line calls `makeResidual` for generating the residual function for the DAE. The residual is a list of expressions, where each expression represents the difference between the left hand side and the right hand side of each equation. The residual function is a callback function that is called when initiating the external solver, using `DAESolver.make`. The residual function itself should have type

```
type Residual = Real -> {Real} -> {Real} -> [Real]
```

where the first argument is the current time, the second argument is a `Real` array with values for the unknowns, and the third an array with values for the differentiated unknowns. The function that generates the residual is

```
let makeResidual model:Equations -> ukmap:UkMap ->
                 ctime:Real -> yy:{Real} -> yp:{Real} -> [Real] =
  let realExpr e:<> -> Real =
    match e with
    | (val f:(Real -> Real -> Real)) e1 e2 ->
          f (realExpr e1) (realExpr e2)
    | (val f:(Real -> Real)) e1 -> f  (realExpr e1)
    | val v:Real -> v
    | 'time -> ctime
    | der x ->(match x with
               | uk:Signal -> Array.get yp (fst (Map.find x ukmap))
               | _ -> error "Derivatives only allowed on unknowns")
    | uk:Signal -> Array.get yy (fst(Map.find e ukmap))
    | _ -> error "Unsupported model construct"
  in
  let traverseEq m:Equations -> acc:[Real] -> [Real] =
    match m with
    | e1 ; e2 -> traverseEq e2 (traverseEq e1 acc)
    | e1 = e2 -> (realExpr e1 -. realExpr e2) :: acc
    | _ -> acc
  in traverseEq model []
```

When creating the residual, the function `makeResidual` is partially applied to the equation system and an `ukmap` function. The rest of the function should be fairly straightforward.

For details, see the full source code listening in Appendix D.9.

Test	Name	Domain	MKL Method	MKL Equations	Modelica Equations
1	SimpleCircuit	Electrical	S/T	23	46
2	ComposedCircuitA	Electrical	S/T	18	37
3	ComposedCircuitB	Electrical	S/T	18	42
4	DriveLine	Mechanical	S/T	42	53
5	Gear	Mechanical	S/T	30	41
6	GearDamper	Mechanical	S/T	45	57
7	OneTorque	Mechanical	S/T	16	21
8	TwoTorques	Mechanical	S/T	20	32
9	SineInertiaDamper	Mechanical	S/T	40	48
10	LotkaVolterra	Plain DAE	S/T	6	3
11	Mechsys120	Mechatronic	T	1588	2922

Table 12.1: *Experimental results when modeling and simulating different systems using the modeling kernel language compared to the Modelica tool Dymola. In the column "method": S = simulated with MKL, T= translated to flat Modelica from MKL.*

12.3 Verification

Different aspects and features of MKL have been discussed in previous chapters, but how do we know that the semantics is correct? Actually, we shall first ask ourselves by what we mean by correct semantics. Because the semantics of the language is the definition of the language, in what aspect can it be correct or incorrect?

One aspect is the relation between the static semantics and the dynamic semantics, i.e., to prove or justify type safety of the language. Even though this is a strong property, it does not guarantee that programs and models behave as expected.

Another part of the semantics is the elaboration semantics with the connection semantics for elaborating acausal connections. In what way can we prove or justify that this semantics is correct?

Our approach to justify the correctness of the semantics with regards to what a user can expect of such asystem is to verify the system against a state of the art solution. We have chosen to view the way of modeling in Modelica as the specification and then verify that we can model the same model components in MKL and that they give the same simulation results. This approach can be viewed as a high level system test, where we tests the following aspects of the system

- The dynamic semantics for collapsing the instance hierarchy, i.e., that the semantics for models and unknowns works as expected (semantics described in Chapter 10).
- The connection semantics for analyzing the equation system and generating connection equations (semantics described in Chapter 11).

Consider Table 12.1, which shows the tests that have been performed.

Tests are performed in the electrical domain, mechanical domain, and the combination, the mechatronic domain (see column three). A standard library for both the mechanical and the electrical domain has been created, where a portion of the continuous-time

model components of Modelica's standard library components are modeled in MKL (see Appendix D for a list of the model part of the library).

Column four of Table 12.1 titled *method* states how the verification was conducted. The letter T stands for *translation*. Each such test follows this procedure:

1. The model is created in Modelica using standard components in Modelica standard library. Sensor components are inserted into the model and connected for later inspection.

2. The same model is created by using components from MKL's standard library. This library has been modeled according to the Modelica library. Sensor components are inserted into the model in the same way as in the Modelica model.

3. The Modelica model is simulated using Dymola 6 [45]. Data from the sensors are plotted and visualized. The number of equations after elaboration were read out from the translation log and added to column 6 of Table 12.1.

4. The MKL model was translated and exported to a flat Modelica file using the MKL transformation code in Appendix D.10. The number of equations is counted and the result value is given in column 5 in Table 12.1[1].

5. The flat Modelica file generated in the previous step is simulated using Dymola 6. The sensor variables are plotted and visualized.

6. The plotted results from the Modelica model and the translated Modelica model are visually compared.

These tests have verified the elaboration semantics and the capabilities for intensional analysis of models. Tests also marked with the letter S which stands for *simultation* also verifies the MKL simulation implementation. For the simulation verification, the first three steps of the above were performed. This was followed by the following steps:

1. The MKL model is simulated using the MKL library `simulation.mkl`, which is listed in Appendix D.9. The underlying numerical DAE solver is IDA from the SUNDIALS solver suite [68]. The result of the sensor variables is plotted using GNU Plot v3.8h.

2. The plotted result from the MKL and the Modelica simulations were compared.

All tests, except test 11, are performed using both MKL simulation and translation to Modelica. In case 11, the IDA SUNDIALS solver could not find a consistent initial solution. Because we have not implemented Pantelides algorithm [118] for index reduction, and the IDA Sundials solver can find consistent initial conditions for so-called semi-explicit index-one problems [68], we believe that test 1 is a higher-index problem. However, the test with translation gave the same simulation result.

[1]The equation count includes equations for sensors and probes. This is the reason that the number can mismatch compared to other counting of equations stated earlier in this thesis. Note also the exceptional test case 10, where MKL has more equations than in Modelica. The reason is that probes give extra unknowns and thus also equations. These equations for probes is also the reason why the model becomes a DAE instead of an ODE.

12.4 Discussion and Evaluation

In the following subsections we will discuss and evaluate the MKL approach according to the following aspects introduced in Section 1.3:

- Safety Aspects
- Expressiveness and Extensibility Aspects

We will also briefly discuss performance aspects regarding the current prototype implementation.

12.4.1 Safety Aspects

In the problem area discussion, we described three main challenges regarding error handling:

- Detecting the existence of an error early.
- Isolating the fault implied by the error.
- Guaranteeing that the faults do not exist.

Because MKL is a language for defining EOO construct using defining standard libraries, error detection and fault isolation can be considered from several different perspectives.

System Modeling Errors

System modeling errors can be introduced by the modeler by entering wrong or illegal equations. The type system of MKL can detect and isolate certain kinds of errors, such that nodes from different domains do not mismatch. We regard this early static checking and fault isolation as very beneficial from a modeling perspective, compared to if the language would have been dynamically typed. The static checking in this regards can be seen as similar to the one in Modelica, but the difference is that in MKL new language constructs (consider e.g., probes or initial values) in almost all cases can be added to the language using libraries. Even these new constructs will be type checked at compile time.

However, several modeling errors will still not be detected by the type checker, e.g. constraint errors or physical unit of measurements errors. We have in earlier work studied both of these properties in the context of Modelica, but not yet for MKL, i.e., the constraint delta approach is not yet adapted and implemented for MKL. However, initial observations have been made that the constraint delta approach could be made much simpler by using MKL's connect semantics. Consider the following model of an inductor, where we have printed out the constraint delta value for each part:

```
let Inductor L:Real -> p:Electrical -> n:Electrical -> Equations =
  let i:Current in              C△=-1
  let v:Voltage in              C△=-1
  ElectricalBranch i v p n;     C△=1
  L *. (der i) = v              C△=1
```

The constraint delta value for specific unknowns, here the current `i`, and the voltage `v` obviously have $C_\triangle = -1$. Recall from Section 11.2.2, that for each branch, in this case an `ElectricalBranch`, rule 3 states that a potential equation should be added for that branch. Hence, a branch (both the binary and the reference branch) always has $C_\triangle = 1$. The last line shows an ordinary equation, which unsurprisingly has $C_\triangle = 1$. Hence, we can directly see that the model is balanced.

What about the other unknowns and equations that are generated during elaboration? There are two cases. Rule 1 states that an unknown potential should be added for each node in the model. Rule 2 states that a sum-to-zero equation should be added to each node in the model. Hence, these two rules always cancel out each other and their delta contribution is zero.

We believe that this approach is simpler and more intuitive to understand than using Modelica's connection semantics (compare with the constraint delta algorithm described in Chapter 6). However, we leave as future work to incorporate this into MKL's type system.

Language Design and Specification Errors

We have in this work formally specified the core of the MKL language. This could be regarded as beneficial in the sense that it becomes less ambiguous and open for interpretation. However, extending such a specification is not trivial and would affect an implementation substantially. Our approach therefore tries to move the semantic extensions of the language into libraries, so that the core can be untouched. Compared to, e.g., Modelica and VHDL-AMS, these languages can grow by adding new *modeling* abstractions without changing the semantics of the language. However, to extend how the models can be *used* typically needs language extensions. In the Modelica case, this is often accomplished by adding new annotations and then informally describing the meaning of these annotations in the language specification. In contrary, in the MKL approach, these extensions are described in libraries. Hence, we avoid the problem of language and specification errors by not introducing this kind of new construct in the language. However, the question is if the language is expressive enough, so that these semantic extensions can be described in libraries. We will discuss expressiveness aspects in the next section.

Tool Implementation Errors

If a language is implemented and used by just one software tool/compiler, there is a risk that the implementation becomes the reference implementation describing the language. However, if several tools exist that should be able to use models created by other tools, it is vital that they treat the models in the same way. In MKL we have the same challenge as for other languages that the implementation must follow the specification. We have tried to mitigate this problem by formally specifying the core of the language using operational semantics. However, the main contributing idea in MKL is that the semantics for how to *use* models is not part of the language. Instead, by implementing the simulation and translation steps in a library, we get an executable specification that can be shared directly by different tool vendors.

12.4.2 Expressiveness and Extensibility Aspects

We will now discuss how well MKL solves the problem of expressiveness and extensibility of an EOO language.

Expressiveness of Modeling Constructs

Evaluating the expressiveness of a programming language is a subjective task. If the languages that should be compared are Turing complete, they can all be used to solve the same problem. In such a case the expressiveness of the language states how "easy" it is to express a certain task. For example, if we consider the continuous-time aspect of an EOO languages where the problem can be specified as a DAE system of equations, the expressiveness power of the system would mean how "easy" it is to model a system.

In Chapter 8, we introduced the idea of higher-order acausal models, and showed that in the MKL setting we use ordinary higher-order functions to define the models. We showed how this could be used to parameterize models with other models, how a model can create new models, and how models could be recursively defined. In Modelica, several of these modeling tasks can be performed with for-equations (expressing repeating connections), model redeclare (parameterizing a model with another model) and conditional components for selecting if a model instance should exist or not. Comparing these language constructs with HOAMs regarding expressiveness is like comparing apples and oranges, i.e., it is subjective to personal preferences and tastes. However, we argue that HOAMs can model most of these special constructs in one uniform way. Hence, this enables simpler and more concisely specified language specifications.

Expressiveness of Translation Constructs

It is a well known fact that statically typed languages can reduce the expressiveness of a language. For example, in the current version of MKL, we do not support parametric polymorphism. Hence, we need to write several similar functions, e.g., a new fold function for each type it can operate on. This is a limitation in the current MKL language. An extension to support parametric polymorphism is planned as future work.

In contrast, in a language where the type checking is performed at runtime, we gain expressiveness/flexibility, but loose the property of compile time checking and early error feedback.

Our design where a model can be of either a specific type, e.g. <Int>, or a model of any type <> is an attempt to find a good trade-off. As we discussed earlier, the specific model type is used to check the correctness of models during modeling. However, by relaxing type equality checks to a consistency check where for example <> is consistent with <Int>, we get an expressive semantics for traversing and inspecting models. However, the expressiveness comes at a price. For example,

```
let v1 = ((val(+)) (val 3)) (val 10)
```

defines an a model where the prefix variant of the plus operator is applied to two integer models. The match expression

```
let v2 = match v1 with
         | x y -> x false
```

matches the model and extracts `x`, which is `(val(+))@(val 3)`, where the model application has been written out explicitly with infix operator `@`. However, `x` has now type `<>` because the static type checker cannot know what specific model type the left term of the model has. Hence, `x false` creates a valid expression `(val(+))@(val 3)@ (val false)`. This transformation can be regarded as nonsense, but is still legal from a type checking point of view. Now, let us define a function foo for deconstructing `v1` and `v2`:

```
let foo v:<> -> String =
  match v with
  | x + (val y:Int) -> int2string y
  | _ -> "Not an Int"
```

Parameter `v` is of any model type. If we apply `foo` to `v1`, we get as result `"10"`, using the first mathch case because we match that the value type is an `Int`. However, if we apply `foo` to `v2`, we get as result `Not an Int`. Hence, even if we can create bogus model terms during transformation, we cannot extract using pattern matching terms that violate the types, i.e., `int2string` cannot in this case accidentally get a boolean value as an argument. We should again note that this is our believed type safety of the system, it is not yet proven.

From our preliminary tests of implementing both elaboration semantics, code generation and export of flat Modelica, as well as simulation code, we have found that the static checking has helped us to find many bugs in the programs. The risk of creating bogus transformations, as the one above, has surprisingly not been a problem so far. However, our tests are as yet far too small and limited to draw any general conclusions in this matter.

Extensibility

Extensibility of a language concerns how easy it is to add new language constructs to the language. If the language has a large, complex, and informal language specification, it might be hard to predict the consequences of adding a particular language feature. Moreover, if there exist several different compiler/simulation tools implemented for the language, new language constructs can give major consequences for each implementation.

In Chapter 4 we argued that the most preferable way to grow a language (to extend it) is growth by *new user defined abstractions*, i.e., that the language is not changed at all. In for example Modelica or VHDL-AMS, this is accomplished for the modeling part by enabling a library developer to create new libraries in different physical domains. However, in the Modelica case, it has turned out that several new language constructs need to be added at each language revision (every 1-2 year). These changes are needed to enable better support for new model libraries. Moreover, the actual use commands, e.g., simulation, checking, etc., of the models cannot be specified by the user, these are defined in the implementation of the used software tool.

In contrast to Modelica, our approach emphasizes the use of new user defined abstractions even for how the model is used. For example, if the language has support for specifying initial values to variables, and later it is decided by the language design committee that a library also needs *initial equations*, the definition of MKL does not need to be changed. For example, by adding the following two lines

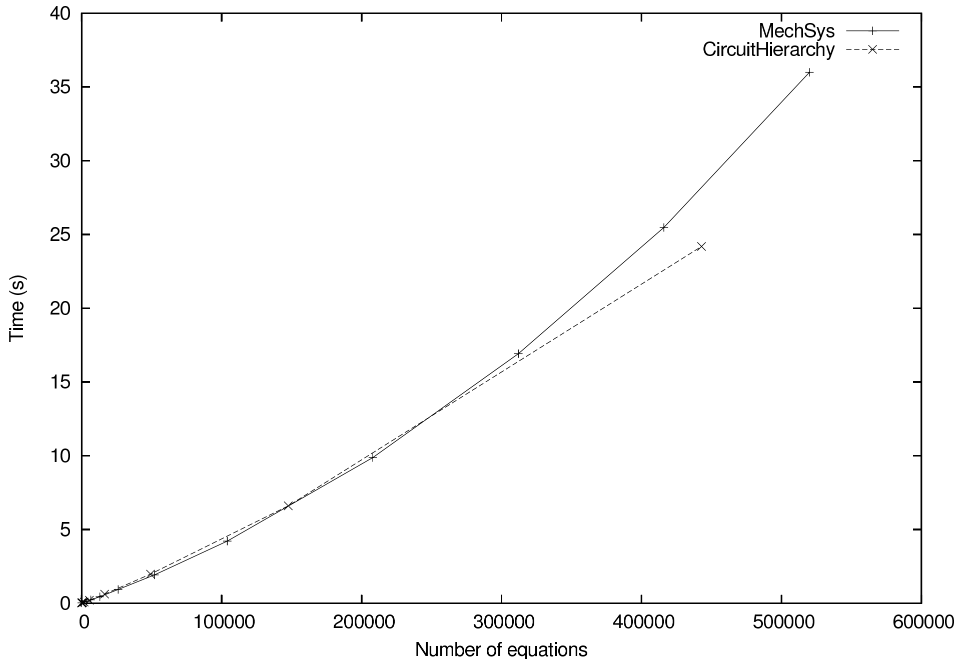


Figure 12.2: Elaboration time in relation to the number of generated equations.

```
let InitEquation : <Real -> Real -> Eqs>
let (=.) : <Real -> Real -> Eqs> = InitEquation
```

we have defined a way to specify initial equations using the infix operator `=.` Of course, the actual semantics for how to handle these equations must be specified by elaboration functions, but this can be done in a library instead of in a software tool such as a model compiler. We see the following main benefits with this approach

- Tool vendors do not need to update their product for each library extension
- Library developers becomes less dependent on tool vendors, i.e, they can extend the language themselves
- The likelihood that a model will behave the same way in different tools increase.

We believe that the examples and implementation of this thesis work give the evidence that this is indeed possible.

12.4.3 Performance Aspects

Performance of simulations, i.e., the time it takes to simulate a model, is important. In this thesis we have implemented an interpreter for evaluation and verification of our solution. However, we argue that performing a performance comparison between our current interpreted implementation with available Modelica implementations would be subject to

bias. Firstly, we do not know if the Modelica implementations are flattening, sorting equations, and also performing other kinds of symbolic manipulation. On the other hand, our solution is interpreted, while for example both Dymola and OpenModelica are running compiled code for the elaboration phase. Also, keep in mind that the comparison is for different EOO languages, MKL in our case and Modelica for Dymola and OpenModelica.

However, a performance aspect that is interesting regardless if the solution is interpreted or compiled, is how well it scales. In this case, we are interested in how the elapsed simulation time increases with respect to the size of the model. The size of a model can be measured in different ways, e.g., number of submodels, number of branches etc. We have chosen to use the number of equations generated after elaboration as the measurement of size. The rationale for this choice is that we can compare models with different structure and domains.

We have performed experiments with two different models. The first model, called `MechSys` is the model with a flexible shaft that was presented in previous chapters. The parameter that has been changed to get different measurements is the number of elements in the shaft. For example, in the largest test case 40000 elements of the shaft generated 520026 equations. The test model for `MechSys` is listed in Appendix D.11.

In the second experiment called `CircuitHierarchy`, a model consisting of three electrical components was created. This model was then instantiated to three new components, which were connected. This procedure were performed recursively, generating 442873 equations for 11 levels. The test model for `CircuitHierarchy` is listed in Appendix D.11.2.

Consider now Figure 12.2 that shows the elapsed simulation time in relation to the number of generated equations for the two models described above.

The `CircuitHierarchy` has a few measurement points with larger equations sizes. The reason is that 10 hierarchy levels generate 147628 equations and 11 levels generate 442873 equations. For larger equation sizes the underlying OCaml runtime system generates an error.

Two obvious observations can be made from the graph. Both cases grow approximately with the same factor and the growth is not entirely linear. However, when considering that half a million equations is many equations if it would have been a model used in practice, we would argue that the solution in these cases can be considered scalable. However, we shall also notice that the elaboration phase is a minor source of the total simulation time, compared to symbolic manipulation and numerical simulation. Hence, it is premature to draw too many conclusions from these tests.

12.5 Chapter Summary and Conclusions

We have in this chapter given an overview of the prototype implementation of MKL as well as shown examples of two ways of using models: for continuous-time simulation and exporting Modelica code.

We explain how the prototype has been verified by simulating several example models using both MKL and a Modelica environment. Finally, we discuss and briefly evaluate our solution from a safety perspective, expressiveness and extensibility perspective, and performance perspective.

We can conclude that our approach gives similar possibilities of continuous-time modeling as a state-of-the-art modeling language Modelica. However, MKL does not yet support hybrid modeling and it is therefore hard to compared larger models from e.g., Modelica Standard Library (MSL). The main difference of MKL and languages such as Modelica is the ability in our approach to not only create reusable modeling libraries, but also to create libraries for inspecting, translating, and using models.

Part III

Related Work and Concluding Remarks

13

Related Work

IN this chapter we discuss the most closely related work. The chapter is structured as follows:

- We first give a short overview of the most prominent equation-based modeling languages available. We then discuss how MKL is related to these languages (Chapter 13.1).
- We discuss related work related to Modelica semantics (Chapter 13.2), as well as MKL semantics (Section 13.3).

13.1 Equation-Based Modeling Languages

In this section, we discuss and describe different equation-based modeling languages primarily used for physical modeling. For some of the languages we also directly compare to MKL. At the end of the section, we summarize the main differences and similarities between our approach and the discussed languages.

13.1.1 Modelica and Predecessors

Modelica is an equation-based object-oriented (EOO) language designed for modeling large and complex physical systems [104]. For a brief overview of the language, see Chapter 2.

The design of the language is an international effort to create a unified object-oriented language for system modeling and simulation [55]. Language designers of a number different equation-based modeling languages have actively participated in the design of Modelica [51]. We now briefly present some of the predecessor languages.

The language Dymola (Dynamic Modeling Language) was introduced by Elmqvist in 1978 [47]. It introduced concepts for facilitating creation of large models by hierarchical composition of sub-models. Dymola stands today for *Dynamic Modeling Laboratory* and is a commercial product based on the Modelica environment [45].

The Omola language by Andersson [6] is an object-oriented acausal modeling language that makes use of several concepts from OOP, including the class concept and inheritance. The language handles both discrete and continuous-time systems.

Another early language for object-oriented acausal modeling is the Natural Model Format (NMF). This language was first developed and introduced for building simulations [128].

Yet another language is SIDOPS+ [16] that supports block-diagram and bond graph models. It was intended as a open language that focused on modeling of mechatronic systems.

Smile is a simulation environment developed for energy systems that is based on object-orientation and equation-based modeling principles [82]. A variant of the Smile system called Smile/M is extended with the capability to also compile Modelica models [50]. The Smile system separates the modeling language from the experimental description language. The modeling language is an extension of Objective C [50].

ObjectMath is an extension of Mathematica for equation-based, object-oriented modeling [58]. The language combines object-oriented, constructs such as classes and inheritance, with the computer algebra facilities of Mathematica.

13.1.2 Extensions to Modelica

There are also a number of languages defined as extensions to the Modelica language.

MOSILAB is a modeling and simulation tool for modeling of structurally dynamic systems [112]. It is implemented as an extension to the Modelica language adding statecharts for modeling discrete model switches.

Optimica is an extension to Modelica for compact formulation of both static and dynamic optimization problems [5, 79]. A key aspect of the extension is that the formulation of the optimization problem is independent of the numerical algorithm that is used for solving the optimization problem.

Both of these extensions show the need for extensibility of an EOO language. Also, Optimica is an example where Modelica models are used for other purposes than simulation.

13.1.3 VHDL-AMS

VHDL-AMS is a hardware description language (HDL) used for describing digital, analog, and mixed-signal systems [10]. It is an extension to VHDL also supporting analog signals; i.e., continuous-time models. It is an ISO standard [72].

A VHDL-AMS model is defined by an *entity* and one or more *architectures*. The entity is the interface of the model and the architecture its implementation. In Modelica, the class type specifies the interface and the model itself is the implementation. Similarly, in MKL, the type of the function abstracting the equation system is the interface, while the body of the function containing equations is the implementation.

In VHDL-AMS, *quantities* represent the unknown variables in a DAE [40]. The language supports three kinds of quantities: free, branch, and source quantities. A free quantity is a signal that can be used for causal modeling. A branch quantity is used for modeling conservative energy systems, i.e., acausal modeling. A source quantity is for modeling of frequency and noise. The free quantity is similar to MKL's standard library's definition of `Signal`. In VHDL-AMS, *terminals* are created by a specific nature, e.g., the `electrical` nature. These terminals are then connected to *ports* of entities. Terminals and MKL nodes are similar and connecting to a port is basically the same as passing a node to a model function in MKL.

Branch quantities can be declared inside architecture bodies. Branch quantities are used for defining *across* and *through* quantities (same as potential and flow variables in Modelica). This has a direct correspondence to branches as defined in MKL. In fact, the naming and idea of branches is inspired by VHDL-AMS.

13.1.4 Verilog-AMS

Verilog-AMS is a hardware description language (HDL) that is an extension to Verilog with analog and mixed-signals [3]. In Verilog-AMS, signals are associated with nodes, which are connected to ports of components. Also, models of conservative energy systems have potential and flow values associated with each node. In Verilog-AMS, branches are the paths between two nodes through a component. Each branch has the potential difference and the flow. Hence, the basic structure for defining branches, nodes, flow etc. is similar between MKL's standard libraries and Verilog-AMS. However, an essential difference is that the MKL language itself does not include these constructs, while they are part of the Verilog-AMS language.

13.1.5 gPROMS

gPROMS is an equation-based language used for combined lumped and distributed parameter processes. It was introduced in 1992 by Barton [13] in his PhD thesis and has evolved to become a commercial product. Its main application domain is chemical process modeling. The language is based on three kinds of entities: models, tasks, and processes. The model describes the continuous-time behavior of the modeled system, including discrete changes. Tasks describe the disturbance and external actions on the system. The process describes the complete simulation experiment. The language has later been extended by Oh and Pantelides to support mixed integral, partial differential, and algebraic equations (IPDAEs) [115].

13.1.6 Hybrid Chi

Hybrid χ (Chi) [59, 146] is a hybrid specification formalism that can describe discrete-event, continuous-time, and hybrid systems. The language is a concurrent language based on Communicating Sequential Processes (CSP) (the discrete-event part), and differential algebraic equations (DAEs) (the continuous-time part). The language does not yet support acausal connections. Its syntax and semantics has been formalized using structured operational semantics [146, 145].

13.1.7 Functional Hybrid Modeling and Hydra

Functional Hybrid Modeling (FHM) is a paradigm that combines functional programming and non-causal (acausal) hybrid modeling [109, 110]. The concept of Functional Hybrid Modeling (FHM) is a generalization of Functional Reactive Programming (FRP) [150], which is based on reactive programming with causal hybrid modeling capabilities. Both FHM and FRP are based on *signals* that conceptually are functions over time. Unlike FRP, which only supports causal modeling, the aim of FHM is to support acausal modeling with structurally dynamic systems.

The most developed language within the FHM paradigm is *Hydra*, which is created by Giorgidze and Nilsson during recent years [61, 62, 63]. However, the basic ideas for the language were sketched already in 2003 by Nilsson *et. al.* [109].

Similar to MKL Hydra is supporting continuous-time modeling using DAEs. Both languages are supporting a kind of model abstraction that is first-class, called *signal relations* in Hydra. Signals in FHM are time-varying values, i.e., a function from time to a value.

In FHM/Hydra, there are two distinct levels: the functional level and the signal level. The former deals with time-invariant expressions, e.g., signal-relations and the latter with time-varying quantities, e.g., signals. Signal relations and *signal relation applications* are used for composing equation systems. That is, in FHM, there is a separation between function application at the functional level and signal relation application at the signal level. In contrast to FHM, MKL uses function abstraction and function application both for unknowns (representing signals during simulation) and for constant values.

Hydra is implemented as an embedded DSL using Haskell with the Glasgow Haskell Compiler (GHC) extensions as the host language [63]. It is designed to support structural dynamic systems where the numerical simulation result is computed using the SUNDIALS solver suite [68] and the residual function is just-in-time (JIT) compiled using LLVM [86]. It is using a *mixed-level* embedded approach, combining both shallow embedding (domain-specific syntax is expressed directly in the host language) and deep embedding (making use of interpretation and compilation). We shall also note that embedding is an implementation technique for Hydra that is not required by the FHM paradigm.

Currently, there does not exist any published formal semantics for Hydra or FHM, besides the formalization available for the host language (Haskell). However, the formalization of the semantics has been one of the main objectives for MKL.

13.1.8 Sol

Sol is a equation-based modeling language designed for modeling and simulation [154]. Its design is based on Modelica, but the objective of Sol is be a minimal research language for simulating variable-structure systems. In his PhD Thesis, Zimmer [155] develops a methodology for dynamic processing of differential-algebraic equation for arbitrary structural changes of the equation systems. The Sol language is used to test and evaluate the methodology within several different domains.

13.1.9 Acumen

A recently developed modeling language called Acumen [153] is aimed for bridging the gap between analytical models and simulation code. It supports continuous-time modeling of PDEs, discrete-time modeling using event-driven functional reactive programming (E-FRP), and hybrid models. It has so far been evaluated in the mechanical domain.

13.1.10 Comparison to MKL

Similarly to all of the above mentioned languages, MKL can be used for continuous-time modeling and simulation using DAEs. However, we have not yet evaluated the approach for hybrid systems. Compared to Sol and FHM, which both have been designed for structurally dynamic systems, MKL has not yet been extended with such language feature. However, as stated in the future work (Section 14.2), we believe that MKL can be a good platform for experimenting with structurally dynamic systems.

According to Definition 8.2.1 on page 115 FHM/Hydra and MKL are the only ones of the above mentioned languages that have full support for higher-order acausal models. Modelica has a construct called `redeclare` that can parameterize models with other models, but models cannot be passed to or created by functions.

In contrast to all the above languages, the semantics of *how to use* the models can also be specified in MKL libraries. This enables extensible formal executable specifications where important language features are expressed through libraries rather than by adding completely new language constructs. For example, probes (Section 11.3), export of equation systems (Section 12.2.1), and simulation (Section 12.2.2), are all constructs implemented as MKL libraries.

We have formally defined both the static and dynamic semantics for a core part of the MKL language as well as proven type safety for the core language. This formalization includes both phase 1 (type checking) and phase 2 (collapsing the instance hierarchy) of the elaboration process (recall Section 11.1 on page 159). Moreover, because phase 3 (the connection semantics) is specified as an executable library within MKL, we have formalized most of the elaboration semantics[1].

13.2 Modelica Semantics

In this section we discuss some related work regarding Modelica semantics. First, we state previous work on specifying the elaboration semantics. Second, we discuss related work on types and isolating faults in models.

13.2.1 Natural Semantics

Already in 1998 Kågedal and Fritzson [84, 85] defined a formal specification for a subset of the Modelica language. It was a *Natural Semantics* [81] specification expressed in the executable specification language Relational Meta-Language (RML) [120]. This work influenced the design of the language and the official Modelica specification. The

[1] Since we in this thesis have not formalized translation steps e.g., pattern compilation and other constructs, e.g., lists and tuples, we do not want to claim that the whole process has been formalized.

executable specification has gradually evolved and is now the code base for the OpenModelica project [117]. In 2006, the code base was automatically translated from RML to Meta-Modelica [56] with the purpose of making it more accessible to software engineers in the Modelica community. Hence, today the project is more intended to be a complete implementation of the language than a specification itself.

13.2.2 Instance Creation

Mauss has made several contributions towards formally describing the elaboration process (called *instance creation* in his work) of a subset of Modelica, i.e., the translation process from a Modelica model into a system of equations [95]. His published work describes an algorithmic specification approach that focuses on Modelica's complex look-up rules and modification semantics, including redeclaration of classes and components. Semantics for describing restrictions on validity of a model, such as types, restricted classes, and most prefixes are not considered.

13.2.3 Modelica Types

To the best of our knowledge our work on the type concept in Modelica is the first study to introduce and differentiate between class types and object types. An overview of the type concept was given by Fritzson [51, Section 3.14] in 2004, but it did not include the insights about class types and object types that are discussed in this thesis.

13.2.4 Balanced Models

In September 2007, a new version 3.0 of the Modelica specification was released [103]. The largest change in the language was the new constraint that all models in the Modelica language must be *locally balanced*, meaning that the number of unknowns and equations must be equal at every hierarchical level of a model [116]. The exception is for partial models, which are not checked. Enforcing local balancing of models is basically the same as stating that constraint delta and effect delta (see Chapter 6) should be zero in all connectors and models. However, even if the approaches are similar, there are some distinct differences.

The balanced model concept in the Modelica specification has taken a "top-down" approach and defines its solution for the whole Modelica language. The constraint delta approach is given for a small subset of the Modelica language, with the purpose of stating a detailed algorithm.

The Modelica specification requires that models are *always* balanced, with the exception of partial classes. The constraint delta concept as explained in Chapter 6 is more flexible, and accepts that sub-components are over or under-determined, as long as the global model has a constraint delta of zero. Both of these approaches are justified by examples and tests.

It should also be noted that the idea of using these approaches were developed in parallel within Dynasim and by the author during year 2006. At the time of the publication of the constraint delta approach [28], the paper handed out and discussed at the Modelica Association design meeting. During the late 2006 and early 2007, further interaction and

discussions have occurred between the author of this thesis, Dynasim, and members of the Modelica Association.

Finally, it should also be noted that there is a paper by Nilsson *et. al.* [109] from 2003, where the idea to incorporate information about the balance between equations and unknowns into the type system is stated. However, no information or strategy on how this should be conducted is presented.

13.2.5 Structural Checking of Models

There has been a number of attempts to perform more sophisticated analysis for detecting and isolating constraint errors. Instead of just counting equations and unknowns, these methods try to determinate if the system of equations is structurally singular, and in such a case isolate the fault.

An early attempt of semiautomatic debugging technique was suggested by Bunus & Fritzson [29]. They implemented a system called AMOEBA (Automatic Modelica Equation-Based Analyzer), which performs a graph algorithm on the flat system of equations. The source of the error is then traced back to the original component. Hence, this approach can detect faults that goes undetermined by the constraint delta approach (the system is singular but the number of equations and unknowns match). However, in the case of an illegal number of equations and variables, the simple counting approach is sufficient.

Nilsson [108] outlines an approach to perform check of structurally singular systems at the type level. The idea is similar to our work on the structural constraint delta, but instead of just annotating each type with a single integer value, Nilsson suggests to expose information about which variable that appears in which equation. However, to not make the type as large as the whole system of equations, Nilsson suggest a way to approximate the structural information for the types.

13.3 MKL Semantics

In this section we first discuss types and dynamic semantics related to the formalization of semantics of MKL. This is followed by discussion about related techniques for metaprogramming.

13.3.1 Formal Semantics

The formalization of the dynamic semantics of the core of MKL (Chapter 10) is given using small-step semantics. An alternative would be to use big-step semantics, also called natural semantics [81]. We choose the small-step style because of the possibility to use the progress and preservation lemmas for proving type safety [124].

The embedding of values in model value expressions `val` e:τ and the use of the any model type <> is inspired by the work of Abadi *et. al.* [1] about dynamic typing in a statically typed language. In their work they provide a language construct called dynamic that is packaging a value together with its type. Such an term has the type Dynamic (note the capital letter) that can later be deconstructed using a typecase construct. A type

system using *gradual typing* allows both static and dynamic type checking in a single programming language [132, 133]. A *type consistency* relation is used instead of type equality. We have adopted a similar approach using type consistency, where the any model types <> is consistent with specific model types $<\tau>$ for some type τ. The main difference compared to MKL's type system is that in gradual typing any type is consistent with the dynamic type (denoted ?), but in MKL only model types can be consistent with <>.

The concept of *generalized algebraic data types (GADTs)* is a powerful generalization of algebraic data types that is part of languages such as Haskell, Standard ML, and OCaml [123]. GADTs have in the literature appeared under different names, such as guarded recursive data types [152], equality-quantified types [131], and first-class phantom types [39]. GADTs can for example be used when embedding DSLs in a host language (e.g., Haskell) and then check if a program in the DSL is well-typed. This has some similarity to MKL's approach to type checking with specific model types. Potentially an approach where GADTs are used in MKL's type system could preserve type information of specific model types during transformation of models. However, it is not clear whether incorporating GADTs would preserve the expressiveness of the current MKL approach, or how the concepts of unknowns and model lifting would be incorporated. We regard this as interesting future research.

Lava is tool for specifying, designing, and implementing hardware [15]. Its language is an HDL embedded into Haskell. Similarly to MKL, Lava is using higher-order functions to compose circuits together. However, Lava is designed for digital circuits and is not concerned with the connection semantics appearing in acausal languages. Claessen [41] describes two approaches for solving the problem in a lazy language that circuits are graphs but viewing them using algebraic data types makes it not possible to distinguish them from infinite trees. He both suggests an approach using monads [149] and an approach called *observable sharing*. The latter makes the graph sharing observable by extending the language with non updateable reference cells and an equality test for sharing detection. Because MKL is a call-by-value language we have taken a less sophisticated approach using the $\nu(\tau)$ expression to generate new unknowns that can be used to represent nodes in a circuit. These nodes can then be directly compared using an equality operator.

Instead of using the λ-calculus as the basic calculus for MKL, the *pattern calculus* by Jay [77] could be an alternative. It is an expressive calculus where patterns are part of the core calculus. However, we have in this thesis taken a more traditional approach and formulated our language as an extension to the lambda calculus. Hence, the pattern matching operations are not part of the core language but instead defined as a translation to the the core language.

13.3.2 Metaprogramming in EOO Context

Most Modelica tools support some kind of scripting facility that can be used to programmatically start simulations and collect results. In the Dymola and OpenModelica cases the script language is a subset of Modelica. Some tools also opens up APIs for inspecting and traversing the abstract syntax tree of a model.

A limited form of metaprogramming facility for OpenModelica was presented in [9],

and followed up by the work of Fritzson *et.al.* in 2005 [56], where the Modelica language was extended with pattern-matching operations and transformations on ASTs. The current language, called MetaModelica, is a heterogeneous metalanguage, meaning that the metaprogramming language and the object language are not the same. In [127] Pop shows how MetaModelica is used as the implementation language of OpenModelica.

In contrast to MetaModelica MKL is a homogeneous metaprogramming language in the sense that it can create models and inspect the models in the same language.

13.3.3 Metaprogramming in General Purpose Languages

During the last two decades, there has been extensive research in the area of statically typed languages aimed at extensional metaprogramming, for example MetaML [140, 143, 144] and its dialects (e.g., MetaOCaml [142]). In these so called multi-stage languages code can be created, combined, and executed.

MKL is not performing intensional analysis on the program code, but on the values of a model type. Because models are treated as data and cannot be executed, the typing gets much easier. Hence, we call our approach intensional analysis *on models*.

Static metaprograms are metaprograms that execute before the load time of the program or code that it manipulates [43]. A popular way of compile time metaprogramming is to make use of the template system of C++. Hence, programs can be generated before execution. Template Haskell [130] is another example of compile time metaprogramming. MKL does not support any compile time metaprogramming. During compile time, expressions are lifted to be models, so that they can later be inspected.

14

Concluding Remarks

In the following final chapter, we state the conclusions (Section 14.1), and outline future work (Section 14.2).

14.1 Conclusions

This thesis concerns the problem of designing and defining semantics of equation-based object-oriented (EOO) modeling languages. We have studied this problem area in two contexts, corresponding to Part I and Part II of the thesis. In the first part, we study the state-of-the-art EOO language *Modelica* both by discussing the current language's properties, and proposing new language solutions. In the second part, we present a new research language called the *modeling kernel language (MKL)*. In MKL, reusable acausal models, based on differential algebraic equations, can in a way analogous to what is done Modelica be defined in model libraries. In contrast to Modelica, the semantics of *how to use* the models, i.e., operations on models, are also defined in MKL libraries, and not in language specifications or separate software tools.

The problem area of designing and defining the semantics of an EOO-language concerns trade-offs between several aspects of the language. We have focused on the trade-offs between safety, expressiveness, and extensibility. We have studied these aspects within three research areas:

- Understanding the semantics of the Modelica language
- Early detection of modeling errors
- Expressive and extensible formal semantics

14.1.1 Understanding the Semantics of the Modelica Language

Studying the semantics of Modelica is interesting and important per se. The language is the state-of-the-art EOO language today and widely used both in academia and in industry. However, the language is large and very complex with an informal language specification. Hence, the semantics of the Modelica specification is hard to interpret unambiguously. We have discussed and analyzed two research questions concerning this area.

The first question concerns if there is a way to restructure an existing large informal specification so that it becomes less ambiguous and still understandable for a general audience. We discuss different alternatives and propose a middle-way approach for Modelica, where the grammar for ASTs shall be formally defined and the translation between AST variants is informally described. Moreover, we discuss different ways of growing an EOO language in general and Modelica in particular. The conclusion is that the objective should be to design the language so that it can grow by allowing the user to define new abstractions in the language, and in such a way make the language more extensible.

The second research question concerns the meaning/semantics of types in Modelica and how it relates to the language's concept of classes. We have analyzed the type concept, which was only indirectly described in the Modelica language specification at that time. The conclusion and also the main contribution of this work was the insight that Modelica has two categories of types: *class types* and *object types*. The reason is that Modelica has a structural type system making types and classes separate concepts. We explain the findings by giving several examples, as well as defining a concrete syntax for specifying the types. It was also concluded that Modelica's prefixes are part of the types and a categorization of an interpretation of the specification was proposed.

14.1.2 Early Detection of Modeling Errors

The third research question concerns the problem of determining if a model is over- or under-constrained before elaborating a model. It is trivial to detect how many equations and unknowns you have after elaboration. However, if the numbers do not match, it is non-trivial to isolate the fault and give good user feedback for where the error is located in the model.

We propose a novel concept that we call *structural constraint delta*. The basic idea is simple; each type is annotated with an integer value stating the difference between the number of equations and the number of unknowns of a model's type. An algorithm is detailed for computing the constraint delta value, which turns out to be fairly complex mainly due to the connection semantics of the Modelica language. We successfully tested the solution on a small subset of Modelica, but from that evaluation it is hard to give any guarantees of correctness for a full Modelica implementation.

The work on structural constraint delta was first published in 2006 [28] and later somewhat influenced the concept of balanced models for Modelica 3.0, released in 2007. In retrospect, with the acceptance of introducing this non-backwards compatible change to the Modelica language, we conclude that the idea of determining over- and under-constrained systems by considering the balancing of models is applicable in practice.

We would also like to emphasize that the idea of structural constraint delta is not limited to Modelica, and ought to be useful for EOO languages in general. In Section 12.4.1,

we briefly discussed how it can be used in MKL. The introduction of constraint delta into MKL's type system is considered future work, but preliminary investigation shows that the method for computing the value seems simpler in the case of MKL's connection semantics than for Modelica.

14.1.3 Expressive and Extensible Formal Semantics

From the earlier study of Modelica, we have concluded that it is hard to formalize such a large and complex language. Our study of Modelica can be viewed as a top-down approach, trying to analyze something that is large and exists. Our next research question concerns the opposite - a bottom-up approach to the design problem.

The fourth question for this thesis concerns the possibility to formally define an EOO language, base it on the proved theory of the simply typed lambda calculus, and achieve the same expressive power as state-of-the-art EOO languages. The result of our work in this area is the first version of our research language MKL. MKL is fundamentally a statically typed functional language with metaprogramming capabilities, that is based on the lambda calculus. We have formally defined the core of the language using small-step operational semantics together with inference rules describing the static type system. This language is based on the simply typed lambda calculus, extended with metaprogramming constructs for handling unknowns and model types. Moreover, we also proved type safety for the core language.

We introduce the concept of *higher-order acausal models (HOAMs)*, where models can be passed around as first class citizens in the language. By giving examples from the electrical and mechanical domain, we show the expressive modeling power, and also argue that several language constructs in Modelica can be subsumed or expressed by using HOAMs. We conclude that HOAMs, demonstrated by using MKL, give a high expressiveness for modeling continuous-time acausal systems. However, for hybrid systems, further research is needed to draw any general conclusions.

We have earlier concluded that a preferable approach of growing an EOO language is that the user can add new language constructs, without the need of language changes. The fifth and last research question concerns the possibility to design a language where not only modeling constructs can be extended by the user, but also semantics for using models (i.e., meta-operations on models).

The solution that we have investigated using MKL is to put the semantics for defining EOO language constructs such as ports, equations, initial value definitions, and probes into MKL libraries. Then, instead of informally specifying e.g., elaboration and simulation semantics in a separate language specification, we have put the formal executable specification of the semantics into MKL libraries. These libraries can then be used to inspect, analyze, and transform models using metaprogramming techniques. Our initial experimental attempts include full elaboration of generating flat form of DAEs from MKL source models, simulating the DAE using numerical DAE solvers, and to generate flat Modelica code from MKL models. We have also compared and discussed our connection semantics with Modelica's approach.

An MKL interpreter has been implemented to verify and evaluate our solution. We have verified by testing at a system level that models in MKL and Modelica behave similarly. Performance tests have also been conducted, where the main conclusion is that the

elaboration phase is scalable.

Finally, we regard our work with the research language MKL as a first attempt of using metaprogramming techniques with *intensional analysis of models* within the EOO language area, i.e., the ability to defining both reusable model libraries as well as reusable libraries for manipulating models within the same language. The approach has been evaluated on a non-trivial set of continuous-time models and our hope is that this work should inspire for future research in the field where more complex models (e.g., multi-body and fluid systems) as well as other uses (e.g., optimization, grey-box system identification, and model reductions) can be realized using metaprogramming techniques within the same language.

14.2 Future Work

Compared to other language paradigms, EOO languages can be seen as a fairly young field. However, new opportunities and problems arise when we introduce concepts from the computer science field in general and the programming language theory field in particular. Examples of such concepts are higher-order functions and metaprogramming techniques. Our work in this thesis is an attempt to go in such a direction. Nevertheless, there are still many outstanding issues and interesting problems left for future research. In the following chapter we highlight some of them.

14.2.1 Extensional Metaprogramming

We have in this thesis described how we use metaprogramming for intensional analysis of models by inspecting and traversing their content. In Chapter 12, we have given an example program for how models can be simulated by using an external DAE solver. Such a solver needs a residual function that computes values based on the DAE. In the example we compute this vector value by interpretation, i.e., we traverse the system of equations and compute the resulting value. However, an idea for future work is to utilize *extensional metaprogramming* [129] for code generation of a function to compute the residual before the simulation. The hypothesis would be that if the generated code is just-in-time compiled, we could get simulation performance comparable to simulation systems that statically compile the residual. We believe that multi-stage programming [141, 143] could be the right choice because this also conforms to a statically typed environment. We have so far performed preliminary tests of multi-stage programming in an MKL extension, where we can simulate the system using extensional metaprogramming. However, our current test prototype is interpreted and therefore only limited speedup is gained. It has been shown that e.g. in MetaOCaml [31], good speedups have been achieved. However, several research challenges remain, where one of them is efficient integration with the runtime environment for the generated residual function and the numerical solver system (e.g., Sundials [68]).

14.2.2 Hybrid and Structural Dynamic Systems

In this thesis we have focused on continuous-time systems. We do not currently see any major challenges of defining simple hybrid constructs directly within MKL, e.g., to define

a `pre` and `reinit` construct similar to Modelica:

```
let reinit : <Bool -> Real -> Real -> Eqs>
let pre: <Real -> Real>
```

A reinit equation could then be defined as

```
reinit (x >=. 0.) v ((pre v) *. -.1.)
```

where the first argument is the condition, the second argument the unknown that should be reinitialized, and the third argument the expression if the event occurs.

However, how a more general solution can be created is more interesting. For example, that **if**-expressions are applicable for models as well, where the guard can be an unknown. In the general case, the aim would be that any expression could be switched if the guard changes over time. Because models are first class in MKL, this would imply that we obtain a structurally dynamic system, where instances of models can switch over time. Structurally dynamic systems with a fixed number of states have previously been developed in MOSILAB [112]. Other research results within this area are Functional Hybrid Modeling (FHM) [62, 109] and the research language Sol by Zimmer [155]. We believe that MKL could potentially be a good framework to further explore structural dynamic systems.

14.2.3 Code Generation and Time Aspects

To achieve high confidence of cyber-physical systems, i.e., systems that mix physical dynamics with software and networks, it is vital that such systems can be modeled and simulated at design time. Hence, it is necessary that both the physical model of the plant and the model of the controller reflect the reality of the real physical system together with generated code running on embedded computer systems. One major problem when designing embedded systems today is the non- deterministic behavior of computation, i.e., it is very hard to estimate the timing of executing embedded code [87]. Traditionally, simulated controllers are idealized to take zero time for computation, resulting in different dynamic behavior between the simulated system and the real system. Synchronous languages [14] used for code generation of discrete-time systems are predictable and repeatable, but currently lack a sound integration with acausal modeling languages with continuous-time behavior. To mitigate these issues, the research challenge would be to establish a way of automatically extract timing information for the target platform for code generation and then include these timings into the simulated system. The simulated system should closely match the real system; giving higher confidence of the modeled system at an early stage of the system design process. We believe that MKL could be a good platform to experiment with such a design.

14.2.4 Structural Constraint Delta

We do not expect that it is too difficult to add the structural constraint delta concept to the type system of MKL's. However, it is more challenging to make it flexible and not dependent on built-in functionality, e.g., that the type system must explicitly know the difference between an `ElectricalBranch`, the independent variable `time` and an

ordinary equation. Hence, there must be a way to describe in the language what kind of construct that contributes to constraint delta.

14.2.5 Polymorphism, Type Classes, and Algebraic Data Types

MKL is a small research language that lacks basic functionality that you expect from a functional language. The most important constructs that would be useful are parametric polymorphism, type classes, and algebraic data types. Adding these language features might be less of a research challenge and more of a development work. However, further investigation must be conducted to see how these language constructs interact with the model type presented in this thesis.

14.2.6 Efficient Compilation

The current implementation prototype of MKL is interpreted. However, to make this solution useful in practice, we believe that a compiled system should be developed. One alternative would be to implement the compiler and then to use e.g., LLVM [86] as the backend. Another alternative would be to use an existing functional programming language as the backend, e.g., to generate OCaml code that can later be compiled and executed.

14.2.7 More Complex Modeling

Further case studies and experiments on more advanced modeling tasks should be investigated, to see how far MKL can be used. For example, case studies of implementing fluid systems or multi-body systems could be the next step. In the fluid case, Modelica's new stream connector is a interesting construct to study [104]. Moreover, other constructs that could be investigated is the possibility to add partial differential equations.

14.2.8 Uses Beyond Simulation

We have in this thesis showed how to use models in two ways; for simulation and for export to flat Modelica code. However, there are many other potential uses of models. For example, the work by Åkesson *et. al.* [5] on optimization of Modelica models. Casella *et. al.* [34] propose several uses of models besides simulation, where *model reduction* is one application area. Another application area could be *grey box* system identification [88], where the equations for the model are known and parameters should be estimated by using measured data from a real system. We see as interesting future work to explore the possibility to use the MKL approach of implementing the semantics of these uses into MKL libraries.

Syntax of MKL

In this appendix we define the concrete syntax for MKL 1.0[1] as well as an abstract syntax for the intermediate language of the prototype implementation.

A.1 Concrete Syntax

A.1.1 Notational Conventions

Terminal symbols are reserved words written with bold typewriter font (e.g., **`let`**, **`if`**), text enclosed between double quotes (e.g. `"="."("`), or tokens written in upper case (e.g., `UINT`, `IDENT`). Nonterminal symbols are written using lower case letters and `typewriter` font. Enclosing text in curly brackets ("`{`" and "`}`") means repetition zero or more times. Enclosing text in square brackets ("`[`" and "`]`") indicates that the item is optional.

A.1.2 Comments

Comments are sequences of characters between `/*` and `*/`. Comments may be properly nested. One line comments start with `//` where the following sequence of characters on the same line is the comment.

A.1.3 Lexical Structure

The input sequence of characters are assumed to be Unicode 5.2.0[2]. The following tokens are defined during lexical analysis:

[1] In previous work, we published a technical report about MKL and a lambda calculus for connection semantics [17]. The semantics for that language and the one presented in this thesis are not the same.

[2] `http://www.unicode.org/`

- An *identifier* `IDENT` is a non-empty sequence of *letters* (`A`,...,`Z` and `a`,...,`z`), digits (`0`,...,`9`), and underscore (_), where the first character must not be a digit.
- An *unsigned integer constant* `UINT` is a non-empty sequence of digits (`0`,...,`9`).
- An *unsigned real constant* `UREAL` is an unsigned integer constant followed by a dot (`.`), possibly followed by an unsigned integer constant, possibly followed by exponential symbols (`E` and `e`), which is followed by an unsigned integer constant, possibly prepended with a sign character (+ or `-`). Examples `12.`, `91.2`, `2e8`, and `3.12e-2`.
- A *string constant* `STRING` is a sequence of character between quotes (`"`), where the character sequence can contain escape sequences, each starting with the escape character \. Possible escape sequences are horizontal tab (`\t`), newline (`\n`), double quote (`\"`), and escape character (`\\`).
- A *primitive constant* `PRIMITIVE` is a built-in constant or function, represented as a sequence of letters, where the start of the sequence is `@@`. For example `@@int_mul`.

A.1.4 Reserved Words

The following are reserved words, that may not be used as identifiers.

```
Array   DAESolver  Int   List     Map    Real
Set     String     else  error    false  fst
fun     if         in    include  let    match
snd     then       true  type     val    when
```

A.1.5 Top-Level

```
top ::=
      let letpat param { "->" param }  Let binder with params
      "=" expr top
    | let letpat "=" expr top          Let binder
    | let letpat ":" ty "=" expr top   Let binder with type
    | let letpat ":" ty top            Let unknown with type
    | type IDENT top                   Type declaration
    | type IDENT "=" ty top            Type synonymous
    | include IDENT top                Include file
    | EOF
```

A.1.6 Types

```
ty  ::=
      tyarrow                           Arrow type for Map
    | tyarrow "=>" ty

tyarrow ::=                             Arrow type for functions
```

```
      tyatom
    | tyatom "->" tyarrow

tyatom ::=
      IDENT                                Identifier
    | Int                                  Integer type
    | Real                                 Real type
    | Bool                                 Boolean type
    | String                               String type
    | "(" ")"                              Unit type
    | "[" ty "]"                           List type
    | List tyatom
    | "{" ty "}"                           Array type
    | Array tyatom
    | "(" ty { "," ty } ")"                Tuple type
    | "<" ">"                              Any model type
    | "<" ty ">"                           Specific model type
    | Map tyatom tyatom                    Map type
    | Set tyatom                           Set Type
    | DAESolver                            DAE Solver instance type
```

A.1.7 Expressions

```
expr ::=
      fun IDENT ":" tyatom "->" expr       Function abstraction
    | let letpat param { "->" param }      Local let binders
      "=" expr in
    | let pat_atom "=" expr in expr
    | let letpat ":" ty "=" expr in expr
    | if expr then expr else expr          IF-expression
    | let letpat ":" ty in expr            Local let unknown
    | match expr with matchcases           Match expression
    | Array "." IDENT { atom }+            Array operation
    | Map "." IDENT { atom }               Map operation
    | Set "." IDENT { atom }               Set operation
    | DAESolver "." IDENT { atom }         DAESolver operation
    | cons

letpat ::=                                  Simple let pattern
      IDENT
    | "_"

param ::=                                  Parameter with type
       IDENT ":" tyatom

cons ::=                                   Cons
      op
    | op "::" cons

op ::=                                     Operators
```

```
      app_left
    | op operator op

operator ::=
      "="   | "~=" | "mod" | "+"  | "-"
    | "*"   | "/"  | "<"   |"<="  | ">"
    | ">="  | "!=" | "+."  | "-." | "*."
    | "/."  | "<." | "<=." | ">." | ">=."
    | "!=." |  "!" | "&&"  | "||" | ";"
    | "++"  | "--" | "--." | "^"  | "^."

app_left ::=
      atom
    | app_left atom                   Application
    | fst atom                        First tuple element
    | snd atom                        Second tuple element
    | val atom                        Model value constructor
    | error atom                      Error

atom ::=
      IDENT                           Identifier
    | true                            True
    | false                           False
    | UINT                            Integer literal
    | UREAL                           Real (float) literal
    | STRING                          String literal
    | PRIMITIVE                       Primitive operation
    | "[" "]"                         Empty list
    | "[" expr { "," expr } "]"       List
    | "{" expr { "," expr } "}"       Array
    | "(" ")"                         Unit literal
    | "(" expr { "," expr } ")"       Tuple
```

A.1.8 Pattern Matching

```
matchcases ::=                        Match cases
      "|" pattern [when expr] "->" expr
    | matchcases "|" pattern [when expr]
      "->" expr

pattern ::=
      pat_op
    | pat_op "::" pattern             Cons pattern

pat_op ::=                            Pattern operator
      pat_left
    | pat_op OP pat_op

pat_left ::=
      pat_atom
```

```
    | pat_left pat_atom
    | fst pat_atom                          First elem of a tuple
    | snd pat_atom                          Second elem of a tuple
    | val IDENT ":" tyatom                  Model value pattern

pat_atom ::=
      IDENT                                 Pattern variable
    | true                                  True
    | false                                 False pattern
    | UINT                                  Unsigned Integer literal
    | UREAL                                 Unsigned Real literal
    | STRING                                String literal
    | "(" ")"                               Unit literal
    | "'" atom                              Pattern expression
    | "[" "]"                               Empty list
    | "[" pattern { "," pattern } "]"       List pattern

    | UK COLON tyatom                       Unknown pattern
    | "(" pattern { "," pattern } ")"       Tuple pattern
    | "_"                                   Wildcard pattern
```

A.2 Abstract Syntax

This section defines an abstract syntax for representing an intermediate language of MKL. The result of parsing the concrete syntax is *translated* into an abstract syntax tree described in this section. The most essential translation steps are:

- File inclusion (see Section 12.1.1).
- Desugar top-level constructs, meaning substitution of type synonymous and translation of top let-binders into local binders.
- Pattern compilation/translation. The process translates `match`-expressions into primitives for deconstructing models, lists and tuples.
- Type checking and model lifting (see Chapter 10 for a formal treatment of the core).

The objective of the abstract syntax for this intermediate language is to give the reader of this thesis a better understanding of which other language constructs that are part of the language, besides what was presented about the core. The aim is not to be a full language specification. Basic definitions:

Variables	$x, y, z \in \mathbb{X}$
Unknowns	$u \in \mathbb{U}$
Integers	$i \in \mathbb{Z}$
Constants	$c \in \mathbb{C} = \{\texttt{true}, \texttt{false}\} \cup \mathit{Int} \cup \mathit{Real} \cup \mathit{String}$

A.2.1 Types

Types in the language are defined as follows:

τ ::=	Bool	Boolean type
$\mid$	Int	Integer type
$\mid$	Real	Real type
$\mid$	String	String type
$\mid$	$\tau \rightarrow \tau$	Function type
$\mid$	()	Unit type
$\mid$	[τ]	List type
$\mid$	$(\tau_i{}^{i \in 1..n})$	Tuple type
$\mid$	$<\tau>$	Model type
$\mid$	<>	Any model type
$\mid$	Bot	Bot type
$\mid$	pseudo_i	User defined pseudo type
$\mid$	$\{\tau\}$	Array type
$\mid$	τ => τ	Map type
$\mid$	Set τ	Set type
$\mid$	DAESolver	DAESolver type

Comments:

- The four first types `Bool`, `Int`, `Real`, and `String` is represented as the ground type Γ in Chapter 10.
- All types, except for Bot type `Bot`, and the user defined pseudo type pseudo_i can be syntactically defined by the user (compare with the types of the concrete syntax).
- The bot type `Bot` is used as the type the element of an empty list or an empty array.
- Each pseudo type is assigned a unique id (i of pseudo_i). The pseudo type is created by giving it a name using the `type` syntax, e.g., **`type`** `Eqs` creates a new pseudo type. All places after this definition where type `Eqs` is used will be the same pseudo type.
- The `DAESolver` solver type is the type of an instance of the ADT DAESolver.

A.2.2 Expressions

e ::=	x	Variable
|	$\lambda x:\tau.e$	Lambda abstraction
|	$e\ e$	Application
|	c	Constant
|	$u:\tau$	Unknown
|	$\nu(\tau)$	New unknown creation
|	$e\ @\ e$	Model application
|	$\texttt{val}\ e:\tau$	Model value
|	$\texttt{decon}(e, d, e, e)$	Model deconstructor
|	$\texttt{fix}\ e$	Fixed-point
|	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$	If-expression
|	$e\ \texttt{==}\ e$	Equality test
|	$e\ \texttt{::}\ e$	List constructor
|	`[]`	Empty list
|	$\texttt{lcase}(e, x, x, e, e)$	List case
|	$\texttt{(}\,e_i\ ^{i\in 1..n}\,\texttt{)}$	Tuple
|	$\texttt{proj}\ i\ \texttt{from}\ e$	Projection
|	adt	Built-in ADT
|	$adtop\ e_i\ ^{i\in 1..n}$	Built-in ADT operation
|	$\texttt{error}\ e$	User defined error

Comments:

- The list case expression $\texttt{lcase}(v, x_1, x_1, e_1, e_2)$ deconstructs a list v. If v has the shape of a cons value v_1 `::` v_2, v_1 is substituted for x_1 and v_2 substituted for x_2 in e_1 that is the resulting expression. If v has the shape of an empty list, expression e_2 is the resulting expression.
- Projection $\texttt{proj}\ i\ \texttt{from}\ v$ returns element number i from v, where v is assumed to be have the shape of a tuple.
- Expression adt is the value of a type $\{\tau\}$, τ `=>` τ, `Set` τ, or `DAESolver`.
- Expression $adtop\ v_i\ ^{i\in 1..n}$ calls a built-in ADT operation.
- Expression `error` v stops execution of the program and returns an error message v, where v is assumed to have type `String`.

A.2.3 Values

v ::=	$\lambda x:\tau.e$	Lambda abstraction
|	c	Constant
|	$u:\tau$	Unknown
|	$v\ @\ v$	Model application
|	$\texttt{val}\ v:\tau$	Model value
|	$v\ \texttt{::}\ v$	List constructor
|	`[]`	Empty list
|	$\texttt{(}\,v_i\ ^{i\in 1..n}\,\texttt{)}$	Tuple
|	adt	Built-in ADT

B

Built-in Abstract Data Types

All types of functions are stated using curried form, but partial application is not syntactically allowed. Type variables for the built-in abstract data types (ADTs) are written with a prepended single quote, e.g., `'a`.

B.1 Array

Array operations for a random access array.

`Array``.length : {'a} -> `**`Int`**

An expression (**`Array`**`.length a`) evaluates to the length (the number of elements) of array `a`.

`Array``.make : `**`Int`**` -> 'a -> {'a}`

An expression (**`Array`**`.make n e`) creates a new array of length `n` filled with element `e`.

`Array``.get : {'a} -> `**`Int`**` -> 'a`

An expression (**`Array`**`.get a k`) evaluates to element with index `k` in array `a`. The first element has number 0. If `k` is outside range 0 to `Array.length a - 1`, then the program terminates.

`Array``.set : {'a} -> `**`Int`**` -> 'a -> ()`

An expression (**`Array`**`.set a k e`) destructively updates array `a` at index `k` with element `e`. If `k` is outside range 0 to `Array.length a - 1`, then the program terminates.

B.2 Set

A pure functional `Set`.

`Set.size : Set 'a -> Int`

An expression (`Set.size s`) evaluates to the cardinality (the number of elements) of set `s`.

`Set.empty : Set 'a`

An expression (`Set.empty`) evaluates to an empty set [1].

`Set.add : 'a -> Set 'a -> Set 'a`

An expression (`Set.add e s`) evaluates to a new set that contains all element of `s` plus element `e`.

`Set.mem : 'a -> Set 'a -> Bool`

An expression (`Set.mem e s`) evaluates to true if element `e` exists in set `s`, else false.

`Set.remove : 'a -> Set 'a -> Set 'a`

An expression (`Set.remove e s`) evaluates to a new set containing all elements of `s` except for `e`.

`Set.toList : Set 'a -> ['a]`

An expression (`Set.toList s`) evaluates to a list representation of set `s`.

B.3 Map

A purely functional finite map.

`Map.size : ('a => 'b) -> Int`

An expression (`Map.size m`) evaluates to the number of elements in `m`.

`Map.empty : ('a => 'b)`

An expression (`Map.empty`) evaluates to a new empty map.

[1] The type of the element of the `Set` is here expressed as a type variable, but will internally use type `Bot` during type checking.

Map.add : ’a -> ’b -> (’a => ’b) -> (’a => ’b)

An expression (**Map**.add k v m) evaluates to a new map that contains all key/value pairs of map m plus a new a binding between key k and value v. If k already exists in m, the previous binding is removed.

Map.find : ’a -> (’a => ’b) -> ’b

An expression (**Map**.find k m) evaluates to the value bound to key k in map m. It terminates the program if the element is not found. Note: use Map.mem before calling Map.find.

Map.mem : ’a -> (’a => ’b) -> **Bool**

An expression (**Map**.mem k m) evaluates to **true** if there exists a binding of key k in map m, else **false**.

Map.remove : ’a -> (’a => ’b) -> (’a => ’b)

An expression (**Map**.remove k m) evaluates to a new map containing all key/value pairs in map m, except for a binding of key k that is removed.

Map.toList : (’a => ’b) -> [(’a,’b)]

An expression (**Map**.toList m) evaluates to a list of tuples containing all key/value pairs in map m.

B.4 DAESolver

ADT DAESolver interfaces the IDA solver from the SUNDIALS suite [68].

```
DAESolver.make : {Real} -> {Real} -> {Real} ->
                 (Real -> {Real} -> {Real} -> [Real]) -> DAESolver
```

An expression (`DAESolver.make yy yp id res`) evaluates to a new instance of a DAE solver. Argument `yy` is an array of initial values for vector y and argument `yp` the initial values for vector $\dot{y}$. Argument `id` is an array of `Real` specifying a differential variable (value 1.0) or an algebraic variable (value 0.0). Argument `id` is used for correction of initial conditions. Argument `res` is the supplied residual function that has type `(Real -> {Real} -> {Real} -> [Real])`, where its first parameter is the independent variable of time, parameter 1 the dependent variable vector $y(t)$ and parameter 2 the vector $\dot{y}(t)$. The output residual is returned as a list of `Real`.

```
DAESolver.step : Real -> DAESolver -> Real
```

An expression (`DAESolver.step t s`) integrates the DAE over a time, where `t` is the next time a computed result is desired and `s` is the DAESolver instance. The result of the function call is the time reached by the solver. If the returned time is zero, an error occurred. The result of the computation is destructively updated in the arrays supplied to `DAESolver.make`.

C

Big-step Semantics of MKL Core

Evaluation Rules $\boxed{e \mid U \;\Rightarrow\; e \mid U}$

$$\frac{}{\lambda x\!:\!\tau.e \mid U_1 \;\Rightarrow\; \lambda x\!:\!\tau.e \mid U_1}\;\text{(BS-ABS)} \qquad \frac{}{u\!:\!\tau \mid U_1 \;\Rightarrow\; u\!:\!\tau \mid U_1}\;\text{(BS-UK)}$$

$$\frac{\begin{array}{c} e_1 \mid U_1 \;\Rightarrow\; \lambda x\!:\!\tau.e_3 \mid U_2 \\ e_2 \mid U_2 \;\Rightarrow\; v_1 \mid U_3 \qquad [x \mapsto v_1]e_3 \mid U_3 \;\Rightarrow\; v_2 \mid U_4 \end{array}}{e_1\ e_2 \mid U_1 \;\Rightarrow\; v_2 \mid U_4}\;\text{(BS-APPABS)}$$

$$\frac{}{c \mid U_1 \;\Rightarrow\; c \mid U_1}\;\text{(BS-CONST)} \qquad \frac{\begin{array}{c} e_1 \mid U_1 \;\Rightarrow\; c \mid U_2 \\ e_2 \mid U_2 \;\Rightarrow\; v_2 \mid U_3 \qquad v_3 = \delta(c, v_2) \end{array}}{e_1\ e_2 \mid U_1 \;\Rightarrow\; v_3 \mid U_3}\;\text{(BS-APPCONST)}$$

$$\frac{u \notin U_1}{\nu(\tau) \mid U_1 \;\Rightarrow\; u\!:\!\texttt{<}\tau\texttt{>} \mid U_1 \cup \{u\}}\;\text{(BS-NEWUK)}$$

$$\frac{e_1 \mid U_1 \;\Rightarrow\; v_1 \mid U_2 \qquad e_2 \mid U_2 \;\Rightarrow\; v_2 \mid U_3}{e_1 @\, e_2 \mid U_1 \;\Rightarrow\; v_1 @\, v_2 \mid U_3}\;\text{(BS-MODAPP)}$$

$$\frac{e_1 \mid U_1 \;\Rightarrow\; v_1 \mid U_2}{\texttt{val}\ e_1\!:\!\tau \mid U_1 \;\Rightarrow\; \texttt{val}\ v_1\!:\!\tau \mid U_2}\;\text{(BS-MODVAL)}$$

$$\frac{e_1 \mid U_1 \;\Rightarrow\; v_1 \mid U_2 \qquad \mathit{match}(v_1, d, e_2, e_2') \qquad e_2' \mid U_2 \;\Rightarrow\; v_2 \mid U_3}{\texttt{decon}(e_1, d, e_2, e_3) \mid U_1 \;\Rightarrow\; v_2 \mid U_3}\;\text{(BS-DECON-T)}$$

$$\frac{e_1 \mid U_1 \;\Rightarrow\; v_1 \mid U_2 \qquad \neg\mathit{match}(v_1, d, e_2, e_2') \qquad e_3 \mid U_2 \;\Rightarrow\; v_3 \mid U_3}{\texttt{decon}(e_1, d, e_2, e_3) \mid U_1 \;\Rightarrow\; v_3 \mid U_3}\;\text{(BS-DECON-F)}$$

Match $\boxed{\mathit{match}(e, d, e, e)}$

$$\mathit{match}(u : \tau, \texttt{uk} : \tau, e, e) \quad \text{(BS-M-UK)}$$

$$\mathit{match}(\texttt{val}\ v : \tau, \texttt{val}\ x : \tau, e, (\lambda x : \tau . e)\ v) \quad \text{(BS-M-MVAL)}$$

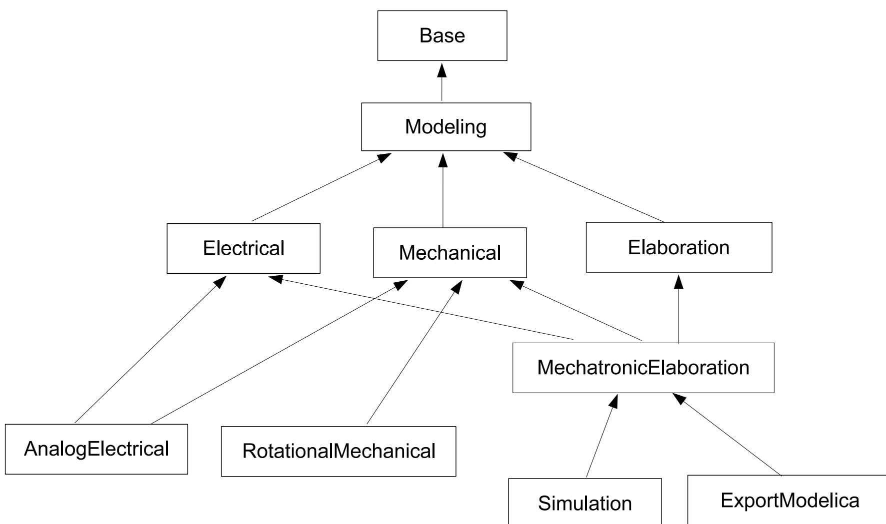
$$\mathit{match}(v_1 @ v_2, x_1 @ x_2, e, (\lambda x_1 : \texttt{<>} . \lambda x_2 : \texttt{<>} . e)\ v_1\ v_2) \quad \text{(BS-M-MAPP)}$$

The big-step semantics above is defined for language $\lambda_L^{\texttt{<>}}$ using syntax in Figure 10.1 on page 138.

D

MKL Library

This appendix lists the MKL source code of the MKL library. The figure below outlines dependencies between different modules in the library.



D.1 Base

```
let (mod) : Int -> Int -> Int = @@int_mod
let (+) : Int -> Int -> Int = @@int_add
let (-) : Int -> Int -> Int = @@int_sub
let (*) : Int -> Int -> Int = @@int_mul
let (/) : Int -> Int -> Int = @@int_div
let (<) : Int -> Int -> Bool = @@int_less
let (<=) : Int -> Int -> Bool = @@int_less_equal
let (>) : Int -> Int -> Bool = @@int_great
let (>=) : Int -> Int -> Bool = @@int_great_equal
let (!=) : Int -> Int -> Bool = @@int_not_equal
let (+.) : Real -> Real -> Real = @@real_add
let (-.) : Real -> Real -> Real = @@real_sub
let (*.) : Real -> Real -> Real = @@real_mul
let (/.) : Real -> Real -> Real = @@real_div
let (<.) : Real -> Real -> Bool = @@real_less
let (<=.) : Real -> Real -> Bool = @@real_less_equal
let (>.) : Real -> Real -> Bool = @@real_great
let (>=.) : Real -> Real -> Bool = @@real_great_equal
let (!=.) : Real -> Real -> Bool = @@real_not_equal
let (!) : Bool -> Bool = @@bool_not
let (&&) : Bool -> Bool -> Bool = @@bool_and
let (||) : Bool -> Bool -> Bool = @@bool_or
let (--) : Int -> Int = @@int_neg
let (--.) : Real -> Real = @@real_neg
let print : String -> () = @@print
let bool2string : Bool -> String = @@bool2string
let int2string : Int -> String = @@int2string
let real2string : Real -> String = @@real2string
let int2real : Int -> Real = @@int2real
let real2int : Real -> Int = @@real2int
let string2bool : String -> Bool = @@string2bool
let string2int : String -> Int = @@string2int
let string2real : String -> Real = @@string2real
let isBoolString : String -> Bool = @@isboolstring
let isRealString : String -> Bool = @@isrealstring
let isIntString : String -> Bool = @@isintstring
let sin : Real -> Real = @@sin
let cos : Real -> Real = @@cos
let tan : Real -> Real = @@tan
let asin : Real -> Real = @@asin
let acos : Real -> Real = @@acos
let atan : Real -> Real = @@atan
let sinh : Real -> Real = @@sinh
let cosh : Real -> Real = @@cosh
let tanh : Real -> Real = @@tanh
let ceil : Real -> Real = @@ceil
let floor : Real -> Real = @@floor
let log : Real -> Real = @@log
let log10 : Real -> Real = @@log10
let sqrt : Real -> Real = @@sqrt
let exp : Real -> Real = @@exp
let (^.) : Real -> Real -> Real = @@exponentiation
let substr : String -> Int -> Int -> String = @@string_substr
let strlen : String -> Int = @@string_strlen
let (++) : String -> String -> String = @@string_concat
```

```
let printLine s:String -> () =
    let _ = print s in print "\n"

let printIntLine i:Int -> () =
    printLine  (int2string i)

let printRealLine i:Real -> () =
    printLine  (real2string i)
```

D.2 Modeling

```
include Base

type Eqs
type Equations = <Eqs>
type Signal = <Real>
let der : <Real -> Real>
let EquationSysNode : <Eqs -> Eqs -> Eqs>
let Eq : <Real -> Real -> Eqs>
let Init : <Real -> Real -> Eqs>
let InitGuess : <Real -> Real -> Eqs>
let time : <Real>
let probe : <String -> Real>

let (=) : <Real -> Real -> Eqs> = Eq
let (;) : <Eqs -> Eqs -> Eqs> = EquationSysNode
```

D.3 Electrical

```
include Modeling

type ElectricalNode
type Electrical = <ElectricalNode>
let ElectricalBranch :
    <Real -> Real -> ElectricalNode -> ElectricalNode -> Eqs>
let ElectricalRefBranch : <Real -> Real -> ElectricalNode -> Eqs>
type Voltage = <Real>
type Current = <Real>
```

D.4 AnalogElectrical

```
include Electrical
include Mechanical

let Resistor R:Real -> p:Electrical -> n:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  R *. i = v

let Capacitor C:Real -> p:Electrical -> n:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
```

```
  ElectricalBranch i v p n;
  C *. (der v) = i

let Inductor L:Real -> p:Electrical -> n:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  L *. (der i) = v

let Ground p:Electrical -> Equations =
  let i:Current in
  let v:Voltage in
  ElectricalRefBranch i v p;
  v = 0.

let SineVoltage V:Real -> f:Real -> p:Electrical -> n:Electrical ->
                Equations =
  let PI = 3.1415 in
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  v = V *. sin(2. *. PI *. f *. time)

let ConstantVoltage V:Real -> p:Electrical -> n:Electrical ->  Equations =
  let i:Current in
  let v:Voltage in
  ElectricalBranch i v p n;
  v = V

let EMF k:Real -> p:Electrical -> n:Electrical -> flange:Rotational ->
                  Equations =
  let i:Current in
  let v:Voltage in
  let w:AngularVelocity in
  let phi:Angle in
  let tau:Torque in
  ElectricalBranch i v p n;
  RotationalRefBranch tau phi flange;
  w = der(phi);
  k *. w = v;
  tau =  k *. i

let VoltageSensor output:Signal -> p:Electrical -> n:Electrical ->
                  Equations =
  ElectricalBranch 0. output p n

let PotentialSensor output:Signal -> p:Electrical -> Equations =
  ElectricalRefBranch 0. output p

let CurrentSensor output:Signal -> p:Electrical -> n:Electrical ->
                  Equations =
  ElectricalBranch output 0. p n
```

D.5 Mechanical

```
include Modeling
```

```
type RotationalNode
type Rotational = <RotationalNode>
let RotationalBranch :
    <Real -> Real -> RotationalNode -> RotationalNode -> Eqs>
let RotationalRefBranch : <Real -> Real -> RotationalNode -> Eqs>
type AngularVelocity = <Real>
type AngularAcceleration = <Real>
type Angle = <Real>
type Torque = <Real>
```

D.6 RotationalMechanical

```
include Mechanical

let Spring c:Real -> flangeA:Rotational -> flangeB:Rotational ->
           Equations =
  let tau:Torque in
  let relphi:Angle in
  RotationalBranch tau relphi flangeB flangeA;
  tau = c *. relphi

let Damper d:Real -> flangeA:Rotational -> flangeB:Rotational ->
           Equations =
  let tau:Torque in
  let relphi:Angle in
  RotationalBranch tau relphi flangeB flangeA;
  tau = d *. der(relphi)

let Inertia J:Real -> flangeA:Rotational -> flangeB:Rotational ->
    Equations =
  let tauA:Torque in
  let tauB:Torque in
  let phiA:Angle in
  let phiB:Angle in
  let phi:Angle in
  let w:AngularVelocity in
  let a:AngularAcceleration in
  RotationalRefBranch tauB phiB flangeB;
  RotationalRefBranch tauA phiA flangeA;
  phiA = phi;
  phiB = phi;
  w = der(phi);
  a = der(w);
  J *. a = tauA +. tauB

let IdealGear ratio:Real -> flangeA:Rotational -> flangeB:Rotational ->
    Equations =
  let tauA:Torque in
  let tauB:Torque in
  let phiA:Angle in
  let phiB:Angle in
  RotationalRefBranch tauA phiA flangeA;
  RotationalRefBranch tauB phiB flangeB;
  phiA = ratio *. phiB;
  0. = ratio *. tauA +. tauB

let Fixed angle:Real -> flangeB:Rotational -> Equations =
```

```
  let tau:Torque in
  RotationalRefBranch tau angle flangeB

let ConstantTorque tau:Real -> flangeB:Rotational -> Equations=
  let phi:Angle in
  RotationalRefBranch tau phi flangeB

let Torque tau:Signal -> flangeB:Rotational -> Equations =
  let phi:Angle in
  RotationalRefBranch tau phi flangeB

let TorqueSensor tau:Signal -> flangeA:Rotational ->
                flangeB:Rotational -> Equations =
  RotationalBranch tau 0. flangeB flangeA

let AngleSensor phi:Signal -> flangeB:Rotational -> Equations =
  RotationalRefBranch 0. (-.phi) flangeB

let SpeedSensor w:Signal -> flangeB:Rotational -> Equations =
  let phi:Angle in
  RotationalRefBranch 0. (-.phi) flangeB;
  w = der(phi)

let AccSensor a:Signal -> flangeB:Rotational -> Equations =
  let phi:Angle in
  let w:AngularVelocity in
  RotationalRefBranch 0. (-.phi) flangeB;
  w = der(phi);
  a = der(w)

let RelAngleSensor phiRel:Signal -> flangeA:Rotational ->
                   flangeB:Rotational -> Equations =
  RotationalBranch 0. phiRel flangeB flangeA

let RelSpeedSensor w:Signal -> flangeA:Rotational ->
                   flangeB:Rotational -> Equations =
  let phiRel:Angle in
  RotationalBranch 0. phiRel flangeB flangeA;
  w = der(phiRel)

let RelAccSensor a:Signal -> flangeA:Rotational ->
                flangeB:Rotational -> Equations =
  let phiRel:Angle in
  let w:AngularVelocity in
  RotationalBranch 0. phiRel flangeB flangeA;
  w = der(phiRel);
  a = der(w)
```

D.7 Elaboration

```
include Modeling

type Expr = <>
type Node = <>
type Unknown = <>
type Branch = <>
type NodeMap = (Node => Unknown)
```

```
type NodeMapList = [(Node,Unknown)]
type BranchSet = (Set Branch)
type BranchList = [Branch]
type ExprMap = (Node => Expr)
type ExprList = [(Node, Expr)]
type ProbeMap =  (String => [Signal])
type InitValMap = (<Real> => (Real,Bool))
type UkSet = (Set <Real>)
type UkMap = (Signal => (Int,Bool))

let elaborateConnections isBranch:(<> -> Bool) ->
                         isRefBranch:(<> -> Bool) ->
                         model:Equations ->
                         Equations =
  let addNode node:Node -> nodemap:NodeMap -> NodeMap =
    if Map.mem node nodemap then nodemap
    else let u:<Real> in Map.add node u nodemap
  in

  let sumexpr branches:BranchSet -> ExprMap =
    let worker branches:BranchList -> emap:ExprMap -> ExprMap =
      match branches with
      | (b i v p n)::bs when isBranch b ->
          let emap1 = if Map.mem p emap
                      then Map.add p ((Map.find p emap) +. i) emap
                      else Map.add p i emap in
          let emap2 = if Map.mem n emap1
                      then Map.add n ((Map.find n emap1) -. i) emap1
                      else Map.add n (-. i) emap1 in
          worker bs emap2
      | (b i v p)::bs when isRefBranch b ->
          let emap1 = if Map.mem p emap
                      then Map.add p ((Map.find p emap) +. i) emap
                      else Map.add p i emap in
          worker bs emap1
      | [] -> emap
    in worker (Set.toList branches) (Map.empty)
  in

  let sumzero m:Equations -> branches:BranchSet -> Equations =
    let worker elist:ExprList -> Equations =
      match elist with
      | (_,e)::es -> e = 0.; worker es
      | [] -> m
    in worker (Map.toList (sumexpr branches))
  in

  let potentials model:Equations -> (Equations,BranchSet) =
    let worker m:Equations -> nodemap:NodeMap ->
               branchset:BranchSet -> (<>,NodeMap,BranchSet) =
      match m with
      | b i v p n when isBranch b ->
        let nodemap2 = addNode n (addNode p nodemap) in
        let eq = (v = (Map.find p nodemap2) -.
                  (Map.find n nodemap2)) in
        (eq,nodemap2,Set.add m branchset)
      | b i v p when isRefBranch b ->
        let nodemap2 = addNode p nodemap in
```

```
        let eq = (v = (Map.find p nodemap2)) in
        (eq,nodemap2,Set.add m branchset)
      | e1 ; e2 ->
         let (e1b,nodemap1,branchset1) =
               worker e1 nodemap branchset in
         let (e2b,nodemap2,branchset2) =
               worker e2 nodemap1 branchset1 in
         (e1b ; e2b, nodemap2, branchset2)
      | _ ->  (m,nodemap,branchset)
    in
      let (model,_,branchset) =
        worker model (Map.empty) (Set.empty)
      in (model,branchset)
  in
     let (model2,branchset2) = potentials model in
     sumzero model2 branchset2

let addProbe s:String -> u:Signal -> ps:ProbeMap =
    if Map.mem s ps then Map.add s (u::(Map.find s ps)) ps
    else Map.add s [u] ps

let elaborateProbes model:Equations -> (Equations,ProbeMap) =
  let elab e:<> -> ps:ProbeMap -> (<>,ProbeMap) =
    match e with
    | 'probe (val s:String) -> let u:Signal in (u,addProbe s u ps)
    | e1 e2 ->
       let (e1b,ps1) = elab e1 ps in
        let (e2b,ps2) = elab e2 ps1 in
       (e1b e2b,ps2)
    | _ -> (e,ps)
  in elab model (Map.empty)

let initValues eqs:Equations -> InitValMap =
  let get eqs:Equations -> acc:InitValMap -> InitValMap =
    match eqs with
    | e1 ; e2 -> get e2 (get e1 acc)
    | Init x (val v:Real) -> Map.add x (v,false) acc
    | InitGuess x (val v:Real) -> Map.add x (v,true) acc
    | _ -> acc
  in get eqs (Map.empty)

let unknowns eqs:Equations -> UkSet =
  let get e:<> -> acc:UkSet -> UkSet =
     match e with
     | e1 e2 -> get e2 (get e1 acc)
     | 'time -> acc
     | uk:<Real> -> Set.add e acc
     | _ -> acc
  in get eqs (Set.empty)

let noUnknowns eqs:Equations -> Int =
  Set.size (unknowns eqs)

let noEquations eqs:Equations -> Int =
  match eqs with
  | e1 ; e2 -> (noEquations e1) + (noEquations e2)
  | e1 = e2 -> 1
  | _ -> 0
```

```
let realUnknowns eqs:Equations -> (Int,UkMap) =
  let get e:<> -> acc:(Int,UkMap) -> (Int,UkMap) =
     match e with
     | e1 e2 -> get e2 (get e1 acc)
     | 'time -> acc
     | der x ->
        (match x with
         | uk:<Real> ->
             let (k,ukmap) = acc in
             if Map.mem e ukmap
             then (k,Map.add e (fst (Map.find e ukmap),true) ukmap)
             else ((k+1,Map.add e (k,true) ukmap))
         | _ -> error "Illegal diffrentiation of expression")
     | uk:<Real> ->
         if Map.mem e (snd acc) then acc
         else ((fst acc)+1,Map.add e (fst acc,false) (snd acc))
     | e1 -> acc
  in get eqs (0,(Map.empty))

let makeUkMap eqs:Equations -> UkMap =
    snd (realUnknowns eqs)
```

D.8 MechatronicElaboration

```
include Mechanical
include Electrical
include Elaboration

let isMechatronicBranch b:<> =
  match b with
  | 'ElectricalBranch -> true
  | 'RotationalBranch -> true
  | _ -> false

let isMechatronicRefBranch b:<> =
  match b with
  | 'ElectricalRefBranch -> true
  | 'RotationalRefBranch -> true
  | _ -> false

let elaborate model:Equations -> (Equations,ProbeMap) =
  let (model2,probes) = elaborateProbes model in
  (elaborateConnections isMechatronicBranch
   isMechatronicRefBranch model2,probes)
```

D.9 Simulation

```
include MechatronicElaboration

type StepVal = (Real,[Real])
type SimulationResult = ([String],[StepVal])

type Residual = Real -> {Real} -> {Real} -> [Real]

let makeResidual model:Equations -> ukmap:UkMap ->
```

```
                 ctime:Real -> yy:{Real} -> yp:{Real} -> [Real] =
  let realExpr e:<> -> Real =
    match e with
    | (val f:(Real -> Real -> Real)) e1 e2 ->
          f (realExpr e1) (realExpr e2)
    | (val f:(Real -> Real)) e1 -> f  (realExpr e1)
    | val v:Real -> v
    | 'time -> ctime
    | der x -> (match x with
                | uk:Signal -> Array.get yp (fst (Map.find x ukmap))
                | _ -> error "Derivatives only allowed on unknowns")
    | uk:Signal -> Array.get yy (fst(Map.find e ukmap))
    | _ -> error "Unsupported model construct"
  in
  let traverseEq m:Equations -> acc:[Real] -> [Real] =
    match m with
    | e1 ; e2 -> traverseEq e2 (traverseEq e1 acc)
    | e1 = e2 -> (realExpr e1 -. realExpr e2) :: acc
    | _ -> acc
  in traverseEq model []

let makeInitValues model:Equations -> ukmap:UkMap -> ({Real},{Real}) =
  let initvalmap = initValues model in
  let size = Map.size ukmap in
  let yy = Array.make size 0. in
  let yp = Array.make size 0. in
  let setvals initvals:[(Signal,(Real,Bool))] -> {Real} =
    match initvals with
    | (u,(v,_))::xs ->
       let _ = Array.set yy (fst (Map.find u ukmap)) v in setvals xs
    | [] -> yy
  in (setvals (Map.toList initvalmap) , yp)

let makeStepVal currtime:Real -> yy:{Real} ->
                probes:[(String,[Signal])] -> ukmap:UkMap -> StepVal =
  let mkvals probes:[(String,[Signal])] -> [Real] =
    match probes with
    | (s,u::us)::ss -> (Array.get yy (fst(Map.find u ukmap)))::
                         (mkvals ((s,us)::ss))
    | (s,[])::ss -> mkvals ss
    | [] -> []
  in (currtime,mkvals probes)

let makeProbeNames probes:[(String,[Signal])] -> n:Int -> [String] =
  match probes with
  | (s,u::us)::ss ->
    let s2 = s ++ (if n > 1 then "_" ++ int2string n else "") in
    s2::(makeProbeNames ((s,us)::ss) (n+1))
  | (s,[])::ss -> makeProbeNames ss 1
  | [] -> []

let initConditionCorrection ukmap:UkMap -> {Real} =
  let id = Array.make (Map.size ukmap) 0. in
  let worker uklist:[(Signal,(Int,Bool))] -> {Real} =
    match uklist with
    | (u,(i,true))::us -> let _ = Array.set id i 1.0 in worker us
    | _::us -> worker us
    | [] -> id
```

```
  in worker (Map.toList ukmap)

let revResult lst:[StepVal] -> acc:[StepVal] -> [StepVal] =
  match lst with
  | x::xs -> revResult xs (x::acc)
  | [] -> acc

let simulate model:Equations -> steptime:Real -> endtime:Real ->
             SimulationResult =
  let (model2,probes) = elaborate model in
  let probelist = Map.toList probes in
  let ukmap = makeUkMap model2 in
  let residual = makeResidual model2 ukmap in
  let (yy,yp) = makeInitValues model2 ukmap in
  let id = initConditionCorrection ukmap in
  let state = DAESolver.make yy yp id residual in
  let simloop currtime:Real -> acc:[StepVal] -> [StepVal] =
    if currtime >=. endtime then acc
    else
      let stepval = makeStepVal currtime yy probelist ukmap in
      let newtime = DAESolver.step steptime state in
      if newtime == 0. then acc
      else simloop (newtime +. steptime) (stepval::acc)
  in
    (makeProbeNames probelist 1,revResult (simloop 0. []) [])

let pprintSimulation res:SimulationResult -> String =
  let (names,stepvals) = res in
  let psteps step:[StepVal] -> first:Bool -> String =
    match step with
    | (t,x::xs)::xxs when first ->
        real2string t ++ "\t" ++ psteps ((t,x::xs)::xxs) false
    | (t,x::xs)::xxs ->
        real2string x ++ "\t" ++ psteps ((t,xs)::xxs) false
    | (t,[])::xxs -> "\n" ++ psteps xxs true
    | [] -> ""
  in psteps stepvals true

let printsim model:Equations -> steptime:Real -> endtime:Real -> () =
      print (pprintSimulation (simulate model steptime endtime))
```

D.10 Export Modelica

```
include MechatronicElaboration

let mkvar uks:(Int,UkMap) -> e:<> -> String =
  match uks with
  | (_,ma) -> "uk" ++ int2string (fst (Map.find e ma))

let pprintModelica name:String ->
                   probelist:[(String,[Signal])] ->
                   model:Equations -> String =
  let uks = realUnknowns model in
  let real2modelicaString r:Real -> String =
    if r <. 0. then "(" ++ real2string r ++ ")" else real2string r
  in
  let pprintUnknowns us:(Int,UkMap) -> initvalmap:InitValMap -> String =
```

```
    let initstr u:Signal -> String =
      if Map.mem u initvalmap then
         let (initval,guess) = Map.find u initvalmap in
         "(start=" ++ real2string initval ++
          (if guess then "" else ",fixed=true") ++ ")"
      else ""
    in
    let prn us:[(Signal,(Int,Bool))] -> String =
      match us with
      | (u,_)::us -> "  Real " ++ mkvar uks u ++
                    initstr u ++ ";\n" ++ prn us
      | [] -> ""
    in prn (Map.toList (snd us))
  in
  let pprintExpr expr:<> -> String =
    match expr with
    | e1 +. e2 -> "(" ++ pprintExpr e1 ++ " + " ++ pprintExpr e2 ++ ")"
    | e1 -. e2 -> "(" ++ pprintExpr e1 ++ " - " ++ pprintExpr e2 ++ ")"
    | e1 *. e2 -> pprintExpr e1 ++ " * " ++ pprintExpr e2
    | e1 /. e2 -> pprintExpr e1 ++ " / " ++ pprintExpr e2
    | -. e ->     "(-" ++ pprintExpr e ++ ")"
    | e1 ^. e2 -> pprintExpr e1 ++ " ^ " ++ "(" ++ pprintExpr e2 ++ ")"
    | 'der e ->   "der(" ++ pprintExpr e ++ ")"
    | 'sin e ->   "sin(" ++ pprintExpr e ++ ")"
    | 'cos e ->   "cos(" ++ pprintExpr e ++ ")"
    | 'tan e ->   "tan(" ++ pprintExpr e ++ ")"
    | 'asin e ->  "asin(" ++ pprintExpr e ++ ")"
    | 'acos e ->  "acos(" ++ pprintExpr e ++ ")"
    | 'atan e ->  "atan(" ++ pprintExpr e ++ ")"
    | 'sinh e ->  "sinh(" ++ pprintExpr e ++ ")"
    | 'cosh e ->  "cosh(" ++ pprintExpr e ++ ")"
    | 'tanh e ->  "tanh(" ++ pprintExpr e ++ ")"
    | 'sqrt e ->  "sqrt(" ++ pprintExpr e ++ ")"
    | 'exp e ->   "exp(" ++ pprintExpr e ++ ")"
    | 'log e ->   "log(" ++ pprintExpr e ++ ")"
    | 'log10 e -> "log10(" ++ pprintExpr e ++ ")"
    | 'time -> "time"
    | uk:<Real> -> mkvar uks expr
    | val r:Real -> real2modelicaString r
  in
  let pprintEqs model:Equations -> String =
    match model with
    | e1 ; e2 -> pprintEqs e1 ++ pprintEqs e2
    | e1 = e2 -> "  " ++ pprintExpr e1 ++ " = " ++ pprintExpr e2 ++ ";\n"
    | _ -> ""
  in
  let pprintOutput probes:[(String,[Signal])] -> n:Int -> String =
    match probes with
    | (s,u::us)::ss ->
        "  output Real " ++ s ++ (if n > 1 then "_" ++ int2string n
                                             else "") ++
        " = " ++ mkvar uks u ++ ";\n" ++ pprintOutput ((s,us)::ss) (n+1)
    | (s,[])::ss -> pprintOutput ss 1
    | [] -> ""
  in
    "model " ++ name ++ "\n" ++ pprintOutput probelist 1 ++
    "protected\n" ++ pprintUnknowns uks (initValues model) ++
    "equation\n" ++ pprintEqs model ++ "end " ++ name ++ ";"
```

```
let exportModelica name:String -> model:Equations -> String =
  let (model2,probes) = elaborate model in
  pprintModelica name (Map.toList probes) model2
```

D.11 Performance Test Source Code

The following program lists the MKL source code for the programs used in performance testing. Note that MechSys is including MechsysBasics.

```
include AnalogElectrical
include RotationalMechanical

let ShaftElement flangeA:Rotational -> flangeB:Rotational ->
                 Equations =
  let r1:Rotational in
  Spring 8. flangeA r1;
  Damper 1.5 flangeA r1;
  Inertia 0.5 r1 flangeB

let DCMotor flange:Rotational -> Equations =
  let e1:Electrical in
  let e2:Electrical in
  let e3:Electrical in
  let e4:Electrical in
  ConstantVoltage 60. e1 e4;
  Resistor 100. e1 e2;
  Inductor 0.2 e2 e3;
  EMF 1. e3 e4 flange;
  Ground e4

let FlexibleShaft n:Int -> flangeA:Rotational -> flangeB:Rotational ->
                  Equations =
  if n == 1 then
    ShaftElement flangeA flangeB
  else
    let r1:Rotational in
    ShaftElement flangeA r1;
    FlexibleShaft (n-1) r1 flangeB
```

D.11.1 MechSys

```
include MechsysBasics
include MechatronicElaboration

let MechSys =
  let r1:Rotational in
  let r2:Rotational in
  let r3:Rotational in
  DCMotor r1;
  Inertia 0.2 r1 r2;
  FlexibleShaft 40000 r2 r3

let main =
  let (eqs,probes) = elaborate MechSys in
```

```
  print ("Unknowns: " ++ int2string (noUnknowns eqs) ++ "\n" ++
         "Equations: " ++ int2string (noEquations eqs) ++ "\n")
```

D.11.2 CircuitHierarchy

```
include MechatronicElaboration
include ExportModelica
include AnalogElectrical

let SubComponent p:Electrical -> n:Electrical =
  let e1:Electrical in
  Resistor 150. p e1;
  Inductor 0.1 e1 n;
  Capacitor 0.01 e1 n

let RecComp lev:Int -> p:Electrical -> n:Electrical -> Equations =
  let e1:Electrical in
  if lev == 1 then SubComponent p n else
  RecComp (lev-1) p e1;
  RecComp (lev-1) e1 n;
  RecComp (lev-1) e1 n

let Circuit =
  let ee1:Electrical in
  let ee2:Electrical in
  SineVoltage 220. 50. ee1 ee2;
  RecComp 11 ee1 ee2;
  Ground ee2

let main =
  let (eqs,probes) = elaborate Circuit in
  print ("Unknowns: " ++ int2string (noUnknowns eqs) ++ "\n" ++
         "Equations: " ++ int2string (noEquations eqs) ++ "\n")
```

Notation

Symbols and Operators

x	State vector
y	Measurement signal
u	Known input signal
$\mathbb{R}$	The set of real numbers
$\mathbb{N}$	The set of natural numbers $\{0, 1, 2, 3, \ldots\}$.
$\forall x$	For all x (universal quantifier)
$\exists x$	For some x (existential quantifier)
$\neg p$	Negation of p
$p \wedge q$	Conjunction of p and q
$p \vee q$	Disjunction of p and q
iff	If and only if
$x \in A$	Element x is a member of set A
$y \notin A$	Element x is not an element of set A
$A \setminus B$	Difference of set A and set B
$A \cup B$	Union of set A and set B
$A \cap B$	Intersection of set A and set B
$L_1 \oplus L_2$	Appends list L_2 to L_1
Γ	Typing Environment
τ	Type
e	Expression

Abbreviations and Acronyms

ADT	Abstract Data Types

AST	Abstract Syntax Tree
BNF	Backus-Naur Form
BTA	Binding-Time Analysis
CBN	Call-by-name
CBV	Call-by-value
CPS	Cyber-Physical System
DAE	Differential-Algebraic Equation
DSL	Domain-Specific Language
EBNF	Extended Backus-Naur Form
FP	Functional Programming
DSP	Domain-Specific Processor
EOO	Equation-Based Object-Oriented
FHM	Functional Hybrid Modeling
FRP	Functional Reactive Programming
GADT	Generalized Algebraic Data Type
GUI	Graphical User Interface
HDL	Hardware Description Language
HOAM	Higher-Order Acausal Models
JIT	Just-in-time
MDA	Model Driven Architecture
MKL	Modeling Kernel Language
MSL	Modelica Standard Library
ODE	Ordinary Differential Equation
OOP	Object-Oriented Programming
PE	Partial Evaluation
SUNDIALS	SUite of Nonlinear and DIfferential/ALgebraic equation Solvers
UML	Unified Modeling Language
YACC	Yet Another Compiler Compiler

Bibliography

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, 1991.

[2] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, USA, 1996.

[3] Accellera Organization. Verilog-AMS Language Reference Manual - Analog & Mixed-Signal Extensions to Verilog HDL Version 2.3.1, 2009. Available from: `http://www.vhdl.org/verilog-ams/` [Last accessed: July 30, 2010].

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, MA, USA, 2nd edition, 2006.

[5] Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and Optimization with Optimica and JModelica.org—Languages and Tools for Solving Large-Scale Dynamic Optimization Problem. *Computers and Chemical Engineering*, January 2010. Doi:10.1016/j.compchemeng.2009.11.011.

[6] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[7] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[8] Peter Aronsson and David Broman. Extendable Physical Unit Checking with Understandable Error Reporting. In *Proceedings of the 7th International Modelica Conference*, pages 890–897, Como, Italy, 2009.

[9] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus, and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In *Proceedings of the 3rd International ModelicaConference*, pages 431–440, Linköping, Sweden, 2003.

[10] Peter J. Ashenden, Gregory D. Peterson, and Darrell A. Teegarden. *The System Designer's Guide to VHDL-AMS: Analog, Mixed-Signal, and Mixed-Technology Modeling*. Morgan Kaufmann Publishers, USA, 2002.

[11] Donald C. Augustin, Mark S. Fineberg, Bruce B. Johnson, Robert N. Linebarger, F. John Sansom, and Jon C. Strauss. The SCi Continuous System Simulation Language (CSSL). *SIMULATION*, 9:281–303, 1967.

[12] Henk Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, revised edition, 1984.

[13] Paul Inigo Barton. *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD thesis, Department of Chemical Engineering, Imperial Collage of Science, Technology and Medicine, London, UK, 1992.

[14] Albert Benveniste and Gérard Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[15] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 174–184, New York, USA, 1998. ACM Press.

[16] A.P.J. Breunese and J.F. Broenink. Modeling Mechatronic Systems Using the SIDOPS+ Language. In *Proceedings of ICBGM'97, 3rd International Conference on Bond Graph Modeling and Simulation*, volume 29 of *Simulation Series*, pages 301–306, 1997.

[17] David Broman. Flow Lambda Calculus for Declarative Physical Connection Semantics. Technical Reports in Computer and Information Science No. 1, LIU Electronic Press, 2007.

[18] David Broman. Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments. Licentiate thesis. Thesis No 1337. Department of Computer and Information Science, Linköping University, December 2007.

[19] David Broman. Growing an Equation-Based Object-Oriented Modeling Language. In *Proceedings of MATHMOD 09 Vienna*, pages 1316–1324, Vienna, Austria, 2009.

[20] David Broman. Should Software Engineering Projects be the Backbone or the Tail of Computing Curricula? In *Proceedings of the 23th IEEE Conference on Software Engineering Education and Training*, pages 153–156, Pittsburgh, USA, 2010.

[21] David Broman, Peter Aronsson, and Peter Fritzson. Design Considerations for Dimensional Inference and Unit Consistency Checking in Modelica. In *Proceedings of the 6th International Modelica Conference*, pages 3–12, Bielefeld, Germany, 2008.

[22] David Broman and Peter Fritzson. Type Safety of Equation-Based Object-Oriented Modeling Languages. PLDI ’06: Poster session at the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Canada, 2006.

[23] David Broman and Peter Fritzson. Ideas for Security Assurance in Security Critical Software using Modelica. In *Proceedings of the Conference on Modeling and Simulation for Public Safety*, pages 45–54, Linköping, Sweden, 2005.

[24] David Broman and Peter Fritzson. Abstract Syntax Can Make the Definition of Modelica Less Abstract. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 111–126, Berlin, Germany, 2007. LIU Electronic Press.

[25] David Broman and Peter Fritzson. Higher-Order Acausal Models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 59–69, Paphos, Cyprus, 2008. LIU Electronic Press.

[26] David Broman and Peter Fritzson. Higher-Order Acausal Models. *Simulation News Europe*, 19(1):5–16, 2009.

[27] David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, pages 303–315, Vienna, Austria, 2006.

[28] David Broman, Kaj Nyström, and Peter Fritzson. Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta. In *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE’06)*, pages 151–160, Portland, Oregon, USA, 2006. ACM Press.

[29] Peter Bunus and Peter Fritzson. Automated Static Analysis of Equation-Based Components. *SIMULATION*, 80(7–8):321–245, 2004.

[30] Peter Bunus and Karin Lunde. Supporting Model-Based Diagnostics with Equation-Based Object Oriented Languages. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 121–130, Paphos, Cyprus, 2008. LIU Electronic Press.

[31] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of Second International Conference on Generative Programming and Component Engineering (GPCE’03)*, volume 2830 of *LNCS*, pages 57–76. Springer-Verlag, 2003.

[32] Luca Cardelli. Type Systems. In *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, second edition, 2004.

[33] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[34] Francesco Casella, Filippo Donida, and Marco Lovera. Beyond Simulation: Computer Aided Control System Design Using Equation-Based Object Oriented Modelling for the Next Decade. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 35–45, Paphos, Cyprus, 2008. LIU Electronic Press.

[35] François E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, USA, 1991.

[36] François E. Cellier. Object-Oriented Modeling of Physical Systems: Promises and Expectations. In *Proc. Symposium on Modelling, Analysis, and Simulation, CESA'96, IMACS MultiConference on Computational Engineering in Systems Applications*, pages 1126–1127, Lille, France, 1996.

[37] François E. Cellier and Ernesto Kofman. *Continuous System Simulation*. Springer-Verlag, New York, USA, 2006.

[38] François E. Cellier and Àngela Nebot. The Modelica Bond-Graph Library. In *Proceedings of the 4th International Modelica Conference*, pages 57–65, Hamburg, Germany, 2005.

[39] James Cheney and Hinze Ralf. First-Class Phantom Types. CISTR TR2003-1901, Cornell University, 2003.

[40] Ernst Christen and Kenneth Bakalar. VHDL-AMS - A Hardware Description Language for Analog and Mixed-Signal Applications. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 46(10):1263–1272, 1999.

[41] Koen Claessen. *Embedded Languages for Describing and Verifying Hardware*. PhD thesis, Department of Computer Science, Chalmers University of Technology and Göteborg University, Sweden, 2001.

[42] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Communications of the ACM*, 32(1):9–23, 1989.

[43] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[44] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[45] Dassault Systems. Multi-Engineering Modeling and Simulation - Dymola - CATIA - Dassault Systemes. `http://www.dymola.com` [Last accessed: July 14, 2010].

[46] Nikolas G. de Bruijn. Lambda Calculus Notations with Nameless Dummies, a Tool For Automatic Formula Manipulation, with Application to the Church-Rosser Theorem. *"Indagationes Mathematicae"*, 34(5):381–392, 1972.

[47] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems.* PhD thesis, Department of Automatic Control, Lund University, Sweden, May 1978.

[48] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica - A Language for Physical System Modeling, Visualization and Interaction. In *Proceedings of the IEEE International Symposium on Computer Aided Control System Design*, 1999.

[49] Hilding Elmqvist and Martin Otter. Methods for Tearing Systems of Equations in Object-Oriented Modelling. In *Proceedings ESM'94 European Simulation Multiconference*, pages 326–332, 1994.

[50] Thilo Ernst, Stephan Jähnichen, and Matthias Klose. The Architecture of the Smile/M Simulation Environment. In *Proceedings 15th IMACS World Congress on Scientific Computation, Modelling and Applied Mathematics*, volume 6 of *Modelling and Applied Mathematics*, pages 653–658, 1997.

[51] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Wiley-IEEE Press, New York, USA, 2004.

[52] Peter Fritzson, Peter Aronsson, Håkan Lundvall, Kaj Nyström, Adrian Pop, Levon Saldamli, and David Broman. The OpenModelica Modeling, Simulation, and Development Environment. In *Proceedings of the 46th Conference on Simulation and Modeling (SIMS'05)*, pages 83–90, Trondheim, Norway, 2005.

[53] Peter Fritzson, Peter Aronsson, Adrian Pop, Håkan Lundvall, Kaj Nyström, Levon Saldamli, David Broman, and Anders Sandholm. OpenModelica - A Free Open-Source Environment for System Modeling, Simulation, and Teaching. In *Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design*, Munich, Germany, 2006. See also the OpenModelica Project. www.openmodelica.org [Last accessed: Dec 30, 2008].

[54] Peter Fritzson, David Broman, François Cellier, and Christoph Nytsch-Geusen. Equation-Based Object-Oriented Languages and Tools. Report on the Workshop EOOLT 2007 at ECOOP 2007. In *Object-Oriented Technology. ECOOP 2007 Workshop Reader*, volume 4906 of *LNCS*, pages 27–39. Springer-Verlag, 2008.

[55] Peter Fritzson and Vadim Engelson. Modelica - A Unified Object-Oriented Language for System Modeling and Simulation. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 1445 of *LNCS*. Springer-Verlag, 1998.

[56] Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, pages 519–525, Hamburg, Germany, 2005.

[57] Peter Fritzson, Adrian Pop, David Broman, and Peter Aronsson. Formal Semantics Based Translator Generation and Tool Development in Practice. In *Proceedings of ASWEC 2009 Australian Software Engineering Conference*, pages 256–266. IEEE Computer Society, 2009.

[58] Peter Fritzson, Lars Viklund, Johan Herber, and Dag Fritzson. High-level mathematical modeling and programming. *IEEE Software*, 12(4):77–87, 1995.

[59] Georgina Fábián. *A Language and Simulator for Hybrid Systems.* PhD thesis, Institute for Programming research and Algorithmics, Technische Universiteit Eindhoven, Netherlands, Netherlands, 1999.

[60] Walter Gellert. *The VNR Concise Encyclopedia of Mathematics.* Van Nostrand Reinhold, 1977.

[61] George Giorgidze and Henrik Nilsson. Embedding a functional hybrid modelling language in Haskell. In *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages*, 2008.

[62] George Giorgidze and Henrik Nilsson. Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In *Proceedings of the 7th International Modelica Conference*, pages 208–218, Como, Italy, September 2009. LIU Electronic Press.

[63] George Giorgidze and Henrik Nilsson. Mixed-level Embedding and JIT Compilation for an Iteratively Staged DSL. In *Proceedings of the 19th Workshop on Functional and (Constraint) Logic Programming (WFLP'10)*, pages 19–34, Madrid, Spain, January 2010.

[64] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[65] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 3rd Edition.* Prentice Hall, 2005.

[66] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 448–472. Springer-Verlag, 1991.

[67] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction.* Cambridge University Press, 2nd edition, 2008.

[68] 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. *ACM Transactions on Mathematical Software*, 31(3):363–396, 2005.

[69] Graham Hutton. *Programming in Haskell.* Cambridge University Press, 2007.

[70] Iain S. Duff. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, 1981.

[71] Iain S. Duff and John K. Reid. An Implementation of Tarjan's Algorithm for the Block Triangularization of a Matrix. *ACM Transactions on Mathematical Software*, 4(2):137–147, 1978.

[72] IEEE 1706.1 Working Group. *IEEE Std 1076.1-1999, IEEE Standard VHDL Analog and Mixed-Signal Extensions*. IEEE Press, New York, USA, 1999.

[73] INRIA. Scicos Homepage. `http://www-rocq.inria.fr/scicos/` [Last accessed: July 10, 2010].

[74] INRIA. The Caml language: Home. `http://caml.inria.fr/` [Last accessed: July 13, 2010].

[75] ISO/IEC. *ISO/ IEC 14882 : Programming language C++*. ANSI, New York, USA, 1998.

[76] ITI. SimulationX. `http://www.iti.de/` [Last accessed: July 10, 2010].

[77] Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer-Verlag, 2009.

[78] JModelica.org. `http://www.jmodelica.org` [Last accessed: July 10, 2010].

[79] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, November 2007.

[80] Gilles Kah. An experiment in partial evaluation: The generation of a compiler generator. In *Rewriting Techniques and Applications*, volume 202 of *LNCS*, pages 124–140. Springer-Verlag, 1985.

[81] Gilles Kahn. Natural semantics. In *4th Annupal Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, volume 247 of *LNCS*, pages 22–39, Passau, Germany, 1987. Springer-Verlag.

[82] Mattihias Kloas, Viktor Friesen, and Martin Simons. SMILE - A Simulation Environment for Energy System. In *Proceedings of the 5th International IMACS-Symposium on Systems Analysis and Simulation (SAS'95)*, pages 503–506. Gordon and Breach Publishers, 1995.

[83] Peter Kunkel and Volker Mehrmann. *Differential-Algebraic Equations Analysis and Numerical Solution*. European Mathematical Society, 2006.

[84] David Kågedal. A Natural Semantics specification for the equation-based modeling languge Modelica. Master's thesis, Linköping University, 1998.

[85] David Kågedal and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proceedings of the Summer Computer Simulation Conference*, 1998.

[86] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Press, 2004.

[87] Edward A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009.

[88] Lennart Ljung. *System Identification: Theory for the User*. Prentice Hall, second edition, 1999.

[89] LMS. LMS International - 1D and 3D simulation software, testing systems and engineering services. `http://www.lmsintl.com/` [Last accessed: July 10, 2010].

[90] Maplesoft. Math Software for Engineers, Educators & Students - Maplesoft. `http://www.maplesoft.com/` [Last accessed: July 10, 2010].

[91] MathCore. MathModelica System Designer: Model based design of multi-engineering systems. `http://www.mathcore.com/products/mathmodelica/` [Last accessed: June 10, 2010].

[92] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/` [Last accessed: November 8, 2007].

[93] Sven Erik Mattsson, Hilding Elmqvist, Martin Otter, and Hans Olsson. Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0. In *Proceedings of the 2nd International Modelica Conference*, pages 9–15, Oberpfaffenhofen, Germany, 2003.

[94] Sven Erik Mattsson and Gustaf Söderlind. Index reduction in differential-algebraic equations using dummy derivatives. 14(3):677–692, 1993.

[95] Jakob Mauss. Modelica Instance Creation. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.

[96] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1, 2003. Available from: `http://www.omg.org`.

[97] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[98] Robin Milner, Mads Tofte, Robert Harper, and David MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[99] John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[100] Modelica Association. `http://www.modelica.org`. [Last accessed: July 10, 2010].

[101] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Version 1*, September 1997. Available from: http://www.modelica.org.

[102] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 2.2*, February 2005. Available from: http://www.modelica.org.

[103] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.0*, 2007. Available from: http://www.modelica.org.

[104] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.2*, 2010. Available from: http://www.modelica.org.

[105] Modelica Association. Modelica Libraries - Modelica Portal, 2010. Available from: http://www.modelica.org/libraries. [Last accessed: July 9, 2010].

[106] Masoud Najafi and Ramine Nikoukhah. Modeling and simulation of differential equations in Scicos. In *Proceedings of the Fifth International Modelica Conference*, pages 177–185, Vienna, Austria, 2006.

[107] IEEE Standards Information Network. *IEEE 100 The Authoritative Dictionary of IEEE Standards Terms.* IEEE Press, New York, USA, 2000.

[108] Henrik Nilsson. Type-Based Structural Analysis for Modular Systems of Equations. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 71–81, Paphos, Cyprus, 2008. LIU Electronic Press.

[109] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *LNCS*, pages 376–390, New Orleans, Lousiana, USA, January 2003. Springer-Verlag.

[110] Henrik Nilsson, John Peterson, and Paul Hudak. Functional Hybrid Modeling from an Object-Oriented Perspective. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 71–87, Berlin, Germany, 2007. LIU Electronic Press.

[111] Kristoffer Norling, David Broman, Peter Fritzson, Alexander Siemers, and Dag Fritzson. Secure Distributed Co-Simulation over Wide Area Networks. In *Proceedings of the 48th Conference on Simulation and Modelling (SIMS 2007)*, pages 14–23, Göteborg (Särö), Sweden, 2007. LIU Electronic Press.

[112] Christoph Nytsch-Geusen et. al. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.

[113] Object Management Group. *Unified Modeling Language: Infrastructure version 2.1.1*, February 2007. Available from: `http://www.omg.org`.

[114] Object Management Group. *Unified Modeling Language: Superstructure version 2.1.1*, February 2007. Available from: `http://www.omg.org`.

[115] M. Oh and Costas C. Pantelides. A modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems. *Computers and Chemical Engineering*, 20(6–7).

[116] Hans Olsson, Martin Otter, Sven Erik Mattsson, and Hilding Elmqvist. Balanced Models in Modelica 3.0 for Increased Model Quality. In *Proceedings of the 6th International Modelica Conference*, pages 21–33, Bielefeld, Germany, 2008.

[117] OpenModelica Project. `http://www.openmodelica.org` [Last accessed: July 10, 2010].

[118] Constantinos C. Pantelides. The Consistent Initialization of Differential-Algebraic Systems. *SIAM Journal on Scientific and Statistical Computing*, 9(2):213–231, 1988.

[119] Terence Parr. ANTLR Parser Generator. `http://www.antlr.org/` [Last accessed: November 8, 2007].

[120] Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 1995.

[121] Linda R. Petzold. A Description of DASSL: A Differential/Algebraic System Solver. In *IMACS Trans. on Scientific Comp., 10th IMACS World Congress on Systems Simulation and Scientific Comp.*, Montreal, Canada, 1982.

[122] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[123] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming*, pages 50–61, New York, USA, 2006. ACM Press.

[124] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[125] Rinus Plasmeijer and Marko van Eekelen. Concurrent CLEAN Language Report - Version 2.1, 2002. Available from: `http://clean.cs.ru.nl/`. [Last accessed: July 13, 2010].

[126] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Department of Computer Science, University of Aarhus, 1981.

[127] Adrian Pop. *Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 2008.

[128] Per Sahlin and E.F. Sowell. A Neutral Format for Building Simulation Models. In *In Proceedings of the Conference on Building Simulation, IBPSA*, Vancouver, Canada, 1989.

[129] Tim Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Workshop on Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *LNCS*, pages 2–44. Springer-Verlag, 2001.

[130] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16, New York, USA, 2002. ACM Press.

[131] Tim Sheard and Emir Pasalic. Meta-programming With Built-in Type Equality. *Electronic Notes in Theoretical Computer Science*, 199:49–65, 2008.

[132] Jeremy Siek and Walid Taha. Gradual typing for functional languages. In *In: Scheme and Functional Programming Workshop*, 2006.

[133] Jeremy Siek and Walid Taha. Gradual Typing for Objects. In *Proceedings of the 21st European conference on ECOOP 2007: Object-Oriented Programming*, volume 4609 of *LNCS*, pages 2–27. Springer-Verlag, 2007.

[134] Simon Peyton Jones. *Haskell 98 Language and Libraries – The Revised Report.* Cambridge University Press, 2003.

[135] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, Portland, Oregon, United States, 1986. ACM Press.

[136] Guy L. Steele. Growing a Language. Videotape (54 minutes) of a talk at the *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, University Video Communications, 1998.

[137] Guy L. Steele. *Common LISP. The Language.* Digital Press, 2nd edition, 1990.

[138] Guy L. Steele. Growing a Language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999.

[139] Bjarne Stroustrup. A history of C++ 1979–1991. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 271–297, New York, USA, 1993. ACM Press.

[140] Walid Taha. *Multi-Stage Programming: Its Theory and Applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, Oregon, USA, November 1999.

[141] Walid Taha. A Gentle Introduction to Multi-stage Programming. In *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 30–50. Springer-Verlag, 2004.

[142] Walid Taha et al. MetaOCaml Homepage. `http://www.metaocaml.org/` [Last accessed: April 8, 2010].

[143] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *PEPM '97: Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 203–217, New York, USA, 1997. ACM Press.

[144] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[145] D.A. van Beek, A.T. Hofkamp, M.A. Reniers, J.E.Rooda, and R.R.H.Schiffelers. Syntax and Formal Semantics of Chi 2.0. SE Report: Nr. 2008-01, Department of Mechanical Engineering, Eindhoven University of Technology, 2008.

[146] D.A. van Beek, K.L. Man, MA. Reniers, J.e. Rooda, and R.R.H Schiffelers. Syntax and consistent equation semantics of hybrid Chi. *The Journal of Logic and Algebraic Programming*, 68:129–210, 2006.

[147] Arie van Deursen and Paul Klint. Little Languages: Little Maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[148] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, 2000.

[149] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[150] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 242–252, New York, USA, 2000. ACM Press.

[151] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[152] Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 224–235, New York, USA, 2003. ACM Press.

[153] Yun Zhu, Edwin Westbrook, Jun Inoue, Alexandre Chapoutot, Cherif Salama, Marisa Peralta, Travis Martin, Walid Taha, Marcia O'Malley, Robert Cartwright, Aaron Ames, and Raktim Bhattacharya. Mathematical equations as executable models of mechanical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 1–11, New York, USA, 2010. ACM Press.

[154] Dirk Zimmer. Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems. In *Proceedings of the 6th International Modelica Conference*, pages 47–56, Bielefeld, Germany, 2008.

[155] Dirk Zimmer. *Equation-Based Modeling of Variable-Structure Systems.* PhD thesis, Swiss Federal Institute of Technology, Zürich, Switzerland, 2010.

Index

A
Abstract data type, 104, 225
Abstract syntax, 221
Abstract syntax tree, 29
Acausality, 9
Acumen, 205
Acusal
    model, 111
ADT
    Array, 225
    Map, 226
    Set, 226
Any model type, 133, 139
Array, 225
AST, 29

B
Balanced models, 206
Binding-time analysis, 128
Bot type, 151

C
Call-by-name, 97
Call-by-need, 97
Cast insertion, 144
Class, 26
Class type, 69
Coercion, 64
Component, 9
Conclusions, 211
Connection semantics, 11, 161
Connector, 27, 162
    inside, 86
    outside, 86
Conservation of energy, 11
Constrained
    over, 76
    under, 76
Constraint delta effect, 81
Contributions, 16
    origin, 20
    overview, 16
CSSL, 7

D
DAE, 5
    DAESolver, 228
    hybrid, 29
DAESolver, 228
Debugging, 90
Depth-first search, 86
Diagnosis, 9
Differential-algebraic equation, 5
Discrete event-based, 7
Domain expert, 8
Domain-specific langauge, 8

DSL, 8
  domain, 8
  domain expert, 8
Dynamic dispatch, 66

E

Eager evaluation, 97
Elaboration, 11, 159
ElectricalBranch, 113
EOO, 6
  origin, 7
EOO Model, 9
Equality, 152
Equation
  modification, 28
Equation-based object-oriented, 6
Error, 12
  detect, 14
Error detection, 90
Errors
  untrapped, 56
Evaluation, 191
  Eager, 97
Executable specification, 173
Experiment, 2
Export DAE, 186
Expressiveness, 14, 193
Extended backus-naur form, 69
Extensibility, 14, 193
Extensional metaprogramming, 214

F

Failure, 12
Fault, 12
Featherweight Modelica, 77
FHM, 204
File includer, 184
Functional programming, 96
Future work, 214

G

GADT, 208
General-purpose language, 8
gPROMS, 1, 7, 203
Gradual typing, 207
Graphical user interface, 30
GUI, 30

H

Hardware description language, 1, 7
HDL, 1, 7
Hierarchy naming, 177
Higher-order acausal model, 115
Higher-order acausal models, 109
Higher-order function, 97, 99
HOAM, 115
Hybrid Chi, 203
Hybrid DAE, 29
Hydra, 204

I

Implementation, 183
Information hiding, 64
Inheritane
  private, 60
Initial conditions, 3
Initial value problem, 3
Inside connector, 86
Isolate error, 14

L

Lemma
  Canonical forms, 149
  inversion, 149
  progress, 149
  substitution, 150
Little languages, 8

M

Map, 226
Match, 102
Mathematical model, 2
MetaOcaml, 209
MKL, 16
  core, 137
  dynamic semantics, 146
  lexical structure, 217
  reserved keywords, 218
  Simulation, 187
  type safety, 148
  type system, 139
Modeification equation, 28
Model, 2
  destructing, 147
  recursive, 117

Model application, 147
Model driven architecture, 2
Model lifting, 143
Model lifting relation, 141
Model type, 126
    any, 139
    specific, 139
Model value, 147
Modelica, 25, 26, 201, 206
    compilation process, 29
    connect, 27
    featherweight, 77
    modeling, 26
    MSL, 25
    process, 29
    subtyping, 79
    type equivalence, 78
    types, 55, 206
Modelica standard library, 25
Modeling, 6
Modeling kernel language, 16
Module, 105
MSL, 25
Multi-domain modeling, 8

N

Natural Semantics, 205
Natural semantics, 34
Node, 112

O

Object, 8
Object type, 69
Object-oriented programming, 8
ODE, 2
Omola, 9
Optimization, 14
Outside connector, 86
Over-constrained, 76
Overloading, 64

P

Pantelides, 12, 30
Parametric polymorphism, 103
Partial application, 98
Partial differential equations, 9
Pattern matching, 102
Pattern variable, 102
Performance aspects, 195
Polymorphism
    parametric, 65
Port, 9
Preservation, 150
Probing, 177
Problem area, 12
Pseudo type, 132, 222
Published papers, 18
Pure language, 96

R

Recursion, 99
Recursive model, 117
Redeclare, 28
Related work, 201
Research method, 21
Research questions, 14
Restriction operator, 140

S

Safe substitution, 80
Safety aspects, 12
Scope, 16
Separate compilation, 76
Set, 226
Side effect, 96
Simulation, 5
    reasons, 6
Simulink, 7
Sol, 204
Specific model type, 139
State-space form, 2
Statically typed, 97
Strict evaluation, 97
Strongly typed, 57
Structural constraint delta, 75, 80
Structural dynamic systems, 214
Structural type system, 78
Subtyping, 58, 79
Symbol table, 184
System, 2
System modeling error, 13

T

TempResistor, 28

Thesis
future work, 214
problem area, 12
related work, 201
research method, 21
research questions, 14
scope, 16
Top-level, 98
Tuples, 101
TwoPin, 28
Type
any model, 133
model, 126
Type consistency, 140
Type equivalence, 78
Type inference, 65
Type safety, 57, 148
Type system, 57
nominal, 61
structural, 61
Types
prefixes, 71
Typing environment, 141

U

Under-constrained, 76
Unified modeling language, 2
Unit type, 101
Unknown, 126, 127
Unknowns, 147
Uses of models, 186

V

Verification, 189
VHDL-AMS, 1, 7, 202

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.

No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.

No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.

No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.

No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.

No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.

No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.

Linköping Studies in Arts and Sciences

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystem-arkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.