Linköping Studies in Science and Technology
Thesis No. 1337

Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments

by

David Broman

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for the degree of Licentiate of Engineering.

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

Linköping 2007

Linköping Studies in Science and Technology
Thesis No. 1337

Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments

by

David Broman

Submitted to Linköping Institute of Technology at Linköping University in partial fulfilment of the requirements for the degree of Licentiate of Engineering.

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

Linköping 2007

An electronic version of the first part of the thesis is available at:
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-10134
Printed by LiU-Tryck, Linköping 2007

Safety, Security, and Semantic Aspects of Equation-Based Object-Oriented Languages and Environments

by

David Broman

December 7, 2007
ISBN 978-91-85895-24-3
Linköping Studies in Science and Technology
Thesis No. 1337
ISSN 0280-7971
LIU-TEK-LIC-2007:46

ABSTRACT

During the last two decades, the interest for computer aided modeling and simulation of complex physical systems has witnessed a significant growth. The recent possibility to create acausal models, using components from different domains (e.g., electrical, mechanical, and hydraulic) enables new opportunities. Modelica is one of the most prominent equation-based object-oriented (EOO) languages that support such capabilities, including the ability to simulate both continuous- and discrete-time models, as well as mixed hybrid models. However, there are still many remaining challenges when it comes to language safety and simulation security. The problem area concerns detecting modeling errors at an early stage, so that faults can be isolated and resolved. Furthermore, to give guarantees for the absence of faults in models, the need for precise language specifications is vital, both regarding type systems and dynamic semantics.

This thesis includes five papers related to these topics. The first paper describes the informal concept of types in the Modelica language, and proposes a new concrete syntax for more precise type definitions. The second paper provides a new approach for detecting over- and under-constrained systems of equations in EOO languages, based on a concept called structural constraint delta. That approach makes use of type checking and a type inference algorithm. The third paper outlines a strategy for using abstract syntax as a middle-way between a formal and informal language specification. The fourth paper suggests and evaluates an approach for secure distributed co-simulation over wide area networks. The final paper outlines a new formal operational semantics for describing physical connections, which is based on the untyped lambda calculus. A kernel language is defined, in which real physical models are constructed and simulated.

This research work was funded by CUGS (the National Graduate School in Computer Science, Sweden), by SSF under the VISIMOD II project, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

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

To my lovely wife Åsa and wonderful daughter Tove

Acknowledgments

First of all, 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 helped me in many situations during the two and a half years of this thesis work; especially by telling me what to prioritize or give up when I get too enthusiastic and make unrealistic plans.

I would also like to thank my co-authors to the articles presented in the thesis. Kaj Nyström for the good and intensive cooperation on the structural constraint delta approach, Sebastien Furic for interesting discussions about Types in Modelica, and Kristoffer Norling for the hard work with all experiments on secure distributed simulations. I'm also grateful for the cooperation with Dag Fritzson and Alexander Siemers from SKF Engineering Research Centre.

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.

Parts of this thesis have been proofread by Johan Åkesson, Peter Bunus, John Wilander, Thomas Schön, Björn Lisper, and Hans Olsson. I'm grateful for all comments and suggestions which substantially have improved this work.

I would also like to thank the opponent at my coming thesis defence, Henrik Nilsson. Not for the opposition; I don't yet know how critical you are going to be, but for the inspiring discussions we had in Berlin and Nottingham, which have influenced the ideas behind Paper E in the thesis.

During this time, 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, Kristian Sandahl for helping me out in critical moments with common teaching efforts, and finally my grandmother Ingrid Broman who has been an almost too friendly host in Linköping.

I would especially like to thank Thomas Schön, my old friend and research "mentor", for giving me inspiration and advices during the work, but in particular for being a really good friend.

Thanks to all my other friends and family for all the support you have given me. Especially, I would like to express my deepest gratitude to my lovely wife, who has encouraged me during this time, even if I know that you sometimes would have preferred that my large interest were something more ordinary and easy to grasp than computer science.

I would also want to thank my wonderful newborn daughter Tove, who continuously helps me when I'm stuck in my work, by reminding me that there are other more important things in life than modeling languages.

Finally, I would like to thank the people in Ethiopia, who first discovered how to make coffee. Without this vital beverage, this thesis would never have been finished.

Linköping, November 8, 2007

David Broman

Contents

Part I

Introduction

1 Background

COMPUTER aided modeling and simulation of complex physical systems, using components from several domains, such as electrical, mechanical, and hydraulic, have in recent years witnessed a significant growth of interest. General-purpose simulation tools, e.g., Simulink [53], using block diagrams and causal connections, have dominated the area for many years. However, during the past two decades a new generation of languages has evolved. This language category is based on object-oriented concepts and acausal modeling using equations. This enables better reuse of components resulting in considerably reduced modeling effort [26]. One such language is *Modelica* [61], which is an attempt to unify concepts and notation from several research projects and industrial initiatives. Other examples of languages with similar modeling and simulation capabilities are gPROMS [6, 68] and VHDL-AMS [18].

This thesis concerns different aspects of safety, security, and semantics of such languages and their development and simulation environments. The thesis is divided into an introductory part, where the background and principles of modeling and simulation for these kind of languages are described. It is followed by the problem area description, research questions, research method, contributions, related work, and conclusions. The second part of the thesis contains the main contributing material presented as four published peer reviewed conference papers and one technical report[1].

1.1 Modeling and Simulation

Modeling and the concept of models are today very active areas 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

[1]Due to copyright issues, the electronic version of this thesis published at Linköping Electronic Press does not contain these articles. Instead, links to the published papers are supplied.

and misunderstanding regarding the subject. During the last decades, modeling of software has become very popular; especially in industry. One of the main driving forces is the Model Driven Architecture (MDA) [55] initiative and the popular graphical modeling framework of the Unified Modeling Language (UML) [66, 67].

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 as models. To be able to reason about the process of modeling and simulation, some definitions of terms have to be clarified. The following definitions are stated in [16], but have first been coined by different authors.

> *"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. A simulation can be seen as a special experiment:

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

Hence, when we are talking about modeling and simulation, we mean modeling of a physical system (e.g., a car, an engine, or an electric circuit) resulting in an artifact: the model. Then, by applying experiments, i.e., perform the simulation on the model, we can answer certain questions about the physical system that the model describes.

There are many reasons why simulations are beneficial. For example:

- It is *too expensive* to perform experiments on real systems.
- It is *too dangerous.*
- The system *may not exist*, i.e., the model is a prototype that is evaluated and tested during development.
- Some *variables are not accessible* in the real system, but can be observed in a simulation.
- It is *easy to use* and modify models, to change parameters and perform new experiments (simulations).

However, as pointed out in both [16] and [30], 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 for which simulations that are suitable to apply on a model, so that the results reach the right level of accuracy.

1.2 Equation-Based Object-Oriented Languages

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 [20], founded the fundamental concepts of object-orientation languages. However, the fundamental principles for *equation-based object-oriented* modeling and simulation have been around for about 30 years, starting with the pioneering work explored in two separate PhD theses[17]: by Hilding Elmqvist[25] and Tom Runge.

Later in the 1990's and forward, a number of languages for modeling and simulation of complex physical systems have emerged. For example, Omola [4], Modelica [61], gPROMS [6, 68], χ (Chi) [28, 88], and VHDL-AMS [18].

Several of these languages support language constructs which are commonly regarded as parts of object-oriented languages. For example the class concept in Omola and Modelica, and inheritance in Omola, Modelica, and gPROMS. χ and VHDL-AMS are not object-oriented languages, but they have much in common when it comes to modeling and simulation of dynamic physical systems.

All these languages are often regarded as modeling languages, which can be classified in several ways. For example, a widely used categorization is how states change over time. *Continuous-time (CT)* languages model systems with infinite number of states (finite number of state variables) where state variables change continuously over time, *discrete-time (DT)* languages change state at discrete points in time, and the combination CT/DT handles both continuous-time and discrete-time models. This latter category is often referred to as *hybrid languages*, and the above mentioned languages are all such languages. A more detailed categorization due to progress of states can be found in [89].

We think that the modeling part has another dimension for classification, especially regarding the object-oriented view. The real physical world is naturally described by objects, where each object's state progresses over time. Hence, the object-oriented view is a natural choice when designing a modeling language for physical systems. However, the needed language construct for such an object-oriented modeling language differs drastically from general-purpose object-oriented languages such as C++ and Java. In these main stream languages, concepts such as classes, objects, dynamic dispatch, methods, message passing, inheritance, polymorphism, encapsulation, etc. are regarded as central. There are many more concepts related to object-oriented languages and as shown in [5] there is no clear consensus what actually defines the core concepts of OO languages.

However, several of these concepts are less important for modeling and simulation. Conversely, other concepts that do not exist in general-purpose OO-languages are vital for physical modeling.

In this thesis we refer to this kind of languages as *Equation-Based Object-Oriented (EOO)* languages[2]. To conclude, we define the concept of EOO language as follows:

Definition 1.2.1 (EOO language). Equation-Based Object-Oriented (EOO) languages provide the following fundamental concepts:

- **Equations** - Equations capable of modeling continuous-time systems.
- **Models (Classes)** – A blueprint for creating instances.
- **Objects** – Model instances describing a system or sub-system. Composes equations and other objects.
- **Inheritance** - Inheritance of behavior between models and/or objects.
- **Polymorphism** - Subtyping (inclusion) and/or parametric polymorphism.
- **Acausal connections** - Connections between objects, describing both potential and flow connections.

The first concept, *equations* for describing continuous-time systems are, in all the mentioned languages applied by using differential algebraic equations (DAEs). The general representation of a DAE can be formulated as

$$f\big(t, \dot{x}(t), x(t), y(t), u(t), p\big) = 0 \quad \text{where}$$

t	time
$\dot{x}(t)$	vector of differentiated state variables
$x(t)$	vector of state variables
$y(t)$	vector of algebraic variables
$u(t)$	vector of input variables
p	vector of parameters and constants

Hence, according to this definition, we have chosen to have continuous-time modeling as a mandatory feature for an EOO language, but letting discrete event capabilities being optional. The main rationale for this decision is that many physical systems can be described without discrete events, while the opposite is not true.

The second and third concepts *models* and *objects*, concern the composition of equations and other objects in a hierarchical fashion. Object-oriented languages can be classified into *class-based languages* and *object-based languages* [1]. In the former, classes are used as blueprints for generating objects. In the latter, the class concept is absent and instead there are specific constructs for creating objects. In the definition of EOO, we are primarily using the term *model* in favour of class, since it gives a better analogy to models of physical systems. Hence, the term model-based languages are used instead of class-based. If an equation-based language lacks the concept of models, we refer to it

[2]The term was first publicly used at a poster session at the conference on programing language design and implementation (PLDI) 2006 [9].

as an *equation-based object-based* language. Models can be represented using functional abstraction and object creation performed by function application. The latter approach is actually the case that will be demonstrated in Paper E of this thesis.

The fourth concept, *inheritance*, means that behavior, primarily described using differential algebraic equations, can be reused from existing models or objects. If the language is model-based, new models can be created statically by extending (sub-classing) existing models. This concept is used in e.g., Modelica. On the other hand, if the language is object-based, new objects can be produced by *cloning* earlier created objects. This approach is used in so-called *prototype-based* languages. A form of inheritance in such languages can be achieved by *embedding* objects inside each other, or by *delegating* responsibility [1][3]. Note that in this definition of EOO, both model-based and object-based principles of inheritance are acceptable.

Polymorphism is a very important feature to enable reuse and expressiveness in a language. In traditional OO interpretation, polymorphism is often implicitly meaning *subtyping polymorphism*. However, with the current definition, a language supporting only *parametric polymorphism* and not subtyping polymorphism, would still be treated as a valid EOO language. For a detailed discussion about different forms of polymorphism, see Paper A [11] in this thesis.

Finally, the last concept of acausal (or non-causal) connections concerns the possibility to connect models using physically correct or non-physical connectors. These connections involve *potential* (sometimes referred to as *across*) variables, which for example is the potential voltage in the electrical domain and an angle in the rotational mechanical domain. The other kind of variables needed in a physical connection are *flow* (also called *through*) variables. In the electrical domain it corresponds to Kirchhoff's current law, i.e., that the current should sum to zero in a node. In the rotational mechanical domain, a flow variable would model the torque.

Looking back at the definition, concepts one and six are special concepts not available in ordinary general purpose languages. However, concepts two to five all correspond to concepts that can be found in other programming languages. Other language features, such as information hiding, can of course also be valuable, but we do not see these as essential in describing models of physical systems.

Note especially that the behavior in a general purpose OO language is described by method calls or message passing, while the main behavior in EOO languages is described using differential algebraic equations.

1.3 Fundamentals of Modelica

EOO languages and especially Modelica are currently primarily used for modeling and simulation (M&S). Nevertheless, there exist attempts to use them for other applications, such as system identification and optimization [44].

The first part of this section describes the most fundamental concepts and constructs available in many EOO languages when used for M&S. We will primarily use Modelica as the target language for our discussion, since it is an open standard with a growing

[3]In fact, state of the art in design patterns [35] for object-oriented design states that object composition should be favored over class inheritance.

```
model Circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  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;
```

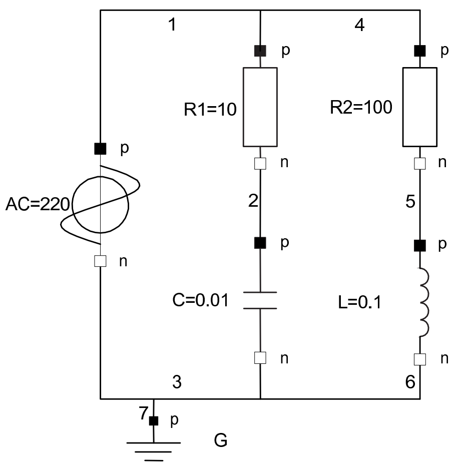


Figure 1.1: Modelica model of an electrical circuit.

and active community. Furthermore, since we will study the Modelica language in depth in Paper A and Paper C, this short introduction aims at giving the reader a fundamental overview of the language.

The second part of this section describes the compilation process, where a model is taken as input and simulation data is the resulting output.

Language Concepts and Constructs

The Modelica language and its modeling environment consist of many fundamental concepts and constructs. In the following listing, we briefly describe the most important ones.

Graphical vs. Textual modeling. Consider the model of a simple electrical circuit given in Figure 1.1. The model can have both a textual representation (left side) and a graphical representation (right side). Tools, such as Dymola [24] and MathModelica System Designer [52] make it possible to modify both these representations concurrently and relatively consistently.

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

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

model Inductor
  extends TwoPin;
  Real L = 0.1;
equation
  L*der(i) = v;
end Inductor;
```

Figure 1.2: *Source code of the `Inductor` model and its base class `TwoPin`.*

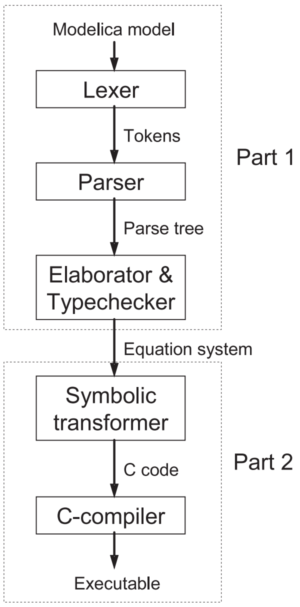


Figure 1.3: *The structure of a Modelica compiler.*

Hierarchical Composition. Instances of classes (in Modelica defined with the keyword `model`) can be hierarchically composed. For example in Figure 1.2, model `Inductor` is defined, while in Figure 1.1 model `Circuit` holds an element named `L`, which is an instance of class `Inductor`.

Continuous-time vs. Discrete-time. If a model only has variables that evolve continuously over time, it is said to be a *continuous-time* model. These models are described using DAEs. Conversely, if a model changes its values only at discrete points in time, it is said to be a *discrete-time* model. Moreover, if a model contains both discrete- and continuous-time variables, it is said to be a *hybrid model.*

Causal vs. Acausal modeling. In a block oriented simulation environment, such as Simulink [53], the interconnected blocks must be stated using a directed data flow with input and outputs. However, this *causal* modeling approach does not reflect the topology of the physical system [26]. Using an *acausal* (sometimes referred to as non-causal) modeling approach, the equations are instead stated in their natural form as differential algebraic equations. With the latter approach, the direction of the data flow is unspecified at the modeling stage.

Connections and Flow variables. Connections between instances are stated by using `connect`-equations; depicted in Figure 1.1. These equations connect *ports* (in Modelica called connectors), and represent several equations. For instance, `connect(L.n, C.n)` represents two equations: `L.n.v = C.n.v` and `L.n.i + C.n.i = 0`. The first equation expresses that the voltage at the connection ends are the same, whereas the second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a node. The latter concept is achieved with the *flow variable* concept, which is part of the Modelica semantics.

Inheritance and Modifications. Equations and elements in one class can be reused when defining another class, using the concept of inheritance. For instance, in Figure 1.2, the `Inductor` inherits behaviour from model `TwoPin`. Moreover, it is also possible to modify declaration equations, such as `Real L=0.1` in model `Inductor`, or even replacing class instances. For example, if a large model of a car is created, it is possible to replace the gearbox without affecting the other parts of the model.

Modelica is a large and complex language, consisting of many constructs, such as inner-outer components, arrays, matrices, expandable connectors etc. For a more comprehensive overview, see [30].

The Compilation Process

To be able to understand the research problem, we will first give a brief overview of the compilation process.

A Modelica compiler can generally be divided into two parts; depicted in Figure 1.3. In the first part, scanning and parsing results in an abstract syntax tree. The Abstract Syntax Tree (AST) is then type-checked and *elaborated* into a *flat system of equations*[4]. This gives us the following definitions:

Definition 1.3.1 (Flat system of equations). A flat system of equations is a set of declared variables of primitive types together with a set of equations referencing these variables.

Definition 1.3.2 (Elaboration). Elaboration is the task of producing a flat system of equations from the AST of a model.

In the second part, different symbolic manipulations and optimizations are performed on the equation system. The symbolic transformation module then generates a program, normally C code. This program is then linked together with a numerical solver, such as DASSL [74], which is used for solving the equation system. Finally, this executable is executed, producing the simulation result.

[4]also called a hybrid DAE, when zero-crossing functions for discrete-events are included.

2 Problem Area

EQUATION-BASED OBJECT-ORIENTED modeling is a rapid way of modeling systems, by reusing well defined components. If the components do not exist, they can be created by using the declarative notation of equations. However, it is not always possible to simulate an EOO model, since the model may be incorrectly specified. Furthermore, even if a simulation result is generated, this does not imply that the result is correct.

We will in the first section outline the overall problems and challenges regarding safety aspects of EOO languages and their environments, followed by a section describing security issues in a distributed simulation environment.

2.1 Safety Aspects in EOO Languages and Environments

By following the terminology defined in the IEEE Standard 100 [63], we define an *error* to be something that is made by human beings. As a 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., commit 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.

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

1. ***Detect* the existence of an error early.** If a simulation fails, it is trivial to detect that an error must exist. However, if a simulation job takes 48 hours to complete, it is not desirable to wait 46 hours before the error is detected. Furthermore, when a simulation produces a result, how do we then know that this result is correct?

2. ***Isolate* the fault implied by the error.** If we have detected that an error must exist, how do we then 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, resulting after elaboration of an equation system 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. ***Guarantee* 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 faults cannot exist in a model, e.g., that a specific type of errors will always be detected?

There are many different sources of errors in an M&S environment. Consider Figure 2.1, 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.

System Modeling Errors. A *system modeling error* can result in that the EOO model contain 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), while other result in simulation failures during simulation (e.g., numerical singularities). Moreover, an engineer can make mistakes while modeling a system, which still gives simulation result, but perhaps incorrect values. One such area where errors easily are introduced is inconsistency with respect to physical units and dimensions. For example, in September 1999, the NASA

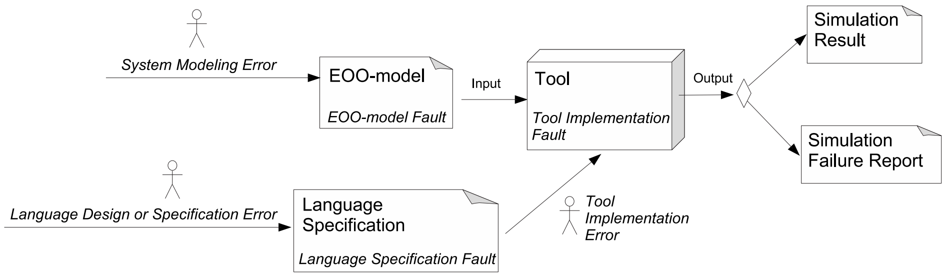

System Modeling Error
EOO-model
EOO-model Fault
Input
Tool
Tool Implementation Fault
Output
Simulation Result
Simulation Failure Report
Language Design or Specification Error
Language Specification
Language Specification Fault
Tool Implementation Error

Figure 2.1: *Relations between possible errors and faults in a M&S-environment.*

Mars Climate Orbiter Mission lost contact with the spacecraft during the Mars orbit maneuver. This failure was eventually traced back to a software flaw when converting between English and metric units [84].

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 [60] using plain text. This fact can lead to *language design errors*, since it is hard to grasp the semantics of the language. 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.

While all different sources of errors may affect the output results from a tool, it is obviously even more challenging to detect and isolate the faults during the tool and language development life-cycles.

2.2 Security Aspects of Modeling and Simulation

Safety aspects of EOO languages and environments concern handling of errors in a sound manner, so that simulations can be produced correctly and are reliable compared to the behavior of the real system being simulated.

Secure modeling and simulation on the other hand, concerns three fundamental concepts of information security:

- *Confidentiality*: protection against unauthorized disclosure of information.
- *Integrity*: protection against unauthorized creation, modification, or deletion of information.
- *Availability*: the assurance that authorized entities have access to correct information when needed.

Within a modeling and simulation environment, there are different types of information that need to be handled in a secure manner. In many companies, the models describe the organizations primary know-how and can therefore be seen as critical business assets. Hence, the model information itself is an important information to be protected.

Larger enterprises are often divided into several departments, modeling different parts of a system. There may exist different confidentiality levels within the organization or between companies. Furthermore, control over how models can be accessed and modified need to be controlled in a need-to-know basis. Since different parts of the organization may be located in different parts of the world, the challenge is how to model and simulate different models together in a distributed environment. The problem concerns both secure handling regarding confidentiality and integrity aspects of the models, as well as availability and performance concerns of the total simulation time.

3 Paper Overview

THIS thesis consists of four peer reviewed published conference papers and one technical report. In the following chapter the overall research questions and problems related to these papers will be outlined. The different research methods used are described and an overview of related work is given. Finally, the main contributions of the work are stated.

3.1 Research Questions

From the problem area in Chapter 2, a number of research questions are formulated below.

3.1.1 Semantics of the Modelica Language

The primary EOO language studied in this thesis is the Modelica language. A common way of detecting and isolating errors statically 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 first question in the study concerns Modelica types.

Research Question 1. What is the actual meaning of types in Modelica and how does it compare to the class concept in the language?

Both the dynamic and static semantics of the Modelica language are informally described using natural language. Since 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 2. How can an informal language specification be restructured to be less ambiguous and still understandable for a general audience?

Research question 1 is primarily covered in Paper A, while question 2 is discussed in Paper C.

3.1.2 Early Detection of Constraint 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 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 equation after elaboration, and then tracing back the faults to the original models [13], leading to the following question:

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

Research question 3 is covered in Paper B.

3.1.3 Formal Operational Semantics of EOO languages

To be able to guarantee the absence of errors statically, in this case without elaborating the model, it is needed to prove properties such as type safety on the language semantics. Hence, a formal definition of the language semantics is needed to prove such propositions. Since the Modelica language is informally described, the fundamental concept of the language needs to be described formally. Hence, the following question concerns the future possibility of proving properties about the language.

Research Question 4. How can the elaboration semantics of an EOO language be formally defined using operational semantics?

Question 4 is handled in the technical report, Paper E.

3.1.4 Secure Simulation

The previous questions are concerned with language safety issues of EOO languages in general and the Modelica language in particular. The final question for this thesis relates to secure simulation.

Research Question 5. How can we perform simulations in a secure manner, using models defined in different tools at different locations over the globe?

The last question 5 is discussed and evaluated in Paper D.

3.2 List of Papers

The research results are presented in the five papers given in Part II of the thesis. The papers are as follows:

Paper A: Types in the Modelica Language

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.

Paper B: Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta

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.

Paper C: Abstract Syntax Can Make the Definition of Modelica Less Abstract

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.

Paper D: Secure Distributed Co-Simulation over Wide Area Networks

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.

Paper E: Flow Lambda Calculus for Declarative Physical Connection Semantics

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

3.3 Research Methods

There are several different paradigms on how to perform research within computer engineering and computer science. The ACM Task Force on the *core of computer science* suggests three different paradigms for conducting research within the discipline of computing [19]:

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 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 consist of stating requirements, defining the specification, designing and implementation of the system, and finally testing the system. The purpose of constructing the system is to solve a given problem.

The 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 in [19], 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 papers.

In Paper A, *Types in the Modelica Language*, the type concept of Modelica is analyzed and interpreted and concrete syntax of types in 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 tested using the parser generator tool ANTLR [70]. In this case, the Modelica specification itself can be seen as the requirements specification. However, due to the fact that the produced artifact is an interpretation of the specification, testing is not applicable.

Paper B *Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta* defines 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. Since Modelica's semantics is not formally defined, it is not possible to conduct any proof for the correctness of the algorithm in relation to the elaboration semantics. Hence, as described in the paper, a test procedure takes place where the correctness of the algorithm is tested using different complex test models, where the model is executed in a commercial Modelica tool, and compared to the type inference algorithm implementation. We should note that this test only checks the correctness of the algorithm, and does not verify that the approach of structural constraint delta actually helps the user to detect the error and isolate the fault.

Paper C *Abstract Syntax Can Make the Definition of Modelica Less Abstract* discusses the problem of finding a middle-way alternative inbetween a totally informal semantics and a formal one. This work is more of a discussion article, where different alternatives are presented and analyzed. Hence, the work does not directly fall into any of the three paradigms, even if the design alternative is probably the closest one. However, since the article describes a suggested approach, no testing of the feasibility of the approach is conducted.

In Paper D *Secure Distributed Co-Simulation over Wide Area Networks* an approach is described (the hypothesis) for performing secure distributed co-simulation. An experiment is conducted and the results discussed and analyzed. Hence, this research is clearly performed within the paradigm of abstraction. The conclusions are drawn using an inductive approach, where experimental data was created in both a simulated environment, and in a real environment.

The final Paper E *Flow Lambda Calculus for Declarative Physical Connection Semantics*, the research method follows a similar approach as in Paper B, where both the theory and the design paradigm are applicable.

A final note should also be made that the level of description of the scientific methods in the papers are adapted to fit the policy for the conference in question.

3.4 Contributions

Each paper states the main contribution of each work. To summarize, the following are the main contributions of this thesis:

- *Paper A:* A description and interpretation of the type concept in Modelica as well as a new definition of the concrete syntax for types of the language.
- *Paper B:* The novel concept of *structural constraint delta*, denoted C_Δ. The approach makes use of static type checking and consists of a type inference algorithm, which determines if a model is under- or over-constrained without elaborating its subcomponents.
- *Paper C:* The described approach of using abstract syntax as a middle-way strategy to define the Modelica language less ambiguously.
- *Paper D:* The discussed and verified approach of secure distributed co-simulation over long distances, which is demonstrated to be both practical and possible in the real world test case.
- *Paper E:* The novel design of the operational semantics for declaratively handling physical flow connections in a correct manner.

3.5 Related Work

Related work for the different areas of research are described in each paper respectively. Hence, this information will not be repeated here. However, certain new related work has been developed since the papers were published, and other earlier related work has come to the authors knowledge.

Structural Constraint Delta and Modelica Specification 3.0

In September 2007, a new version 3.0 [61] of the Modelica specification was released. In this version, the intention was to improve the readability of the language and to simplify it. The readablity of the specification has increased dramatically, by using a new structure and better descriptions. However, the specification is still informally described and the amount of text has increased since the last version.

The largest change in the language is the new constraint that all models in the Modelica language must be balanced to be valid, i.e., that the number of equations and unknowns should be equal. Furthermore, the language is restricted to have balanced connections, i.e., that the number of potential and flow variables must be equal in a connector and connection. These concepts are basically equivalent to the suggested approach given in Paper B about structural constraint delta. Balanced models would in that case require the constraint delta to be zero. The concept of balanced connectors corresponds to that the effect delta is zero. 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 explaining the core concept in a sound manner.
- The Modelica specification requires models to be *always* locally balanced, with the exception of partial classes. The constraint delta concept as explained in the article is more relaxed, and accepts locally over and under-determined models, as long as the global model has constraint delta zero.
- The Modelica specification approach detects the model constraint by elaborating the models. The constraint delta approach uses the model types to annotate the constraint information.

Both of these approaches are justified by examples and tests, but due to the absence of formal semantics it is impossible to prove correctness.

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 [12], the constraint delta approach was shown 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 [64] from 2002, 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.

3.6 Paper Errata

A few typos and errors have been corrected in the papers attached to this thesis, compared to the original published versions.

- In Paper A, Figure 4, the names of the constructors in classes Resistor2 and Inductor have been corrected.
- In Paper A, in Footnote 1, it is clarified that the Modelica language can only be regarded as a safe language, if the tool unconditionally detects all errors and terminates the execution with an error message.
- Paper B has been updated with an error correction of the type inference algorithm. Changes has been made to Algorithm 2 and in the last bullet item on page 69.

4 Concluding Remarks

IN this section, the conclusions of the thesis are summarized and some direction for future research are proposed.

4.1 Conclusions

In this thesis we have discussed and suggested different approaches related to language safety and secure simulation for equation-based object-oriented languages.

Two of the research questions given for the work concern the semantics and specification of the Modelica language. We have seen that it is very hard to formally specify the Modelica language, due to its size and complex semantics. The type concept in relation to the class concept has been discussed in detail, and it has been shown that the current status of the language is to a high degree open for interpretation. Moreover, a strategy for improving the informal specification using abstract syntax was outlined. Since the papers regarding this area were published, a new version 3.0 of the specification has been released. This specification has a clearer description, but the semantics is still described using natural language.

A new approach for detecting over- and under-constrained systems of equations at the model level has been proposed and demonstrated for a subset of the Modelica language. Compared to earlier attempts at static debugging techniques, this approach makes use of a static type system and detects the errors before elaboration to a flat equation system. Furthermore, the new version of the Modelica specification has now also incorporated a similar idea as presented in this thesis, where models are always forced to be balanced with the same number of equations as unknowns.

To enable future guarantees of static properties, such as physical unit checking or constraint errors, a formal operational semantics for the physical connection semantics has been developed within the context of the untyped lambda calculus. We believe that this work can be useful for future languages, wishing to rely on the sound basis of lambda calculus, and still incorporate acausal aspects of EOO languages.

Finally, one of the papers did not concern language safety and semantics, but security aspects of distributed co-simulation. A suggested approach using co-simulation with transmission line modeling (TLM) was given and tested in both a simulated wide area network (WAN) and between sites over the world separated by long distances.

4.2 Future Research

EOO languages can still be seen as a very young area, where most of the research has been conducted from the engineering side, with focus on the back-end numerical and symbolic solver solutions. However, new opportunities and problems appear when state of the art results from computer science and programming language theory is introduced.

The following list shows some very interesting areas of future research.

- *Structural dynamics for acausal modeling languages.* In state of the art EOO languages, e.g., Modelica, the models are elaborated down to an equation-system, which is then solved by a simulation engine. This means that model instances or objects are only created once, before simulation. In contrary, in a structural dynamic system, objects can be created and deleted *during the simulation.* There is currently active research in this area [64, 91].

- *Structural constraint delta with well-constrained models.* The concept of structural constraint delta only requires that the number of equations and variables are equal. However, there are systems where this is true, but the system is structurally singular, meaning that there are no permutations of the incidence matrix that can form a non-zero diagonal. An open question is if this is possible to detect at the type level, without deducing any further information from the content of the term.

- *Type-safty proofs of EOO languages based on the lambda calculus.* Since we have defined a flow-lambda calculus where physical connection semantics is possible, a natural next step is to define a type system for the untyped semantics, that includes the concept of constraint delta. However, to be able to guarantee the absense of errors, a next relevant step would be to prove type safety for the new language.

- *Define model transformation and solution methods within the language itself.* Many so called meta-modeling tasks are today performed with current EOO tools using different forms of scripting languages, i.e., languages that are separate from the modeling language. Furthermore, the post-processing routines such as numerical solvers and symbolic manipulation routines are today implemented in the tools. An interesting alternative worth exploring is to extend the core of the modeling language to be Turing complete, so that these back-end algorithms can be defined as libraries within the same language as the models were defined. In such a way, the flexibility to change and prototype new methods and algorithms can potentially be increased significantly.

- *Detecting and isolating unit errors and faults.* One area for static checking of physical models is unit checking. This area of research is far from a new. Many library-based approaches exist for imperative programming languages, such as a package approach for Ada [38] and a template approach in C++ [87]. In Kennedy's PhD thesis [46], an extension of a core calculus of ML with support for type inference over dimension types is given. Lately, dimension and unit checking has also been addressed in a nominally typed object-oriented language [3]. Besides the work on gPROMS [71, 77], few attempts have been tried to incorporate dimensional and / or unit checking in EOO languages. In addition, even though Modelica today supports syntax for stating units of variables, no sound solution exists that guarantees the absence of unit errors. This kind of guarantee must be performed using mathematical proofs, where the formal semantics of the language we are proving must exist.

- *Other domains than simulation.* There are several more related domains which can be beneficially used in the context of an EOO language. The new PhD thesis [44] by Johan Åkesson explains several alternative application areas, with focus on optimization.

Part II

Papers

Paper A

Types in the Modelica Language

Authors: David Broman, Peter Fritzson, and Sébastien Furic

Edited version of paper originally published in *Proceedings of the Fifth International Modelica Conference*, pages 303-315, Vienna, Austria, 2006.

Types in the Modelica Language

David Broman[⋆], Peter Fritzson[⋆], and Sébastien Furic[†]

[⋆]Department of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden
E-mail: {davbr,petfr}@ida.liu.se

[†]Imagine SA
3, cours Albert Thomas
69416 LYON Cedex 3, France

Abstract

Modelica is an object-oriented language designed for modeling and simulation of complex physical systems. To enable the possibility for an engineer to discover errors in a model, languages and compilers are making use of the concept of types and type checking. This paper gives an overview of the concept of types in the context of the Modelica language. Furthermore, a new concrete syntax for describing Modelica types is given as a starting point to formalize types in Modelica. Finally, it is concluded that the current state of the Modelica language specification is too informal and should in the long term be augmented by a formal definition.

Keywords: type system, types, Modelica, simulation, modeling, type safety.

1 Introduction

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 is 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 and formal definitions of types and type systems. This may result in incompatible compilers and unexpected behavior when using the language.

The purpose of this paper 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 and likewise the main contribution of this work is the definition of a concrete syntax for describing Modelica types. This syntax together with rules of its usage can be seen as a starting point to more formally describe the type concept in the Modelica language. To the best of our knowledge, no work has previously been done to formalize the type concept of the Modelica language.

The paper is structured as follows: Section 2 outlines the concept of types, subtypes, type systems and inheritance, and how these concepts are used in Modelica and other mainstream languages. Section 3 gives an overview of the three main forms of polymorphism, and how these concepts correlate with each other and the Modelica language. The language concepts and definitions introduced in Section 2 and 3 are necessary to understand the rest of the paper. Section 4 introduces the type concept of Modelica more formally, where we give a concrete syntax for expressing Modelica types. Finally, Section 5 state concluding remarks and propose future work in the area.

2 Types, Subtyping and Inheritance

There exist several models of representing types, where the *ideal model* [15] 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 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, are called *ideals*. Figure 1 gives an example of the universe V and two ideals: 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.

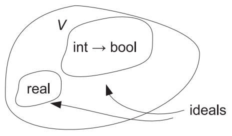


Figure 1: *Schematic illustration of Universe V and two ideals.*

2.1 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 [14].

- *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 run-time 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 *run-time checks,* also called *dynamic checking.* However, note that the distinction between compile-time and run-time 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* [90]. 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"[56]. 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 [90]. Using this technique, informally stated, type safety hold if and only if the following two statements holds:

- *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 corresponds 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 1 lists a number of programming languages and their properties of being type safe [58][14]. The table indicates if the languages are primarily designed to be checked statically at compile-time or dynamically at run-time. 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. Currently, ML [56] and subsets of the Java language [86] [41] has been proven to be safe.

Language	Type Safe	Checking
Standard ML	yes	static
Java	yes	static
Common LISP	yes	dynamic
Modelica	yes	static[1]
Pascal	almost	static
C/C++	no	static
Assembler	no	-

Table 1: *Believed type safety of selected languages.*

2.2 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* [75]. In this case, S is said to be a subtype of T, which is written as

$$S <: T \tag{1}$$

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*[2] 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 3.

2.3 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.

[1]One can argue whether Modelica is statically or dynamically checked, depending on how the terms compile-time and run-time are defined. Furthermore, since no exception handling is currently part of the language, semantics for handling dynamic errors such as array out of bound 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.

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

Figure 2 shows an example[3] where inheritance is used in Modelica. A *model* called `Resistor` extends from 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`. Since `Resistor` extends from `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 2: *Example of inheritance in Modelica, where a new subclass `Resistor` is created by extending the base class `TwoPin`.*

However, a common misunderstanding is that subtyping and inheritance is the same concept [58]. 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 Alan Snyder [80], 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 object in a *first-in-first-out (FIFO)* manner. A stack has the same operations, but are 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.

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

Figure 3 shows two C++ classes modeling the properties of a dequeue and a stack. Since 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, but is

```
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 3: *C++ example, where inheritance does not imply a subtype relationship.*

it a subtype? The answer is no, since 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.

2.4 Structural and Nominal Type Systems

During type checking, regardless if it takes place at compile-time or run-time, 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, since integers 4 and 10 are type equivalent. However, this is of course not a valid equation, since 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 4, which illustrates a C++ model similar to the resistor example earlier given as Modelica code in Figure 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`, since `Inductor` does not inherit from `TwoPin`. Moreover, `Resistor2` is not type equivalent to `Resistor` even if they have the same structure and inherits from the same base class, since they are opaquely declared.

In a *structural type system* [75], 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 in [1] and is using transparent type declarations, i.e., Modelica has a structural type system. Consider the `Resistor` example given in Figure 2 and the two complementary models `Inductor` and `Resistor2` in Figure 5. Here, the same relations holds between `TwoPin` and `Resistor`, i.e., that 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, since 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, since `Inductor` contains all public elements (type and name) of the one 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.

```
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 4: *`Resistor` inheritance example in C++.*

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

```
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: *Complementary `Inductor` and `Resistor2` models to the example in Figure 2.*

3 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 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 [58]:

- Subtype Polymorphism
- Parametric Polymorphism
- Ad-hoc Polymorphism

There are other similar categorizations, such as Cardelli and Wegner's [15], where the ad-hoc category is divided into *overloading* and *coercion* at the top level of categories.

3.1 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 an `TwoPin` type and a `Resistor` type. This statement can also be shown according to the rule of subsumption (t-sub) described in Section 2.2.

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 run-time 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[4]. 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, since it might cast down to the wrong subtype. In Java, before version 1.5 when *generics* were introduced, this safety issue was handled using dynamic checks and raising dynamic exceptions if an illegal down-cast was made. Subtype polymorphism is sometimes called "poor-man's polymorphism", since it enables polymorphic behavior, but the safety of down-casts must be handled dynamically [75].

The Modelica language supports subtyping as explained previously, but does not have any operation for down-cast. Since 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 instantly 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.

3.2 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 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 usage of the function.

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

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

Figure 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 of the function is generated for the two function calls. Standard ML on the other hand is using *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 limited version of parametric polymorphism, by using the *redeclare* construct on local class declarations.

3.3 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 [15], 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* [75].

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 have two operands associated with it. The type of these operands decide how the operation should be carried out, i.e., which implementation that should be used.

Overloading can take place at either compile-time or at run-time. Overloading used at run-time is often referred to as *dynamic lookup* [58], *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 run-time 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 [15]:

```
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 2.2 standard.

4 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 [60] 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 [48], but without the aim to more precisely define the exact meaning of a type in the language.

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). Since 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. Since the Modelica language specification is open for interpretation, the presented type definition is our interpretation of the specification.

4.1 Concrete Syntax of 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 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 run-time 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.

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 7: *Concrete syntax of partial Modelica types.*

Let us now take a closer look at the grammar listed in Figure 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 usage 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 usage of prefixes in the next subsection.

Now, let us introduce another model A, which extends from 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. Since Modelica has a structural type system, it is the structure that is interesting, and thus a type only contains the collapsed structure of 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, since 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.

4.2 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 [60] and to the more accessible description in

[30]. Most of the languages prefixes have been introduced in the grammar in Figure 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 [24] and OpenModelica [32], 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.

$M =$	**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 8: *Abbreviation for describing allowed prefixes. Default prefixes are underlined.*

In Figure 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 9, Figure 10, and Figure 11.

In Figure 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 that 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

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

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

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 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 use default prefixes. The defined default prefixes are show underlined in Figure 8. If no underlined prefix exists in a specific set, this implies that the prefix must be explicitly stated.

Analogous to the description of Figure 9, Figure 10 shows the allowed prefixes for elements of object types contained in a class type and Figure 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 11 does not contain any abbreviation a. Thirdly, the `replaceable` prefix does not make sense in object types, since 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, since 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.

4.3 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. Since the latter information is currently not available, the precise type definition is obviously not possible to validate.

5 Conclusion

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

The current Modelica specification uses *Extended Backus-Naur Form* (EBNF) for specifying the syntax, but the semantics and the type system are informally described. Moreover, the Modelica language has become difficult to reason about, since it has grown to be fairly large and complex. By giving the types for part of the language we have illustrated that the type concept is complex in the Modelica language, and that it is non-trivial to extract this information from the language specification.

Consequently, we think that it is important to augment the language specification by using more formal techniques to describe the semantics and the type system. We therefore propose that a subset of Modelica should be defined, which models the core concepts of the language. This subset should be describe using operational semantics including formal type rules. For some time, denotational semantics has been used as the semantic language of choice, however it has been shown to be less cumbersome to prove type soundness using operational semantics [90].

In the short term, this proposed core language is supposed to be used as basic data for better design decision-making, not as an alternative or replacement of the official Modelica specification. However, the long term goal should, in our option, be to describe the entire Modelica language formally.

Acknowledgments

Thanks to Thomas Schön and Kaj Nyström for many useful comments of this paper.

This research work was funded by CUGS (the Swedish National Graduate School in Computer Science), by SSF under the VISIMOD project, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

Paper B

Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta

Authors: David Broman, Kaj Nyström, and Peter Fritzson

Edited version of paper originally published in *Proceedings of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 151-160, Portland, Oregon, USA. ACM Press 2006.

Determining Over- and Under-Constrained Systems of Equations using Structural Constraint Delta

David Broman, Kaj Nyström, and Peter Fritzson

Department of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden
E-mail: {davbr,kajny,petfr}@ida.liu.se

Abstract

Computer aided modeling and simulation of complex physical systems, using components from multiple application domains, such as electrical, mechanical, and hydraulic, have in recent years witnessed a significant growth of interest. In the last decade, equation-based object-oriented (EOO) modeling languages, (e.g. Modelica, gPROMS, and VHDL-AMS) based on acausal modeling using Differential Algebraic Equations (DAEs), have appeared. With such languages, it is possible to model physical systems at a high level of abstraction by using reusable components.

A model in an EOO language needs to have the same number of equations as unknowns. A previously unsolved problem concerning this property is the efficient detection of over- or under-constrained models in the case of separately compiled models.

This paper describes a novel technique to determine over- and under-constrained systems of equations in models, based on a concept called structural constraint delta. In many cases it is also possible to locate the source of the constraint-problem. Our approach makes use of static type checking and consists of a type inference algorithm. We have implemented it for a subset of the Modelica language, and successfully validated it on several examples.

Keywords: Equation-based, modeling, object-oriented, separate compilation, type checking, over-constrained, under-constrained.

1 Introduction

Computer aided modeling and simulation have for years provided engineers in all disciplines with powerful tools to design and test complex systems in a faster and more cost-efficient way than physical prototyping. Computerized models also give the advantage of easy extraction of measurements from the model, even those that would be hard or even impossible to get from a physical system.
Historically, imperative implementation languages like Fortran and C have to some extent been replaced by specialized modeling platforms such as Simulink [53]. In recent years, new kinds of modeling languages have emerged, which combine the concept of object-orientation with specification of models using Differential Algebraic Equations (DAEs). We call these languages Equation-based Object-Oriented languages, or EOO languages for short. Modelica [30, 60] is an example of such a language. Other examples are gPROMS [68], χ [28], and VHDL-AMS [18].

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

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

1.1 Constraint Checking of Separately Compiled Components

A model in an EOO language is actually a system of equations describing the model's behavior. The existence of a single 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 when one or more components in the model have been separately compiled.

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. This is almost equivalent to a total recompilation of the entire car model and all its components. This in turn means that separate compilation of the subcomponents would have been completely unnecessary.

1.2 Error Detection and Debugging

If a model intended for simulation has not the same number of equations as variables, it is an error. This can be detected after compiling the model into a system of equations. To

[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.

locate and resolve the error, the system of equations must be inspected. Consider again the car model from Section 1.1. 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 tools [24] and methods [13] that help the user in this process, but they require information of the model’s whole system of equations.

1.3 Contributions

The main contribution of this work is the novel concept of *structural constraint delta*, denoted C_Δ. Our approach makes use of static type checking and consists of a type inference algorithm, which determines if a model is under- or over-constrained without elaborating its subcomponents. This enables separate compilation of components in EOO languages. Furthermore, the concept also allows detection of constraint-errors at the subcomponent level and improves the possibilities of locating the source of the errors.

1.4 Outline

The remainder of this paper is structured as follows. Section 2 describes basic concepts and objectives of object-oriented equation-based modeling. Section 3 gives an overview of a Modelica compiler. Section 4 introduces a minimal EOO language called Featherweight Modelica (FM), its syntax and informal description of semantics and type system. Section 5 defines 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 describes our prototype implementation, Section 7 discusses related work, and Section 8 presents conclusions of this paper.

2 Equation-Based Modeling in Modelica

In this section we illustrate several important concepts in modeling with EOO languages using the Modelica language as an example.

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). Just like in other OO languages, a class contains variables, i.e., class attributes representing data. These attributes are called *elements* of the class and can be instances of classes or built-in types. If the element is an instance of a `model`, this element is also called a `component`. The main difference compared with traditional OO languages is that instead of methods, Modelica primarily uses *equations* to specify behavior. Equations can be written explicitly, like `a=b`, or be inherited from other classes. Equations can also be specified by special `connect`-equations, also called *connections*. For example `connect(v1, v2)` expresses coupling between elements `v1` and `v2`. These elements are called *connectors* (also known as ports) and belong to the connected objects. This gives a flexible way of specifying the topology of a physical system.

2.1 Modelica Model of an Electric Circuit

As an introduction to Modelica, we present a model of an electrical circuit (Figure 1). A composite class like the `Circuit` model specifies the system topology, i.e., the components and the connections between the components. In the declaration of the resistor `R1`, `Resistor` is the class reference, `R1` is the component's name, and `R=10` sets the default resistance, `R`, to `10`.

```
model Circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  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;
```

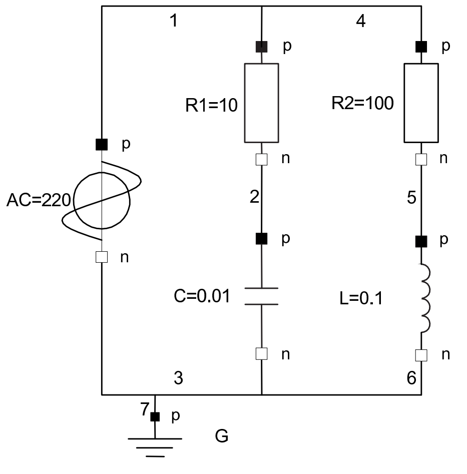


Figure 1: *Modelica model of an electrical circuit.*

2.2 Connector Classes

A connector must contain all quantities needed to describe an interaction. For electrical components we need the variables voltage `v` and current `i` to define interaction via a wire connection. A connector class is shown below:

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

A connect-equation `connect(R1.p,R2.p)` with `R1.p` and `R2.p` being instances of the connector class `Pin`, connects the two pins so that they form one node. This connect-equation generates two standard equality equations: `R1.p.v = R2.p.v` and `R1.p.i + R2.p.i = 0`. The first equation expresses that the voltage of the connected wire ends are the same. The second equation corresponds to Kirchhoff's current law saying that the currents sum to zero at a node. The sum-to-zero equations are generated when the prefix `flow` is used. Similar laws apply to flow rates in a piping network and to forces and torques in mechanical systems. If a model contains an unconnected connector with a flow variable, the compiler will implicitly set this variable's value to zero.

2.3 Base Classes and Inheritance

A common property of many electrical components is that they have two pins. Thus it is useful to define a "base" `TwoPin` component as follows:

```
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 component has two pins `p` and `n`, a quantity `v`, that defines the voltage drop across the component and a quantity `i` that defines the current into the pin `p`, through the component and out from pin `n`. 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 the equation governing the capacitor's behaviour.

```
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.

2.4 Modification and Redeclaration

When extending or declaring an element, we can add *modification equations*. The simplest form of modification is assigning 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.

In this example, we create a model `B` from model `A` and at the same time change resistor `R1` to be a `TempResistor`.

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

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

2.5 Acausal Modeling and Dynamic Systems

Modelica uses acausal modeling, which means modeling based on equations. Equations do not specify if a variable is used for input or output. In contrast, for assignment statements, variables on the left-hand side are always outputs (results) and variables on the right-hand side are always inputs. Thus, the causality of equations-based models is unspecified and becomes fixed only when the corresponding equation systems are solved. In practice, this means that when simulating a model, the user does not have to specify what variable he is interested in. The simulation will produce results for all variables present in the model.

Modelica is primarily used for modeling dynamic systems, where a model's behaviour evolves as a function of time. This means that all variables in a model have a value for every time step for which the model has been simulated. In addition, since we are working with DAEs, all derivatives of variables (denoted `der(v)` for the derivative of `v`) are derivatives with respect to time. An example of using the derivative function is shown in the `Capacitor` model in Section 2.3.

3 The Modelica Compiler

In order to understand the current problem involved in separate compilation of Modelica models we must first explain how a typical Modelica compiler works. The structure of a typical compiler is depicted in Figure 2. It is common to view a Modelica compiler as consisting of two parts. The first part produces a system of equations and the second part produces an executable that solves this system of equations. The first steps in the compilation are scanning and parsing which transforms the Modelica source code into a parse tree. This parse tree is then *elaborated* into equations and variables.

3.1 Elaboration and Type Checking

First we need a few definitions; specific for this kind of language.

Definition 3.1 (Flat system of equations). A flat system of equations is a set of declared variables of primitive types together with a set of equations referencing these variables.

Definition 3.2 (Elaboration). Elaboration is the task of producing a flat system of equations from the parse tree of a set of models.

We will show how elaboration of a model is done by an example. The task is to elaborate model `A` in Figure 3. 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 C`. Our action is then to simply copy the contents of model `C` into our working copy of model

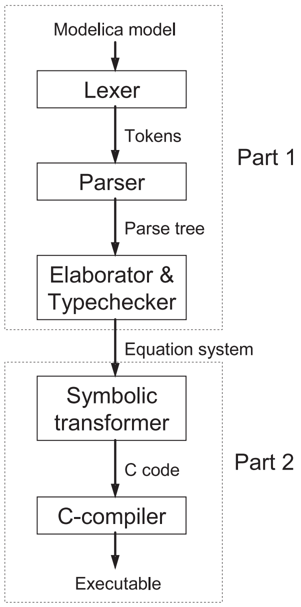


Figure 2: *The structure of a Modelica compiler.*

```
model B                model C                 model A
  Real y;                 Real z=10;             extends C(z=5);
  Real x;                 Real t;                B b;
equation               equation                end A;
  y=der(x);               t=z*2;
end B;                 end C;
```

Figure 3: *Example models to elaborate and type check.*

A. The modification equation to variable z in the extends clause replaces the modification 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 shown in model A1 in Figure 4.

We do not have to do anything about declarations of variables with primitive types. However, the component b must be elaborated since B is not of primitive type. We investigate model B and find that it contains the declarations Real y and Real x. These declarations and all equations in 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 since there are only primitive types left in our working model. The final result of the elaboration is shown as model A2 in Figure 4.

In this compilation strategy, type checking is completely interleaved with the elaboration.

3.2 Symbolic Transformation and Code Generation

After elaboration, a number of operations are performed on the system of equations. Typically, the first step is to check that the number of equations and variables are equal. If this criterion is fulfilled, the compiler can go on to perform symbolic transformation tasks, such as BLT transformation [22, 23, 39, 40]. We will not go into detail of these operations

```
model A1                    model A2
  Real z;                     Real z;
  Real t;                     Real t;
  B b;                        Real b.x;
equation                      Real b.y;
  z=5;                      equation
  t=z*2;                      b.y=der(b.x);
end A1;                       z=5;
                              t=z*2;
                            end A2;
```

Figure 4: *Stepwise elaboration of model A from Figure 3.*

as they are not necessary for the understanding of this paper.

The symbolic transformation module then generates a program, usually in C code. The program uses a numerical solver such as DASSL [74] for solving the system of equations. The generated program can then be compiled with a C compiler which produces the executable which in turn will produce the simulation results.

3.3 Separate Compilation

Separate compilation in Modelica would ideally work as depicted in Figure 5.

The problem with separate compilation in Modelica is that while components may be separately compiled, it is hard to check if a model containing separately compiled components is under- or over-constrained. It seems that we must look at the entire elaborated model (flat system of equations) in order to determine this property.

We now return to the example of the car model mentioned in Section 1.1. Let us assume that Engine, Gearbox, and Axis are very complex models consisting of more than 20000 equations developed by separate teams. It is clearly undesirable to recompile the entire system in order to check how it is constrained. Instead, we only want to elaborate the connect equations and to check the interfaces of the components.

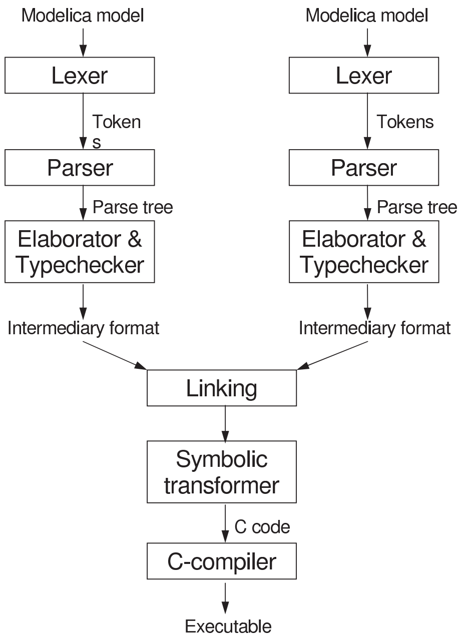


Figure 5: *Separate compilation in Modelica.*

3.4 Concluding Remarks

There are two deficiencies with the current practice in the Modelica compiler 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 since the error is detected at the level of flat system of equations rather than on a component/model level.

4 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.

4.1 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 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 boldface.

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 subscribed numbers 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 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 `Real`s can be modified in FM.

4.2 Type-Equivalence and Subtyping

Modelica is using a so called *structural type system* [75], where the type of a class is determined by the structure of its components. In contrast, other object-oriented languages, such as Java, are using primarily a *nominal type system*, where the name of the declared class identifies the type.

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

Figure 6: *Syntax of Featherweight Modelica.*

The Modelica language specification [60] is informally describing the semantics and type system of the language. From the specification, the following definition of *type equivalence* can be extracted:

Definition 4.1 (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`, since the type only represents the *interface* of the class and not the private *implementation* or *semantic* part, such as equations.

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

Definition 4.2 (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 class `S`'s type.

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

Figure 7: *Three different Modelica models.*

Now, consider the three models given in Figure 7. According to Definition 4.2, we can see that `B <: A` since 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, since both `B` and `C` hold the same public elements, it can be concluded from Definition 4.1 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* [75]. 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 7, 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 observation still regard `B` as a safe replacement of `A`? We think not, and will in the next subsections propose a solution.

5 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 elaboration. We start by giving a definition:

Definition 5.1 (Structural Constraint Delta, $C_\triangle$). Given an arbitrary class C, containing components, equations, and connections, the type of C has a defined integer attribute called structural constraint delta, $C_\triangle$. 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, $C_\triangle = 0$ for a system of equations does not guarantee a unique solution, it will only indicate that a single solution might exist. If $C_\triangle < 0$, we have an under-constrained problem with more unknowns than equations, which might give an infinite number of solutions. If $C_\triangle > 0$, we have an over-constrained system of equations, which most likely will not give a unique solution. However, since the algorithm for computing $C_\triangle$ does not check if equations are linearly independent or not, a system with $C_\triangle > 0$ 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. Since this is obviously not possible when inspecting components separately, the value of $C_\triangle$ only provides a good indication whether a system of equations has a unique solution or not.

For example, if $C_\triangle$ is to be calculated for the types of the models given in Figure 7, the difference between the number of equations and variables in the model gives the value of $C_\triangle$. In this case, $C_\triangle = 0$ for `A` and `C`, but $C_\triangle = -1$ for `B`. Since our models so far only contain variables and equations, calculating $C_\triangle$ 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 $C_\triangle$, we need another definition:

Definition 5.2 (Constraint Delta Effect, $E_\triangle$). 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 $E_\triangle$ is a type attribute of both C1 and C2, which states the effect E has when computing $C_\triangle$ of C.

Note that $C_\triangle$ is not the same as $E_\triangle$. Simply stated, we say that $E_\triangle$ of two elements represents the change of the current model's $C_\triangle$ when an equation or connection is introduced between the two elements. For example, if we in model `B` in Figure 7 introduce a new equation `q = 2 * p`, this equation will have the effect of changing model `B`'s $C_\triangle$ from -1 to 0. Therefore, involved variables `q` and `p`, are said to have $E_\triangle = 1$ (or to be precise; the attributes to the types of the elements). However, we will soon see that elements do not always have $E_\triangle = 1$.

5.1 Algorithms for Computing $C_\triangle$ and $E_\triangle$

In this section, we present algorithms for calculating $C_\triangle$ and $E_\triangle$. Even if the algorithms for calculating the type attributes $C_\triangle$ and $E_\triangle$ 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 paper is not only computer scientists, but also engineers from the modeling and simulation community.

It is important to stress that $C_\triangle$ and $E_\triangle$ 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 $C_\triangle$ and $E_\triangle$ for each subelement, since they are already defined by the type of the elements. The process of calculating $C_\triangle$ and $E_\triangle$ 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.
The algorithm for computing $C_\triangle$ is given in Algorithm 1. This algorithm uses a help function defined in Algorithm 2. The algorithm for computing $E_\triangle$ is listed in Algorithm 3. Note that the indentation of the algorithms is significant and delimits blocks for the `foreach`, `if`, and `switch` statements.
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 $C_\triangle$ 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.
- **hasDefualtValue**(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 checking, since two outside or inherited connectors are connected, or a connected connector is both outside and inherited.

Algorithm 1: Compute C_Δ of a class

```
Input: An arbitrary $Class$
Output: $C_\Delta$ of the class
$C_\Delta \leftarrow 0$
switch $Class$ do
    case model
        foreach $e \in$ getElements($Class$) do
            $C_\Delta \leftarrow C_\Delta +$ getDelta($e$)
            if hasDefaultValue($e$) then
                $C_\Delta \leftarrow C_\Delta + 1$
            foreach $m \in$ getModifiedElements($e$) do
                if not hasDefaultValue($m$) then
                    $C_\Delta \leftarrow C_\Delta + 1$
        foreach $e \in$ getEquations($Class$) do
            $C_\Delta \leftarrow C_\Delta +$ getEffect($e$)
        foreach $c \in$ getConnectors($Class$) do
            $P_{outside} \leftarrow$ FALSE
            $P_{inherited} \leftarrow$ FALSE
            if  not isVisited($c$) then
                traverseConnectorGraph($c$)
                if $P_{outside}$ then
                    $C_\Delta \leftarrow C_\Delta +$ getOutsideAdjustment($c$)
        foreach $b \in$ getBaseClasses($Class$) do
            foreach $m \in$ getModifiedElements($b$) do
                if not hasDefaultValue($m$) then
                    $C_\Delta \leftarrow C_\Delta + 1$
            $C_\Delta \leftarrow C_\Delta +$getDelta($b$)
    case record
        foreach $e \in$ getElements($Class$) do
            $C_\Delta \leftarrow C_\Delta +$ getDelta($e$)
        foreach $b \in$ getBaseClasses($Class$) do
            $C_\Delta \leftarrow C_\Delta +$ getDelta($b$)
    case connector
        foreach $e \in$ getElements($Class$) do
            if  not hasFlowPrefix($e$) then
                $C_\Delta \leftarrow C_\Delta +$ getDelta($e$)
            foreach $b \in$ getBaseClasses($Class$) do
                $C_\Delta \leftarrow C_\Delta +$ getDelta($b$)
    case variable
        $C_\Delta \leftarrow -1$
end
```

Algorithm 2: traverseConnectorGraph(c_1)

```
Input: Connector $c_1$ from which graph traversal starts
Output: Global variables $P_{outside}$, $P_{inherited}$, and $C_\triangle$
if ((isOutside($c_1$) and isInherited($c_1$)) or ((isOutside($c_1$)
     or isInherited($c_1$)) and ($P_{outside}$ or $P_{inherited}$)) then typeCheckingFailed()
else
    markAsVisited($c_1$)
    $P_{outside} \leftarrow P_{outside}$ or isOutside($c_1$)
    $P_{inherited} \leftarrow P_{inherited}$ or isInherited($c_1$)
    foreach $c_2 \in$ getAdjacencyConnectors($c_1$) do
        if not isVisited($c_2$) then
            $C_\triangle \leftarrow C_\triangle +$ getEffect(getTypeOf($c_2$))
            traverseConnectorGraph($c_2$)
```

Algorithm 3: Compute $E_\triangle$ of a class

```
Input: An arbitrary $Class$
Output: $E_\triangle$ of the class
$E_\triangle \leftarrow 0$
switch $Class$ do
    case record
        foreach $e \in$ getElements($Class$) do
            $E_\triangle \leftarrow E_\triangle +$ getEffect($e$)
        foreach $b \in$ getBaseClasses($Class$) do
            $E_\triangle \leftarrow E_\triangle +$ getEffect($b$)
    case connector
        foreach $e \in$ getElements($Class$) do
            if hasFlowPrefix($e$) then
                $E_\triangle \leftarrow E_\triangle -$getEffect($e$)
            else $E_\triangle \leftarrow E_\triangle +$ getEffect($e$)
        foreach $b \in$ getBaseClasses($Class$) do
            $E_\triangle \leftarrow E_\triangle +$ getEffect($b$)
    case variable
        $E_\triangle \leftarrow 1$
end
```

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 $C_\triangle$ for each of the models without any knowledge of the internal semantics of the subcomponents, i.e., the equations. Calculated $C_\triangle$ and $E_\triangle$ 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 $C_\triangle$ value for this class. First, we can see that calculating $C_\triangle$ of a record simply adds the $C_\triangle$ value for each element (rows 26-27), which in the case of record `R` gives $C_\triangle = -2$ since `R` holds 2 variables. In Algorithm 3, we can see that calculating the effect of `R` gives $E_\triangle = 2$. But what does this mean? Recall that $E_\triangle$, given in Definition 5.2, states the effect on $C_\triangle$ 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 $E_\triangle$ for `R` is 2. The rest of the procedure for computing $C_\triangle$ of model `A` should be pretty straightforward by following Algorithm 1. Note that only $C_\triangle$ and not $E_\triangle$ is given for models, since models are not allowed to be interconnected.

The more interesting aspects of calculating $C_\triangle$ in this example are shown in model `M`. First of all, we can see that model `M` extends from `A`, which results in that $C_\triangle$ of `A` is added to $C_\triangle$ of `M` (see rows 20-24 in Algorithm 1). Since variable `p` is modified with `p=1`, we see that $C_\triangle$ is increased by $E_\triangle$ of the type of `p`, i.e., `Real`. Hence, the $C_\triangle$ contribution from base class A is -2. The $C_\triangle$ 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 effect $C_\triangle$, since 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 $C_\triangle$ value of -1.

Computing $C_\triangle$ - Connectors, Connections, and Flow Variables

Consider the source code listing and graphical representation given in Figure 8. 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 represents 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 $C_\triangle$ of connector C can be achieved by using rows 30-35 in Algorithm 1. On row 32, we can see that $C_\triangle$ 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., $C_\triangle = 0$. Further inspection of the algorithm, yields $C_\triangle = -2$ for model K.

```
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;
```

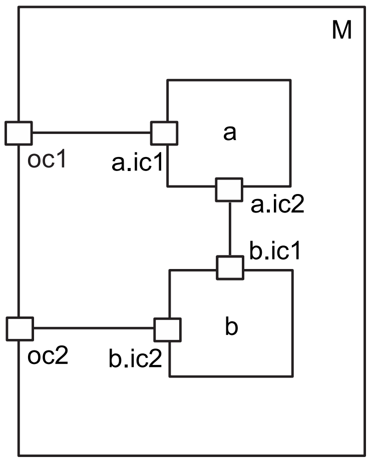


Figure 8: *Model M with inside connectors (e.g. a.ic1 and b.ic2) and outside connectors (oc1 and oc2).*

Calculating $C_\triangle$ 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 $C_\triangle$. Each connect-equation (edge) in the graph contribute to the $C_\triangle$ of the class being computed, by adding $E_\triangle$ of a connector in the connection (see row 9 in Algorithm 2). Since 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 $E_\triangle$.

By using Algorithm 3, rows 9-12, we can see that $E_\triangle = 0$ for connector `C`. Consequently, all the connections in model `M` will not change the value of $C_\triangle$. 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, $E_\triangle$ 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 $E_\triangle = -1$.

There are several aspects covered by the algorithms, which we will not be able to explain in detail in this paper, due to space limitations. 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 since without it, the type of a class must be extended with information regarding the connectors that are connected.

5.2 Extending the Type System with C_Δ

The latter sections described how we can calculate C_Δ and E_Δ 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 FM$_\Delta$, is defined by extending Definition 4.1 and Definition 4.2 for type-equivalence and subtyping with the following definitions:

Definition 5.3 (Type-equivalence and C_Δ). For any types T and U to be type-equivalent, Definition 4.1 must hold and the C_Δ-value of T and U must be equal.

Definition 5.4 (Subtyping and C_Δ). For any types S and C, S is a supertype of C and C is a subtype of S if Definition 4.2 holds and the C_Δ-value of S is equal to that of C.

Hence, the extended language FM$_\Delta$ 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 7, we can now see that model `C` is a subtype of model `A`, but model `B` is not.

6 Prototype Implementation

To validate and verify our algorithms, a prototype Featherweight Modelica Compiler (`fmc`) was implemented consisting of a type-checker for FM$_\Delta$, where C_Δ and E_Δ automatically are inferred and represented as attribute to the types. The prototype compiler was implemented as a batch-program, which takes a FM$_\Delta$ `.mo`-file (containing FM$_\Delta$ 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$_\Delta$.
2. Run the prototype compiler for FM$_\Delta$ 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 [24] and OpenModelica version 1.4.1 [32].

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 components, and error detection and debugging. In the examples, `fmc` and Dymola version 6 are used when testing the models.

6.1 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 are 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 $C_\triangle$=0
  public final connector objtype $C_\triangle$=-1 $E_\triangle$=0
    nonflow Real objtype v;
    flow Real objtype i;
  end p;
  public final connector objtype $C_\triangle$=-1 $E_\triangle$=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              model VsourceAC
    Pin p;                    extends TwoPin;
  equation                    Real VA = 220;
    p.v = 0;                  Real f = 50;
  end Ground;                 Real PI = 3.1416;
                            equation
                              v = VA*sin(2*PI*f*time);
                            end VsourceAC;

  model Inductor            model Circuit
    Pin p;                  protected
    Pin n;                    replaceable Resistor R1(R=10);
    Real v;                   replaceable Inductor L(L=0.1);
    Real i;                   VsourceAC AC;
    Real L = 1;               Ground G;
  equation                  equation
    L*der(i) = v;             connect(AC.p, R1.p);
  end Inductor;               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.

6.2 Error Detection and Debugging

Now the following question arise: How can we know where the problem is located? The user needs to either analyse 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 $C_\triangle$ 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, since 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.

7 Related Work

We have used the Modelica language as an example to explain the problems associated with over- and under-constrained systems. These problems arise in languages using hierarchical modeling with components, where the component semantics contain DAEs. While it is trivial to count equations in a simple model, we have seen that the complexity increases when introducing connect semantics, existing in e.g. the χ [28] language. Both flow variables, used in e.g. VHDL-AMS [18] (called `through`) and inheritance part of e.g. gPROMS [68], complicate matters further.

The Modelica language includes all these concepts, and there exist methods for locating errors at the level of flat system of equations [13]. The Modelica tool Dymola [24] detects constraint-errors at the flat system of equations, and can sometimes also pinpoint the errors. One downside with these approaches is that the entire model must be elaborated, making separate compilation difficult.

An attractive simplification related to the C_Δ concept would be to require all separately compiled models to have the same number of equations as unknowns, i.e., $C_\Delta = 0$. However, it is an open question if this approach is not too conservative for expressing models in the general case.

To the best of our knowledge, no solution has previously been presented for any applicable language that determines if a model is under- or over-constrained, without elaborating the model.

8 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.

Acknowledgments

Thanks to Peter Bunus, John Wilander, Thomas Schön, and Åsa Broman for many useful comments. This research work was funded by CUGS (the National Graduate School in Computer Science), SSF in the Visimod-II project, MathCore Engineering, and by Vinnova in the NETPROG Safe & Secure Modeling and the GridModelica projects.

Paper C

Abstract Syntax Can Make the Definition of Modelica Less Abstract

Authors: David Broman and Peter Fritzson

Edited version of paper originally published in *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT'07)*, pages 111-126, Berlin, Germany. Linköping University Electronic Press, 2007.

Abstract Syntax Can Make the Definition of Modelica Less Abstract

David Broman and Peter Fritzson

Department of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden
E-mail: {davbr,petfr}@ida.liu.se

Abstract

Modelica is an open standardized language used for modeling and simulation of complex physical systems. The language specification defines a formal concrete syntax, but the semantics is informally described using natural language. The latter makes the language hard to interpret, maintain and reason about, which affect 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 paper, 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. 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.

1 Introduction

Modelica is an open standard language aimed primarily at modeling and simulation of complex physical systems. The first language specification 1.0 [59] was released in September 1997. Since then, the current specification 2.2[60] has evolved to be large and complex with many constructs.

During these past ten years, the user community has grown fairly large and the Modelica Standard Library has evolved to include several physical domains. The dominating Modelica tool has for a long time been the commercial tool Dymola [24]. However, during recent years, alternative tools have emerged; both open source (OpenModelica [33, 69]) and commercial environments (e.g., MathModelica System Designer [52], MOSILAB [27], and SimulationX[43]).

The rapidly growing user community and increasing number of tool vendors augment the demand of the language specification being precise so that different tools will be compatible. Hence, the Modelica Association, who is responsible for the language specification, has defined the goals for the next language version both to make the specification clearer and to simplify the language itself.

1.1 Specification of the Modelica Simulation process

Modelica's compilation and simulation process can be divided into several stages or subprocesses. Consider Figure 1, where a Modelica model is *elaborated*[1] into a Hybrid Differential Algebraic Equation (Hybrid DAE) and then transformed into an executable, which after execution produces a simulation result.

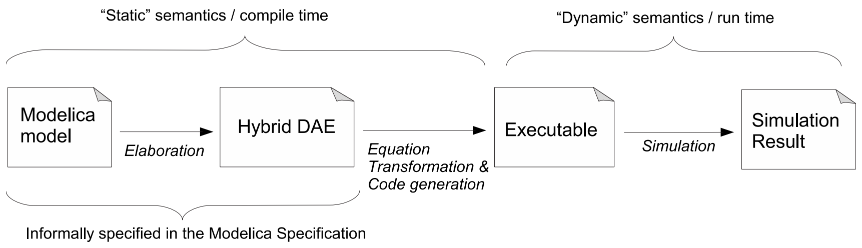

"Static" semantics / compile time
"Dynamic" semantics / run time
Modelica model
Elaboration
Hybrid DAE
Equation Transformation & Code generation
Executable
Simulation
Simulation Result
Informally specified in the Modelica Specification

Figure 1: *Overview of a typical Modelica compilation and simulation process.*

The syntax and semantic analysis take place at compile time and the generation of simulation output is produced at run-time.

In the current specification 2.2 [60], the concrete syntax is stated formally using Extended Backus-Naur From (EBNF), but only the semantics of the first part of the process

[1]In this paper, we call this process *elaboration*. In the Modelica specification 2.2, this process is called *instantiation*. Sometimes, this transformation is also referred to as the *flattening* phase, since it creates a flat system of equations. However, we think that both these terms are misleading. The former, since it is performed at compile time and is not allocating memory analogous to instance creating in standard programming languages. The latter, since the final equation system does not need to be flat - it can still be represented in a hierarchical structure.

is informally described using natural language backed up with concrete source code examples.

Due to the fact that output of this process is not precisely defined, and that the semantics is described informally using natural language, the current specification is to a high degree open for interpretation.

1.2 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 [76], the language semantics is precise and can in some cases be proved to have certain properties, such as type safety [75, 90]. However, to understand and interpret a formal language specification require a rigorous theoretical computer science knowledge. Furthermore, even if great effort has been spent during the last decades in formalizing mainstream programing languages, only a few, e.g., Standard ML [57], 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 [36], C++ [42], and Modelica [60]. 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.

1.3 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 on the next language specification includes substantial restructuring and a more detailed description of the semantics of the language. However, it is not planned to include any formal descriptions, apart from an appendix containing one possible definition of Modelica abstract syntax.

Natural Semantics

Already in 1998 Kågedal and Fritzson [47, 48], created a formal specification for a subset of the Modelica language, influenced by the language specification examples described in the 1997 version of [31]. The specification was using *Natural Semantics* [45] and the executable specification language Relational Meta Language (RML) [73]. This work influenced the design of the language and the official Modelica specification. The executable specification has gradually evolved and is now the code basis for the OpenModelica project[69]. In 2006, the code base was converted from RML to Meta-Modelica[34] with the purpose of making it more accessible for 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. One lesson learned from this specification project was that for an almost complete specification of an early Modelica language version, the formal specification became hard to get an overview of, since it grew to be very large.

Elaboration

Jakob Mauss has made several contributions to formally describe 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. The published work [54] describes an algorithmic specification approach, which focuses on Modelica's complex lookup 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. It exists also a refined version of this work, which uses a more compact notation. However, this work is still unpublished.

Modelica Types

In our previous work on types in the Modelica language[11], we concluded that the type concept is only implicitly defined in the Modelica language specification. In that work, we proposed a concrete syntax of specifying Modelica types and gave a suggestion for constraining information of element prefixes in the types. Furthermore, it was emphasized that Modelica has a *structural type system*, which implies that a *class* and a *type* are two separate language concepts. In this paper, we will not cover types, even though parts of a specification can also be described using type rules.

A common dominator 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 question is how to make this possible in practice, since all attempts so far only model subsets of the real language.

1.4 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, since this process is tedious and error prone, it is very hard to ensure that the ambiguity decreases. Moreover, 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 model, the result is normally stored internally as an *Abstract Syntax Tree (AST)*. Hence, one particular model results in a specific AST, which can be seen as an instance of the language's abstract syntax. The abstract syntax can be specified using a *context-free grammar*, and an AST can also have a corresponding textual representation.

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 paper we propose that the intermediate representations between the transformation steps (recall Figure 1) should be described by specifying its abstract syntax.

However, specifying different forms of abstract syntax *cannot* replace the semantic specification need in the transformation process, since the syntax only describes the *structure* of a 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 more described using techniques such as Syntax-Directed Translation Schemes (SDT)[2] or different forms of operational semantics. However, as earlier described, 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 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 details that can be specified of the abstract syntax, we will later in the paper 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, great parts of the rejecting semantics must still be described using another semantic specification form.

In the following sections, we will gradually introduce more motivations and descriptions of the abstract syntax approach. Section 2 gives an overview of different aspects of specifying a language specification in the context of Modelica. The discussion on different specification alternatives and aspects forms the basis for Section 3, which more concretely elaborates on our proposal. Finally, in Section 4 concluding remarks are stated and future work is outlined.

2 Specifying the Modelica Specification

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 Differential Algebraic Equations (Hybrid DAE), or the final simulation result.

- *Rejection*, 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.

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

In the introduction section of the Modelica specification 2.2 [60], 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 handled 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.

So, the 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, since the current specification is open for interpretation. In this subsection, we give our view of a typical Modelica transformation process.

Recall Figure 1, where the high-level view of a typical Modelica compilation and simulation process is outlined. The translation process is divided into three sub-processes, each having an artifact as input and output.

Elaboration

The *elaboration* process (also called *instantiation* and sometimes *flattening*) takes as input a source code Modelica model and transforms it into a Hybrid DAE. This is the main part described in the Modelica specification, which includes among other things parsing, type checking, redeclarations, connection elaboration, and generation of equations. The output is the Hybrid DAE, which includes items such as equations, function calls, algorithm sections, declaration of variables etc.

Equation Transformation and Code Generation

The Hybrid DAE is simplified and transformed (index reduction, generation of Block Lower Triangular form (BLT)). Finally, target code is generated (typically C-code), which is linked together with a numerical solver, such as DASSL[74].

Simulation

The final transformation step is basically running the executable, where the actual simulation takes place. During this step, numerical integration of the continuous system and discrete event handling occurs.

Static vs. Dynamic.

In the example above, 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. If the tool is implemented as an interpreter, the distinction becomes less clear. In such a case, it is natural to view all three processes as the dynamic semantics. Even if this is only a matter of definitions, it becomes significantly important when reasoning about type checking and separate compilation.

From the discussion above, 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.

2.2 Rejection 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 it 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 have great implications on the tools ability to guarantee detection of invalid models.

Figure 2 outlines a simplified view of the earlier described compilation and simulation process, where sub-processes of equation-transformation, code generation and simulation are combined into one transformation step.

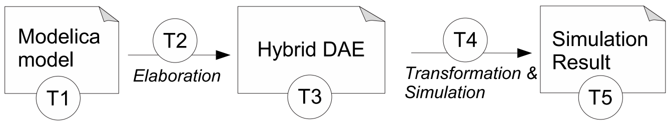


Figure 2: *Possible checking-time during the process*

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*, since it is performed during runtime. Errors which can occur here are for example numerical singularities after events or array out-of-bound errors. Since 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 the control 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, since 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[12] or to enable separate compilation.

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 paper.

As evaluation criteria, it is natural to use the specification goals of *understandability*[2] 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	Low
Reference Implementation	Low	High	High

Table 1: *Possible specification approaches with estimated evaluation criteria.*

A natural language specification can be understandable and expressive, depending on the

[2]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.

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 [75] is here seen as having low understandability, since 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, since we can introduce context dependent information in the grammar using metavariables. An example of this will be given in the next section.
We have also, for the sake of completeness, included related 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 state which description that has precedence if ambiguities are discovered. Finally, a reference implementation can also be seen as a specification, even if it is hard to get an good overview and reason about it.

3 An Abstract Syntax Specification Approach

In the following section we will go into more details about the proposal 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 2.1, followed by a discussion about the specification and representation of the syntax. Finally a small example of abstract syntax grammar is given and discussed.

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 an compiler can be defined as transformations from one intermediate representation to another. ASTs are a natural form of intermediate representation.

Consider Figure 3, 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 sections into one list, since the Modelica syntax allows several algorithm and equation sections to be defined in a model.

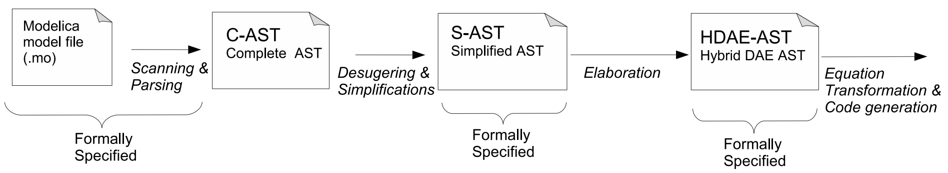


Figure 3: *Modelica's compilation process divided into intermediate representations in the form of abstract syntax trees (ASTs).*

- *Canonical 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 (e.g., Section 3.2.2.1 in [60]).

- *Checking model validity.* One of the purposes with S-AST is that it is more restrictive than the C-AST. Hence, some C-AST are not valid S-AST. This restriction gives the possibility to ensure certain model properties, which in the current Modelica specification is 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[57]. However, initially, we do not see a similar short and precise way of specifying the transformation from C-AST to S-AST, as the transformation rules are given in the Standard ML specification.

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.

3.2 Specifying the Abstract Syntax

The specification of the syntax must be described using some kind of *grammar*, or data type construct in a language such as in Haskell, Standard ML, or MetaModelica [34].

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. The notation has to some extent similarities to and is inspired by the abstract syntax definition of Featherweight Java[41].

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 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* [75]. 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[47] 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.

3.4 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 4, where an example of the non-terminal for a `connector` is listed using a variant of Extended BNF[3]. As usual, alternatives are seprated using the '|' symbol, and curly brackets $(\{\ldots\})$ denote that the enclosing elements can be repeated zero or more times.

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

[3]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.

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

Figure 4: *Example of a grammar for the connector non-terminal.*

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).

3.5 What can and should be specified by the abstract syntax?

In the previous sections we have briefly outlined how an abstract syntax grammar can specify the structure of input and output of a transformation, but also as a method for specifying context-dependent information about rejection of illegal models. The question then arise: what should be specified using this grammar approach, and what should be addressed with other semantic rules?

The proposed grammar approach with meta-variables is declarative in the sense that it does not state information about how the rejecting rules should be implemented. Hence, it is less formal compared to e.g. a formal type system. However, it is still more precise than giving the rules using natural languages.

We believe that as long as the alternative semantic description is using natural languages, the abstract syntax approach can both be easier to understand and less ambiguous. Furthermore, if it can be complemented with aspects which are more precisely described, e.g. the lookup-process, it can clarify the specification even more. However, several parts of the rejection aspect, e.g. subtyping rules, cannot be described with the abstract syntax grammar. The other aspect of transformation semantics can of course not be specified with this approach.

The concept is still at a very early stage, and further investigations need to be performed, to see if this approach can cover the current Modelica language.

4 Conclusion

In this paper we have given an overview of different aspects of defining a modeling language; using the Modelica language's syntax and semantics.

Furthermore, we have argued that an approach which uses *abstract syntax* to describe both the input to Modelica's elaboration process (S-AST) as well as its output (HDAE-AST) can both clarify the transformation process as well as the rejection of invalid models. Furthermore, while developing the language, this approach promotes the focus on semantic issues, to avoid getting trapped in the common syntax pitfall.

The obvious next step for future work would be to design and implement the S-AST and HDAE-AST, and to verify that the ASTs meets most of the current code base publicly available.

We have described this as an evolutionary approach, which is intended to be practical in the short-term. However, in the long term, we still think that it is important that a formal semantics is given for the Modelica language.

Acknowledgments

We would like to thank the anonymous reviewers for their suggestions and Johan Åkesson and Åsa Broman for useful comments and feedback.

This research work was funded by CUGS (the Swedish National Graduate School in Computer Science), by SSF under the VISIMOD project, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

Paper D

Secure Distributed Co-Simulation over Wide Area Networks

Authors: Kristoffer Norling, David Broman, Peter Fritzson, Alexander Siemers, Dag Fritzson

Edited version of paper originally published in *Proceedings of the 48th Scandinavian Conference on Simulation and Modeling (SIMS'07)*, Gothenburg, Sweden. Linköping University Electronic Press, 2007.

Secure Distributed Co-Simulation over Wide Area Networks

Kristoffer Norling[⋆], David Broman[⋆], Peter Fritzson[⋆],
Alexander Siemers[†], Dag Fritzson[†]

[⋆]Department of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden
E-mail: {x06krino,davbr,petfr}@ida.liu.se

[†]SKF Engineering Research Centre
MDC, RKs-2, SE-415 50 Göteborg, Sweden
E-mail: alexander.siemers@skf.com
dag.fritzson@skf.com

Abstract

Modeling and simulation often require different tools for specialized purposes, which increase the motivation to use co-simulation. Since physical models often are describing enterprises' primary know-how, there is a need for a sound approach to securely perform modeling and simulation. This paper discusses different possibilities from a security perspective, with focus on secure distributed co-simulation over wide area networks (WANs), using transmission line modeling (TLM). An approach is outlined and performance is evaluated both in a simulated WAN environment, and for a real encrypted co-simulation between Sweden and Australia. It is concluded that several parameters affect the total simulation time, where especially the network delay (latency) has a significant impact.

Keywords: Modeling, Co-Simulation, Transmission Line Modeling, Security, Data communication.

1 Introduction

The interest for modeling and simulation of complex physical systems, such as aircrafts and trains, has dramatically increased during the last decades. There exist both commercial modeling and simulation environments for specific domains, such as Adams [81] for mechanical systems, and specialized environments for certain application areas, e.g., SKF's BEAST [82, 83] dedicated for bearing simulations. Moreover, new standardized acausal modeling languages for mixed-domains, e.g., Modelica [61] and VHDL-AMS [37], are gaining popularity in various applications, leading to a wide variety of models and specialized tool environments within an enterprise. These models and components are often dependent on each other and commonly need to be simulated together.

One technique that has proven to be both stable and efficient for simulating existing models is *co-simulation*, where sub-models are coupled together using *transmission line modeling (TLM)* [29, 49, 50, 62]. The technique makes use of physically motivated time delays, enabling both numerically robust simulations and better performance using parallel processing.

Since modeling and simulation is becoming a common activity within enterprises, large amount of money and knowledge are invested into such models. Hence, models are becoming critical business assets describing primary know-how.

Within larger enterprises, different departments can be spread out over of the world, have special modeling competencies, and model different parts of systems. Usually, there are defined confidentiality levels within an organization, requiring models to be protected against unauthorized disclosure. Furthermore, models need to be protected from modification by mistake and still being available for large scale reuse. Even if confidentiality issues with sub-contractors are today normally regulated by contracts, the need to protect and keep control of critical business assets is still vital.

Hence, in a co-simulation environment, it is important to have a sound approach for *secure modeling and simulation*, i.e., to create, modify, distribute, and simulate models in an acceptable manner according to the enterprise's security policy.

1.1 Approaches to Secure Modeling and Simulation

Information security can be divided into three aspects[1]:

- *Confidentiality*: protection against unauthorized disclosure of information.
- *Integrity*: protection against unauthorized creation, modification, or deletion of information.
- *Availability*: the assurance that authorized entities have access to correct information when needed.

The importance of these three areas for secure modeling and simulation will be outlined in the following section.

[1]Even if most practitioners in the field of information security recognize these three aspects as fundamental, there is no exact border between e.g., robustness and availability. Here we treat robustness of the system as being part of the availability of simulation and model information.

The obvious first phase in modeling and co-simulation is the modeling phase, which can be divided into three steps, each requiring different expertise [62, 78]:

- Modeling of sub-models in specialized environments, e.g., Adams and BEAST models.
- Encapsulate sub-models and define interfaces.
- Design a *meta-model*, where the different sub-models are integrated and connected to each other.

It is of great importance to perform these steps in a secure manner. However, in this paper we have chosen to focus on how the models are *securely distributed and simulated*, leaving secure modeling as future research.

We have identified two general possible approaches for secure distribution and simulation:

- *Centralized simulation with secure model exchange.* The models are centralized and simulated at one location. The models must be encrypted to avoid unauthorized disclosure of the model content.
- *Secure distributed co-simulation.* The models are kept within their departments/-companies and simulated locally. Simulation data is exchanged between the locations during simulation, making use of co-simulation technologies.

Both these approaches raise several questions regarding security aspects of confidentiality, integrity and availability, as well as practical simulation feasibility.

The former centralized approach uses traditional simulation techniques, where all models are locally available. To provide confidentiality and avoid disclosure of model content, the models need to be encrypted. This sounds initially as a sound approach, but even if the model was encrypted when sent to the other locations, it still needs to be decrypted at the centralized location. This means that the simulation environment needs to know the encryption key. Hence, the information is only *practically confidential*, i.e., it might be hard to get the model in clear text, but not theoretically impossible[2].

A second alternative to model encryption can be *model obfuscating*, i.e., to rearrange the models structure making it unreadable, without changing the model's behavior. Another variant is to distribute the models in another format, e.g., as binary executables.

A certain level of integrity protection, e.g., to achieve need-to-know access (least privilege), can be used by applying message authentication codes (MAC) on the models. However, the same problematic issues regarding keys are unfortunately applicable here as well. Moreover, the centralized approach cannot control the number of times a model is used or that it is not illegally redistributed.

Figure 1 outlines a scenario of the second approach of distributed co-simulation, where a car model is simulated at a location in the USA and the bearing model is simulated in Sweden. To make this scenario possible, a co-simulation technique such as the

[2]In cryptography, it is never impossible to decrypt data without a key, but it can be *computationally very hard.* E.g., using a brute-force approach, it might take hundreds of years to decrypt a specific model.

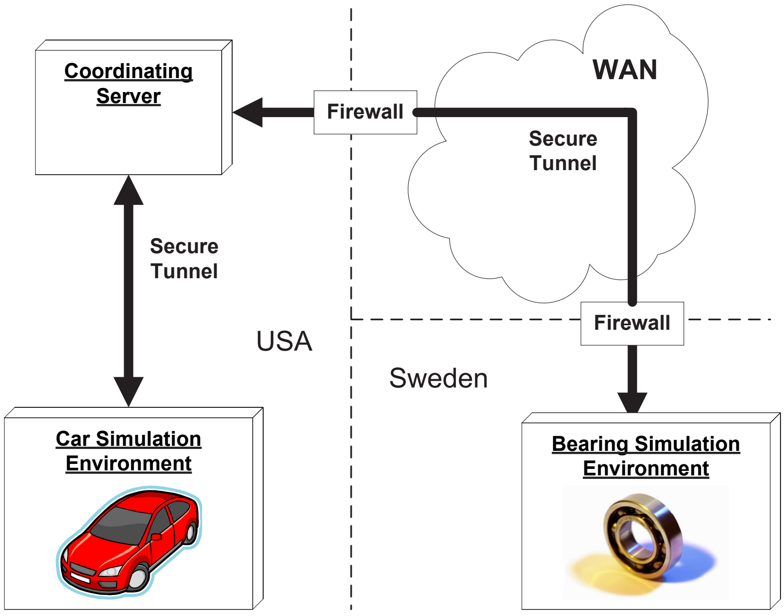


Figure 1: *A scenario of secure distributed co-simulation distributed over long distances.*

one based on Transmission Line Modeling (TLM) can be used. Models are not sent between the different locations. Instead data describing states of the models' interfaces are exchanged between the locations during simulation, using a wide area network (WAN), e.g., the Internet. This scenario leads to several security observations:

- *Confidentiality:* The models are never exchanged between the parties and are therefore never exposed to the other party, i.e., confidentiality is kept using a black-box view of the model. If the TLM data sent between the simulation nodes is encrypted, unauthorized disclosure of the traffic is avoided from third parties. Furthermore, the permission to reuse models for simulation can be controlled, since the models never leave the original location.

- *Integrity:* Unauthorized modification or changes by mistake of models are avoided due to the fact that models are never sent between the locations. Note however that the meta-model needs to be defined and maintained at one location. Integrity protection of simulation traffic can be protected using standard MAC technique.

- *Availability:* Co-simulation using TLM has been shown to provide numerically robust results. However, if the data communication link is not robust or is too slow, and data is not delivered in time, the simulation can be slowed down or terminated. Hence, performance, robustness, and total simulation time are critical factors concerning distributed co-simulation.

Both approaches have pros and cons. A centralized approach raises several problematic issues regarding true confidentiality and integrity of models. On the other hand, centralization can guarantee better communication networks and may therefore result in better performance.

The distributed approach is appealing due to the fact that the black-box characteristics give both better integrity and confidentiality properties of models. In this approach, computational resources from different locations can also be shared and used in a distributed manner. However, the extra overhead needed for protection of data sent during simulation and the cost of transmitting data traffic over large distances can naturally affect performance of the total simulation time.

1.2 Challenges and Contributions

We have chosen to focus on the second approach of *secure distributed co-simulation*, due to its promising properties of confidentiality and integrity. Hence, this paper will primarily deal with aspects of performance and robustness, with the following questions:

- Is it even practically possible to co-simulate stiff bearing models in Sweden together with other mechanical models, which are simulated as far away as the USA or Australia?
- If it is possible, how much longer time does it take to simulate?
- Which parameters and factors affect the total simulation time? Which dependencies exist between these parameters? Are they linear or are there certain breakpoints?

The main contribution of this work is the described approach of secure distributed co-simulation together with conclusion draw from analysis of an experimental simulation test.

1.3 Paper Outline

The paper is structured as follows. Section 2 introduces the fundamental theory of Transmission Line Modeling (TLM) for co-simulation and the basic concepts of data communication. Parameters and factors that may affect the total simulation time are discussed. Section 3 outlines the experimental setup used for generating simulation data from a model reflecting the scenario described in the introduction. The physical model, simulation framework, deployment structure, and simulation tools are described. Section 4 presents the results from the experiment followed by analysis of how different parameters affect the duration of the total simulation time. Finally, section 5 states conclusions of the work.

2 Parameters Affecting the Total Simulation Time

In a distributed co-simulation environment, several parameters and factors can affect the total simulation time. First, we will consider the equations stating transmission line modeling. Second, different factors of the data communication link will be discussed.

2.1 Transmission Line Modeling

The theory of transmission line modeling (TLM) has evolved from the telegraph equations, which concern transmission of electrical signals over long wires.

The TLM method can also be used in other domains, such as mechanical systems where force and velocity are affecting a system. The main motivation to use TLM in such a system is that it describes physically relevant time delays. These delays allow different parts of the system, which are separated by TLM interfaces, to be simulated independently of each other's time steps. Hence, this technique enables the subsystems to be simulated in parallel. Moreover, since exchange of data is needed only between welldefined interfaces, it is also possible to co-simulate between different simulation environments and tools. The latter fact is used in SKF's co-simulation framework [62, 78, 79].

Figure 2 outlines the main variables involved in a transmission line in a mechanical context. The velocity and reaction forces on each side of the line affect the wave variables c_1 and c_2.

The equations involved in the TLM connection are as follows [49, 62]:

$$\begin{aligned}
c_1(t) &= F_2(t - T_{TLM}) + Z_c v_2(t - T_{TLM}) \\
c_2(t) &= F_1(t - T_{TLM}) + Z_c v_1(t - T_{TLM}) \\
F_1(t) &= Z_c v_1(t) + c_1(t) \\
F_2(t) &= Z_c v_2(t) + c_2(t)
\end{aligned}$$

Here c_1 and c_2 denote the force waves, F_1 and F_2 reaction forces and v_1 and v_2 velocities. There are two parameters specified in the model: T_{TLM} indicating the delay time through the line and Z_c that is the characteristic impedance. These parameters must be assigned values, which are within physically acceptable boundaries. See [49, 50, 62] for a more detailed discussions about the physical aspects of the model.

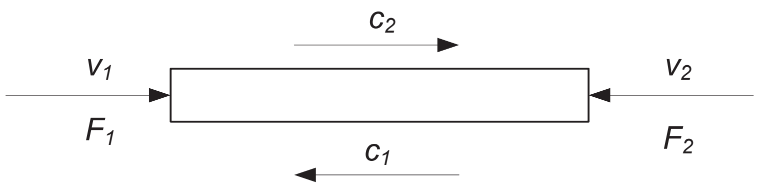


Figure 2: *Delay line with wave variables c_1, c_2, velocity v_1, v_2, and reaction forces F_1, F_2.*

From the equations above, we can clearly see that only old values of F_2 is needed to calculate F_1, and only old values of F_1 is needed to calculate F_2. The allowed length of this delay depends on the parameter T_{TLM}. Larger values of T_{TLM}, will allow the simulations to take larger time steps, which may result in shorter simulation time. Hence, T_{TLM} is an important parameter affecting the total simulation time.

2.2 Data communication

During the co-simulation a continuous flow of data packages containing TLM information are transmitted between the computation nodes. Before the data is transmitted over the WAN, it is being encrypted and message authentication codes (MAC) are created for integrity checks. When the data packet is received at the other node, it is being decrypted and forwarded to the simulation software.

There are two fundamental measurable characteristics of network performance [72].

- *Bandwidth* – the number of bits that can be transmitted over the network during a specific period of time. Normal measurement is in Bits/s.
- *Delay (or latency)* – the time period it takes for a very small message to be sent over the network. Delay is measured strictly in time, e.g., number of milliseconds.

A network always has these two characteristics, even though they can change over time, depending on routing and traffic load. This variation can lead to another important phenomenon called *jitter*, which is the variation of the delay for different network packets in a data stream. In the rest of the paper B_{WAN} denotes the band-width for the simulation environment and T_{WAN} the delay between the nodes.

Note that in this paper we will discuss two different time scales. The first one is the simulated time corresponding to the modeled physical process. Integration time steps, time stamps sent within the TLM framework, and the T_{TLM} parameter correspond to this time scale. The second time scale belongs to the real-time of the computation of the simulation result. The total simulation time and delays in the network, i.e., T_{WAN} belong to this scale.

When measuring delays in a system, the time is often referred to as the round-trip-time (RTT), which is the time it takes for the packet to travel to the destination and back. The delay of the system can be approximated to RTT/2.
Another central concept is the *delay* $\times$ *bandwidth product*, depicted in Figure 3. The formula is as follows:

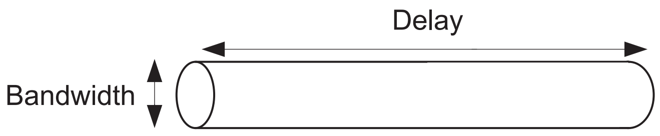


Figure 3: *The delay $\times$ bandwidth product P*

$$P = B_{WAN} \times T_{WAN}$$

The bandwidth can be seen as the diameter of a pipe, and the delay as the length of the pipe. Hence, P represents the maximal natural queue of data located in the network.

Besides the TLM equations and the delay / bandwidth discussion above, there are of course several other factors that can affect the outcome. One such important factor is the load and balanced computation power among the nodes. Another factor is the integration step size / tolerance level used on the simulation nodes. Other potential requirements, such as encryption protocols and algorithms used, can both affect T_{WAN} and B_{WAN}. If the bandwidth is critical, lossless data compression techniques can be used to increase the bandwidth [7]. However, note that delay cannot be improved by data compression.

3 Experimental Setup

This section provides a detailed description of the environments surrounding the experiment, the simulation framework, and software used in the experiment.

3.1 Meta-Models and Components

A *component* is a model instance created in a specialized modeling and simulation tool (e.g., Adams or BEAST). A *meta-model* defines the interconnections between two or more components.

The objective of the experiment is to measure how secure communication over large distances will affect the meta-model simulation in terms of total simulation time. To model the scenario described in Figure 1, a highly simplified model is used instead of a real car model. We are interested in how the data communication factors, T_{WAN} (the delay) and B_{WAN} (the band-width), together with the TLM delay T_{TLM} affect the simulation time.

The meta-model used in the experiment is a double pendulum. As shown in Figure 4, the pendulum is constructed from three different components, two shafts, which both have TLM connection to a bearing.

In terms of simulation time and resources, the two shaft components represent the less demanding components in our meta-model.

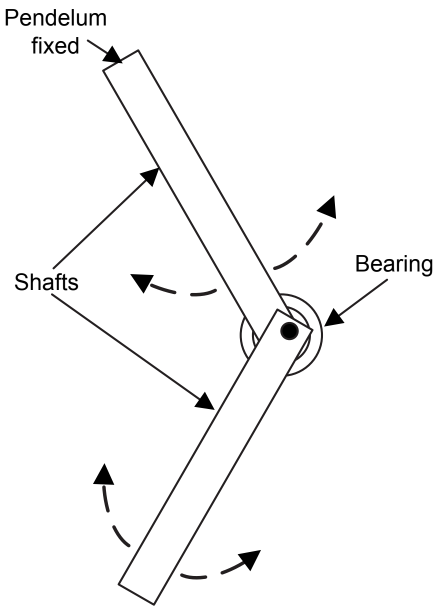


Figure 4: *The bearing-shaft model to be simulated.*

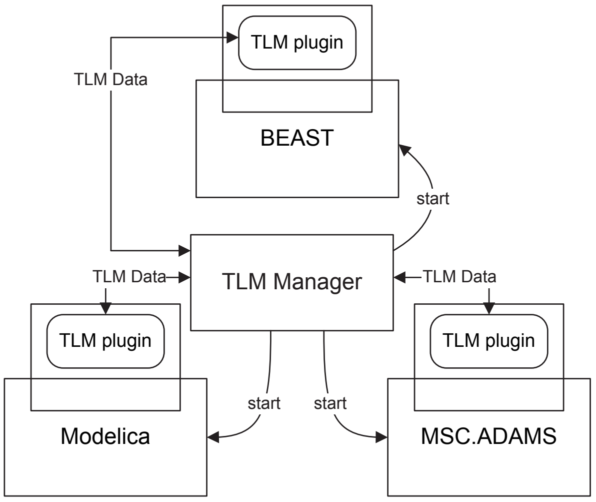


Figure 5: *SKF's TLM system framework.*

3.2 Simulation Framework

To conduct the experiment, we have used a centralized TLM co-simulation application referred to as the TLM manager. Figure 5 presents an overview of the TLM co-simulation framework.

The TLM manager reads a meta-model specification defining the simulation components and the TLM interfaces in the meta-model.

The simulation components are simulated in a specialized simulation environment, such as Adams, BEAST, or Modelica. In order to make the framework functional, the specialized simulation environments need to be incorporated into the framework. The TLM manager and the specialized environment communicate through TLM plug-ins, as depicted in Figure 5.

During simulation, the simulated components interact by sending time-stamped data and delayed position and orientation data. The TLM data is transmitted to relevant node via the TLM manager. For a more detailed description of the framework, see [62].

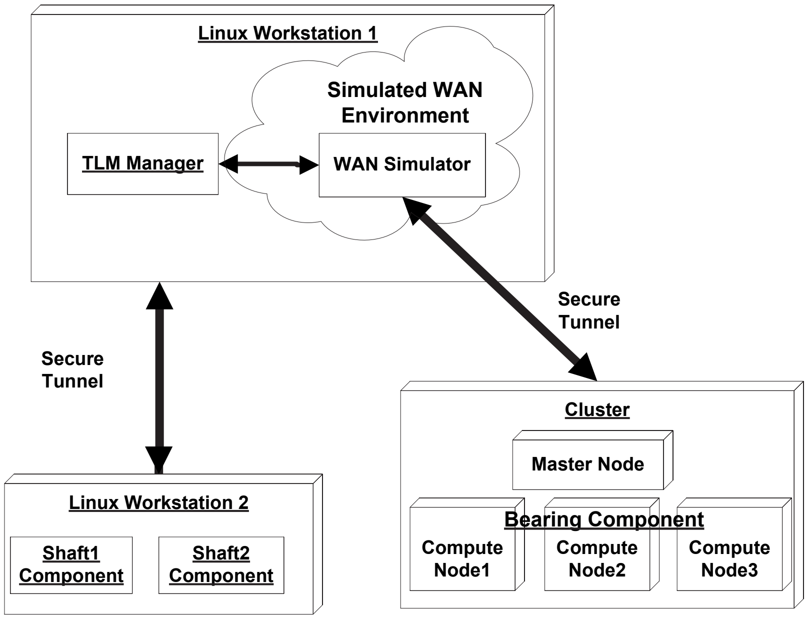


Figure 6: *The deployment structure of the static structure of the system.*

3.3 Deployment Structure

The static deployment structure is depicted in Figure 6. Two Linux workstations and a computer cluster are used for simulation of the components. The simulation is started from the workstation labeled *Linux Workstation 1*; this is also where the TLM-Manager and WAN simulator are instantiated.

In order to evaluate what impact delay and bandwidth in the WAN environment have on the simulation it is desirable to have full control of these parameters. This control is obtained through the WAN simulator. The WAN simulator is a software service that intercepts all data sent between the cluster and Linux Workstation 1. It will then apply the desired data communication factors and pass on the data to the correct destination. A more detailed overview of the WAN simulator will be given in the end of this section.

The shaft components are simulated using BEAST. These simulations are run on the workstation labeled Linux Workstation 2. The bearing component is simulated on the cluster using BEAST as well. The two tunnels are ensuring secure communication between the TLM manager and Linux Workstation 2 and between the TLM manager and the cluster. In the experimental setup SSH version 2 (SSHv2) [51] is used to create the secure tunnels. Another strategy would be to incorporate the security layer within the application. With such an approach the transport layer security (TLS) protocol [21] may be a sound alterative.

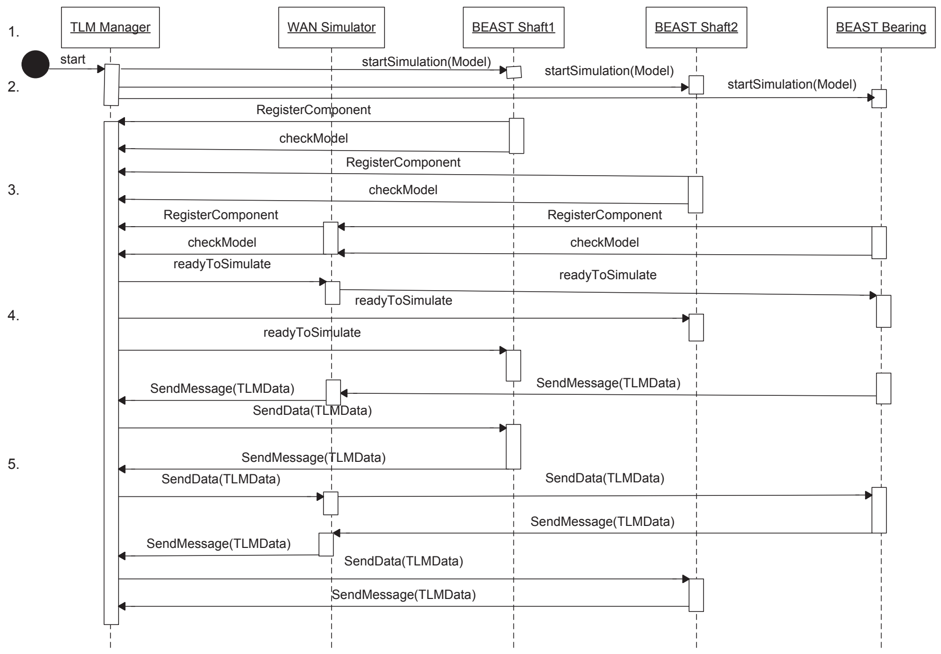


Figure 7: *Sequence diagram showing basic information flow in the co-simulation environment.*

3.4 Dynamic System Behavior

This section describes the interaction between the simulation components and the TLM manager. The sequence diagram in Figure 7 can be described as follows:

1. There are five different objects involved in the interaction during the simulation. To the right, there are the three simulation components, *BEAST Shaft1*, *BEAST Shaft2*, and *BEAST bearing*. The *TLM manager* is handling the data exchange between the components. Finally, the *WAN simulator* is intercepting and forwarding all calls between the bearing simulation component and the TLM manager.

2. The meta-model simulation is initiated by a script. The script will start the TLM manager on the local machine. It will also start the specialized simulation environments of all the simulation components.

3. Each simulation component has to register itself to the TLM manager. The component will also register all of its TLM interfaces. Once this has been accepted by the manager the simulation component will send a check model request to the manager. This is a request asking if the entire meta-model is ready to be simulated.

4. Once all simulation components and TLM interfaces are accounted for, the manager will reply to the simulation components that the meta-model is ready to be

simulated. At this point the actual simulation will begin.

5. The simulation components will regularly send data for its TLM interfaces to the manager. The manager will then pass the data to the appropriate destinations. At the bottom of the diagram we can see how the bearing component through the TLM manager sends some TLM data directed to the shaft1 component. The WAN simulator intercepts this data, holds it for a while, and then forwards it to the TLM manager. The manager forwards the data to the shaft1 component that after some simulation time sends new data to the manager. This communication is not intercepted by the WAN simulator since the data is not passing through our simulated WAN environment. The manager will receive the new data from shaft1 and forward it to the bearing component. This message, from the manager to the bearing, will once again be intercepted by the WAN simulator, and then forwarded to the bearing component. This type of data exchange will continue until the simulation has finished.

For a more detailed specification of the interaction in the communication protocol, see [62].

3.5 WAN Simulator

The WAN simulator is a specific application designed for this experiment with the purpose of controlling the data communication (T_{WAN} and B_{WAN}) between the bearing simulation component and the TLM manager. The simulator captures all data sent between the TLM manager and the bearing component. The simulator consists of two queues, as depicted in Figure 8. One queue holds the data sent from the simulation component to the manager and one queue holds the data sent from the manager to the simulation component. The WAN simulator reads and writes data in the queues concurrently.

To control the delay, the WAN simulator will, when capturing a data message, put a timestamp on it, before it is added to the queue. This data message will not be released from the queue until it has been held for the entire duration of the delay time.

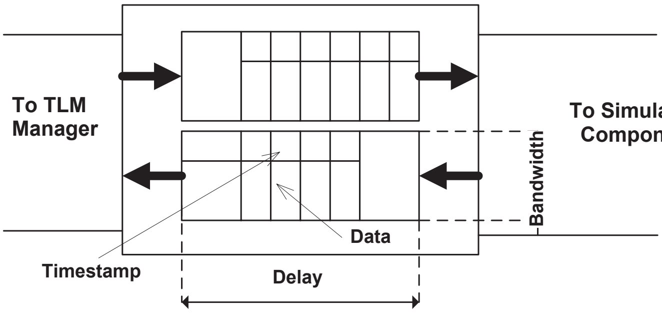


Figure 8: *Conceptual outline of the WAN simulator.*

For example, the simulation component sends some data to the TLM manager. The WAN simulator captures it and stamps it with the time of the capture. The message together with the timestamp is added to the last position in the queue of data waiting to be forwarded to the TLM manager. When the data reaches the first position in the queue it is next in line to be forwarded to the manager. The WAN simulator will then perform a check of the timestamp on the data. If the condition

$$T_{WAN} \leq (currentTime - timestamp)$$

is true, the data will be released from the queue and forwarded to the TLM manager. Else, the data will be held until enough time has passed to fulfill the condition. The second queue is working in the same way, but in the opposite direction.

Bandwidth may be controlled with the aid of the delay $\times$ bandwidth product, P, described in Section 2. The queues may hold up to P bits of data at any given time without risk of exceeding the desired bandwidth restrictions in respective communication direction. When a queue is filled, further data that wish to pass the WAN simulator will be delayed until data has been released from the queue and space has been made available.

4 Experiment Results and Analysis

In the following section we will present the experimental results produced by performing several test simulations in the experimental setup. The result is followed up by discussions and analysis of how different parameters affect the overall simulation time.

4.1 Experiment Results

The experiment was divided into two parts, where the first part used the experimental setup with WAN simulator according to Section 3. In the second part, co-simulation was performed over the Internet between Sweden and Australia.

Part 1 of the experiment was implemented by performing four different test sequences, where the data communication delay T_{WAN} was varied for each sequence. Figure 9 shows the four sequences for different values of T_{TLM}.

In the first three sequences, 3 computation nodes in the cluster (see Figure 6) were used, while the fourth sequence used only 2 nodes (denoted 3N and 2N in the figure). In addition to the values shown in this diagram, simulations were performed for $T_{WAN} = 500$ ms and $T_{WAN} = 1000$ ms for some of the sequences. These values are not shown in the diagram due to illustration reasons, but will be used in the analysis in the next section.

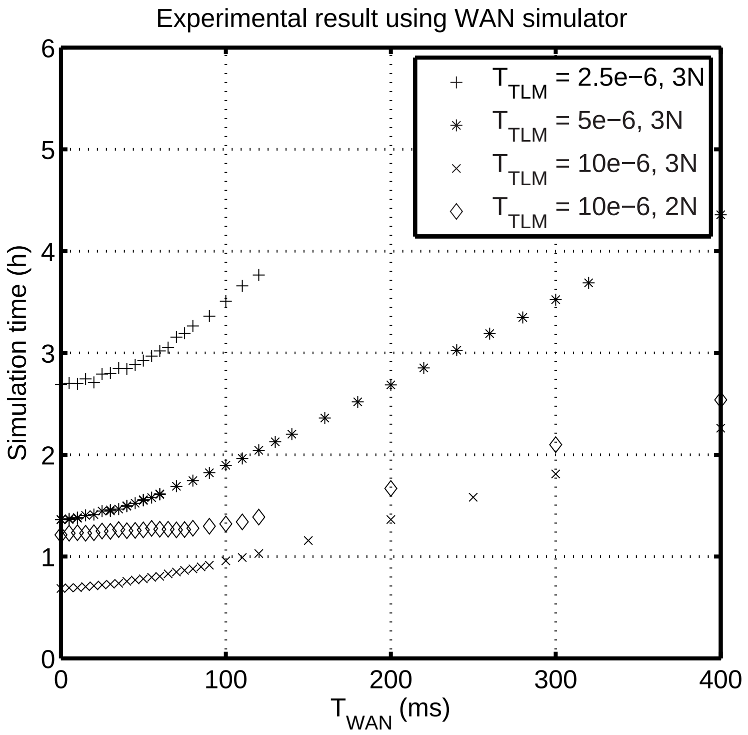


Figure 9: *Plot of total simulation time in relation to the network delay T_{WAN}.*

Part 2 of the experiment was performed without the WAN simulator, i.e., the same setup as depicted in Figure 6 was used, except that the TLM manager communicates directly to the cluster via an encrypted tunnel. Three computation nodes were used in the cluster for these experiments. The results are given in Table 1.

Type of simulation	$T_{TLM} = 2.5e-6$	$T_{TLM} = 5e-6$	$T_{TLM} = 10e-6$
LAN, No Encryption	120 min	62 min	31 min
LAN, Encryption	160 min	82 min	41 min
Increased time due to encryption	33%	32%	32%
Round-trip simulation to University of Queensland, Australia	495 min	277 min	133 min
Increased time due to delays to Australia (round-trip)	312%	344%	332%

Table 1: *Simulation time when executed at the local area network (LAN) in Sweden, both with and without SSH tunnel encryption enabled. Round-trip simulation to and from Australia, including SSH encryption.*

The table shows the increased time needed in percent using an encryption tunnel (SSH) compared to performing the simulation without it.

When executing the simulation via Australia, both the shafts and the bearing were simulated in Sweden, but the traffic between the cluster and the TLM manager was transmitted via a tunnel from Sweden to Australia and then back again. Hence, the experiment is actually performed for a distance twice as long as Australia-Sweden. Note also that the simulation time for just one way cannot be calculated simply by dividing in half.

While performing the simulations, the data throughput in the WAN simulator was being monitored and measured. Since the WAN simulator captures all data between the TLM manager and the bearing simulation component, it was measured that the data throughput did not normally exceed 35 Kbit/s during the TLM data exchange. However, when the simulations were initiated, a small peak in throughput of about 140 Kbit/s was noticed.

In the simulations performed in this experiment, no major robustness problems were discovered and all started simulation jobs were finished without unexpected termination.

All in all, the simulation experiment results presented in this section required approximately 250 hours of simulation time.

4.2 Discussion and Analysis

The main purpose of this work is to investigate if it is possible to co-simulate over long distances and which parameters that affect the total simulation time. In Table 1 it was shown that the simulation was indeed possible to perform, even for very long distances. We will now analyze and discuss the affected parameters and then finally compare the experimental data obtained using the WAN simulator with the simulation times resulting from the simulation via Australia.

Recall the diagram given in Figure 9 in the previous section. For small values of T_{WAN}, a non-linear behavior can be seen, while for larger values the curves have a more linear characteristic.

Consider Figure 10, where the values are plotted for $T_{TLM} = 5e - 6$ and $T_{TLM} = 10e - 6$, where $T_{WAN} \leq 250$ ms.

Linear regression using the least square method is used to adjust a line for $T_{WAN} \geq 80$ ms. Both measured values and the adjusted curves are plotted in the diagram. Note that all measured values above 80 ms are used for finding the curve, including measurements 500 ms and 1000 ms.

In both curves, we can imagine a smooth breakpoint somewhere around 75 ms, where

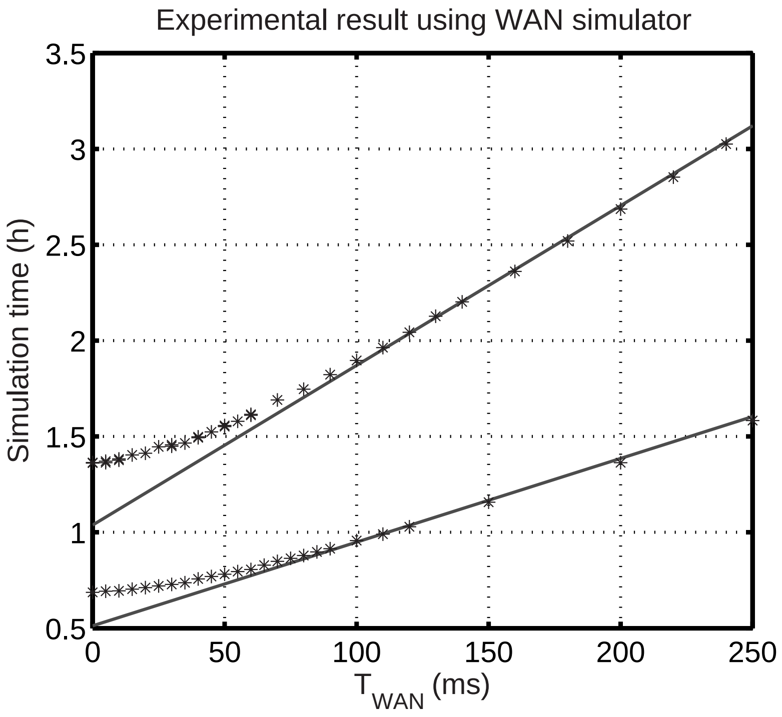


Figure 10: *Simulation time with linear regression estimations.* $T_{TLM} = 5e-6$ *(the upper curve).* $T_{TLM} = 10e-6$ *(the lower curve). Both sequences are using 3 cluster nodes.*

$$f_{2.5e-6,3N}(t) = 7964.4 + 46.8t \quad (1)$$
$$f_{5e-6,3N}(t) = 3735.5 + 30.0t \quad (2)$$
$$f_{10e-6,3N}(t) = 1845.1 + 15.7t \quad (3)$$
$$f_{10e-6,2N}(t) = 2911.0 + 15.9t \quad (4)$$

Figure 11: *Linear approximations after using the least square method on measured simulation data. $f(t)$ states simulation time in seconds and t represents T_{WAN} in ms.*

the derivative becomes fixed. What can cause this breakpoint? One reasonable cause is that to the right of the breakpoint, one of the computation nodes is idle for some time while waiting on TLM data input, i.e., the delay causes the bottleneck of the system to be switched from one of the computation nodes to the delay value in the communication link. For this reason, we call this breakpoint the bottleneck breakpoint, denoted T_{BBP}. Hence, from the data shown so far, we clearly see that T_{WAN}, affects the simulation time significantly, especially above T_{BBP}. The approximated formulas for all experiment sequences are given in Figure 11.

All formulas in Figure 11 were approximated for T_{WAN} larger or equal to 80ms, except for the last case where $T_{WAN} \geq 110$ ms, since the linear characteristic started at larger values. One interesting observation is that changing the computational power for the bearing calculation changes the total simulation time, but not the derivative for the linear part of the curve ($15.7 \approx 15.9$).

Another important parameter in data communication is the bandwidth in the communication link. Does the limitations of the bandwidth affect the total simulation time? The WAN simulator tool has the capability to simulate a smaller bandwidth. However, for this model, we noted that only a small bandwidth of about 35 Kbit/s was needed. Since leased lines normally have the capacity of several MBit/s, this throughput requirement can be negligible. However, for larger models with many TLM-interfaces, the bandwidth factor may be important. To draw any further conclusions about how larger models affect the bandwidth, more experiments must be conducted, which is suggested as future research.

There are of course several other factors that may affect the total simulation time, which we have not measured in this experiment. One such parameter is the time step or tolerance level of the numerical integration routine for the simulation.

In the experiments, SSH was used to protect the TLM data between the nodes. The encryption and hash computations do indeed affect the delay in the system. As we can see in Table 1, the effect is not negligible (around 32% increased simulation time). We have not made any comparisons in this experiment what the effect would be when applying different encryption algorithms, even if there could be expected to be differences. Note however that if the co-simulation is performed on a trusted WAN, e.g., leased private lines, it might be acceptable to disable data encryption to improve performance.

To be able to compare the simulated delays of T_{WAN} with delays in real world networks, a number of measurements were performed over the Internet using the standard ping application. The tests were performed both using a connection at Linköping University (LIU), and a best effort connection from Stockholm, Sweden. Since we know that the simulation time grows linear to T_{WAN}, we can approximate the simulation time for different cities. The measured delays and estimated simulation times are presented in Table 2.

For values below $T_{BBP} \approx 75$ ms, estimations are taken from measured data and not the linear approximation. However, can we really expect that these approximations correspond to real distributed simulation? Since the experiments with the WAN simulator were made in an idealized environment, it can be expected that these results are optimistic.

University, Location	T_{WAN} (ms) Linköping	T_{WAN} (ms) Stockholm	Simulation Time (min)
University of Trier, Germany	21	20	85
University of Cambridge, UK	19	19	85
MIT, Massachusetts, USA	57	56	96
Stanford, California, USA	96	87	108
University of Tokyo, Japan	142	143	134
University of Queensland, Australia	170	178	149

Table 2: *Measured communication delays between Sweden and different locations all over the world. The optimistic approximated simulation times, using formula (2) in Figure 11, are given for* $T_{TLM} = 5e-6$*.*

In the second part of the experiment, as given in Table 1, a real simulation via Australia was performed. A comparison between measured and approximated results is depicted in Table 3.

	T_{WAN}	$T_{TLM} = 2.5e-6$	$T_{TLM} = 5e-6$	$T_{TLM} = 10e-6$
Ping RTT	348 ms	*404 min*	*236 min*	*122 min*
Max, RTT, SSH	470 ms	*499 min*	*297 min*	*154 min*
Min, RTT, SSH	340 ms	*398 min*	*232 min*	*120 min*
Mean, RTT, SSH	405 ms	*473 min*	*265 min*	*137 min*
Measured, Australia	-	495 min	277 min	133 min
Difference Measured to Ping RTT	-	-18.3%	-14.7%	-8.4%
Difference Measured to Mean, RTT, SSH	-	-4.5%	-4.4%	2.8%

Table 3: *A comparison between approximated simulation time and real measured simulation time.*

Note that all times are round-trip-times and not one-way delays. Simulation times given in italic indicate an approximated value, using (1)-(4) in Figure 11. Row 5 corresponds to the experiment in Table 1.

It was discovered that the actual delay during the simulation varied from 340 ms to 470 ms (row 2 and 3), indicating a certain jitter in the communication[3].

In the table we can see that the ping RTT gives an optimistic value, resulting in an 18% to 8% too short simulation time. However, if we calculate the mean (row 4), a closer approximation is achieved.

Using formula (2) in Figure 11, together with the measured simulation time of the round-trip simu-lation in Table 1, we can compute the total simulation time it would take to simulate the shafts in Australia and the bearing in Sweden, i.e., use the delay instead of the RTT. For $T_{WAN} = 5e - 6$, an approximated simulation time is ≈ 170 min; a 174% increase in time compared to an unencrypted local simulation that took 62 minutes.

[3]Note that these delays show the RTT for the TCP packets of the SSH tunnel, which does not include the delay of the actual encryption/decryption.

5 Conclusions

We have in this paper discussed different aspects of secure modeling and simulation. Focus was put on secure distribution and simulation and both a centralized and a decentralized approach were discussed and compared from a security perspective. The decentralized approach was argued to have benefits regarding confidentiality and integrity, while giving open questions about performance and robustness.

An experiment was performed where a simple double pendulum with a bearing was co-simulated between Sweden and Australia. No robustness issue was discovered during simulation, and the simulation time was increased from approximately 62 min to 170 min ($\approx 174\%$ time increase), compared to a local simulation without encryption of data traffic.

To investigate which parameters that affect the total simulation-time, four series of tests were performed using a simulated environment of the wide area network (WAN). The main findings can be summarized as follows:

- The *network bandwidth* between simulation nodes has little effect, since throughput of TLM data is small.
- The *network delay (latency)* between simulation nodes has great impact on the total simulation time. The growth is linear after a certain breakpoint. Consideration must be taken to *jitter* (delay variations over time).
- The *TLM delay* of the model affects the simulation time significantly. Larger delay gives shorter simulation time, since the solver is allowed to take longer time steps.

Finally, we would like to conclude that secure co-simulation over long distances seems to be both a practical and possible solution for secure distribution and simulation of models within, and potentially between, enterprises.

Acknowledgments

This research was funded by CUGS (National Graduate School in Computer Science), by SKF, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

Paper E

Flow Lambda Calculus for Declarative Physical Connection Semantics

Authors: David Broman

Edited version of Technical Reports in Computer and Information Science No. 1.
Linköping University Electronic Press. 2007.

Flow Lambda Calculus for Declarative Physical Connection Semantics

David Broman

Department of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden
E-mail: {davbr}@ida.liu.se

Abstract

One of the most fundamental language constructs of equation-based object-oriented languages is the possibility to state acausal connections, where both potential variables and flow variables exist. Several of the state-of-the art languages in this category are informally specified using natural language. This can make the languages hard to interpret, reason about, and disable the possibility to guarantee the absence of certain errors. In this work, we construct a formal operational small-step semantics based on the lambda-calculus. The calculus is then extended with more convenient modeling capabilities. Examples are given that demonstrate the expressiveness of the language, and some tests are made to verify the correctness of the semantics.

Keywords: Flow connection, Flow Lambda Calculus, Operational Semantics

1 Introduction

Modeling and simulation have been an important application area for several successful programming languages, e.g., Simula [20] and C++ [85]. These languages and other general-purpose languages can be used efficiently for discrete time/event-based simulation, but for continuous-time simulation, other specialized tools such as Simulink [53] are commonly used in industry. The latter supports causal block-oriented modeling, where each block has defined input(s) and output(s). However, during the last decades, a new kind of language has emerged, where differential algebraic equations (DAEs) can describe the continuous-time behaviour of a system. These languages enable modeling of complex physical systems by combining different domains, such as electrical, mechanical, and hydraulic. Examples of such a languages are Modelica [61], Omola [4], gPROMS [6, 68], VHDL-AMS [18], and χ (Chi) [28, 88]. Several of these languages (e.g., Modelica and Omola) support object-oriented concepts, where physical models can be composed and reused. One of the fundamental concepts enabling this composition, is the use of acausal connections between model instances, with the use of *potential* and *flow* variables. These kinds of variables are common in most physical domains and describe the preservation of energy in a system. For example, in the electrical domain, potential variables denote voltage potential and flow variables denote electric current, which obey Kirchhoff's current law, i.e., that the current should sum to zero in a node. As another example, in the rotational mechanical domain, angles are expressed using potential variables and torque is represented using flow variables.

1.1 Motivation and Contribution

Languages of this sort have been developed from an engineering perspective with the focus on numerical solution strategies and run-time semantics for handling mixed discrete / continuous-time (hybrid) systems. Several of these languages have grown to be large and are informally specified using natural language. This can make the languages hard to interpret, maintain, and reason about, which affects both tool development and language evolution. Moreover, the need for static detection and isolation of certain modeling errors is essential for productive modeling and simulation. Such errors can concern over- and under-constrained systems of equations, and consistency checking of physical units and dimensions. Even if current tools support checking for these kind of errors, a formal semantics of the language is needed to be able to develop checking algorithms that *guarantee* the absence of faults.

Hence, there is a concrete need to be able to express the core concepts of such equation-based object-oriented (EOO) languages using formal semantics. We have in this paper developed a novel small-step operational semantics that captures the essential constructs in such languages, including acausal connections, potential and flow-variables, and model abstraction. The semantics is built on the untyped λ-calculus, which is extended with semantics for handling flow-connections.

1.2 Outline

The remainder of this paper is structured as follows. Section 2 gives an informal introduction to acausal physical modeling using the concept of higher-order models and functional abstraction. An example of a simple circuit is modeled and the concept for model reuse and specialization is outlined. Section 3 states the formal abstract syntax and operational semantics of the untyped flow λ-calculus (written $\tilde{\lambda}$). This syntax and semantics forms the basis of the modeling kernel language (MKL), which is presented in Section 4. The additional formal syntax and formal semantic rules are given and syntactic derived forms are described. The language presented in this section is the one used in the modeling examples in Section 2. Section 5 describes the prototype implementation and gives a short evaluation of the language semantics and Section 6 presents related work. Finally, Section 7 states concluding remarks.

2 Informal Language Syntax and Semantics

In this section, an informal introduction to an experimental language called Modeling Kernel Language (MKL) is outlined. The language is not intended to be a full fledged modeling language, but to demonstrate the fundamental modeling possibilities when a equation-based modeling language is based on the lambda calculus.

2.1 A Simple Electrical Circuit

To illustrate the basic modeling capabilities, the simple circuit shown in Figure 1 is to be modeled and simulated.

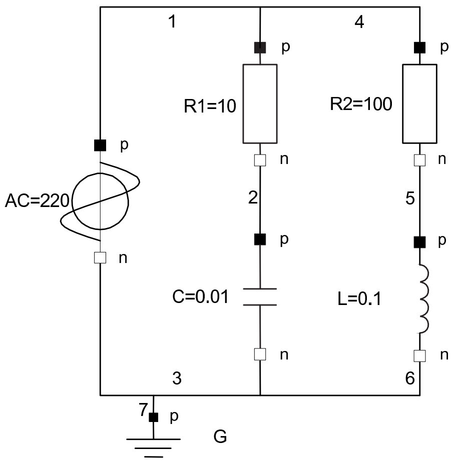


Figure 1: *Graphical outline of a simple electrical circuit.*

The model is described by the following source code:

```
def Circuit = model()
{
    def w1 = Wire();
    def w2 = Wire();
    def w3 = Wire();
    def w4 = Wire();
    Resistor(w1,w2,10);
    Capacitor(w2,w4,0.01);
    Resistor(w1,w3,100);
    Inductor(w3,w4,0.1);
    VSourceAC(w1,w4,220);
    Ground(w4);
};
```

The code shows the definition of a new model called `Circuit`. The model takes zero formal parameters, given by the empty tuple to the right of the model keyword, `model()`. The content of the model is given within curly braces. The first four statements define four new *wires*, e.g., connection points from which the different components (model instances) can be connected.

The six components defined in this circuit correspond to the layout given in Figure 1. Consider the first resistor instantiated using the following:

```
Resistor(w1,w2,10);
```

The two first arguments state that wires `w1` and `w2` will be connected to this resistor. The last argument expresses that the resistance for this instance will be 10 Ohm. Wire `w2` is also given as argument to the capacitor, stating that the first resistor and the capacitor are connected using wire `w2`.

2.2 Connections, Variables, and Flow Nodes

The concept of wire is not built into the language. Instead, it is defined as follows:

```
def Wire = func(){(var(),flow())};
```

Here, a function called `Wire` is defined by using the anonymous function construct `func`. The definition states that function takes an empty tuple `()` as argument and returns the expression within curly braces. In this case, a tuple `(var(),flow())` with two elements is returned. A tuple is expressed as a sequence of terms separated by commas and enclosed in parentheses.

The first element of the defined tuple expresses the creation of a new unknown continuous-time variable using the syntax `var()`. The variable could have been given an initial value, which is used as a start value when solving the differential equation system. For example, creating a variable with initial value 10 can be written using the expression `var(10)`. Variables defined using `var()` correspond to *potential* variables, i.e., the voltage in this example.

The second part of the tuple expresses the current in the wire by using the construct `flow()`, which creates a new flow-node. This construct is the essential part in the semantics presented in coming sections. In this informal introduction, we just accept the fact that Kirchhoff's current law with sum to zero at nodes is managed in a correct way.

In the circuit definition we used the syntax `Wire()`, which means that the empty tuple `()` is supplied as a tuple argument to the function `Wire`. The function call will return the tuple `(var(),flow())`. Hence, the `Wire` definition is used for encapsulating the tuple, allowing the definition to be reused without the need to to restate its definition over and over again.

2.3 Models and Equation Systems

The main model in this example is already given as the `Circuit` model. This model contains instances of other models, such as the `Resistor`. These models are also defined using model definitions. Consider the following two models:

```
def TwoPin = model((pv,pi),(nv,ni),v)
{
    v = pv - nv;
    0 = pi + ni;
};

def Resistor = model(p,n,R)
{
    def (_,pi) = p;
    def v = var();
    TwoPin(p,n,v);
    R*pi=v;
};
```

Models are defined anonymously using the keyword `model` followed by a formal parameter and the model's content stated within curly braces. The formal parameter can be a pattern and *pattern matching* is used for decomposing arguments. Inside the body of the model, definitions, components, and equations can be stated in any order within the same scope.

The general model `TwoPin` is used for defining common behavior of a model with two connection points. `Twopin` is defined using an anonymous model, which here takes one formal parameter. This parameter specifies that the argument must be a 3-tuple with the specified structure, where `pv`, `pi`, `nv`, `ni`, and `v` are pattern variables. Here `pv` means positive voltage, and `ni` negative current. Since the illustrated language is untyped, illegal patterns will be discovered first during run-time.

Both models contain new definitions and equations. The equation `v = pv - nv;` in `TwoPin` states the voltage drop over a component that is an instance of `TwoPin`. The definition of the voltage `v` is given as a formal parameter to `TwoPin`. Note that the direction of the causality of this formal parameter is not defined at modeling time.

The resistor is defined in a similar manner, where the third element `R` of the input parameter is the resistance. The first line `def (_,pi) = p;` is an alternative way of pattern matching where the current `pi` is extracted from `p`. The pattern _ states that the matched value is ignored. The second row defines a new variable `v` for the voltage. This variable is used both as an argument to the instantiation of `TwoPin` and as part of the equation `R*pi=v;` stating Ohm's law. Note that the wires `p` and `n` are connected directly to the `TwoPin` instance.

The capacitor and inductor models are defined as follows:

```
def Capacitor = model(p,n,C)
{
    def (_,pi) = p;
    def v = var(0);
    TwoPin(p,n,v);
    C*der(v) = pi;
};

def Inductor = model(p,n,L)
{
    def (_,pi) = p;
    def v = var(0);
    TwoPin(p,n,v);
    L*der(pi) = v;
};
```

It should be noted here that each of these models contains a differential equation. For example in equation `L*der(pi) = v;`, the `pi` variable is differentiated with respect to time using the built-in `der` operation.

Finally, to make the example complete, the voltage source and the ground are defined as follows:

```
def VSourceAC = model(p,n,VA)
{
    def v = var(0);
    TwoPin(p,n,v);
    def f = 50;
    def PI = 3.14;
    v = VA*sin(2*PI*f*time);
};

def Ground = model((pv,_))
{
    pv = 0;
};
```

An instance of the `Circuit` model can be created in the top-level scope using the following code:

```
Circuit();
```

The resulting simulation result is shown in Figure 2.

2.4 Reuse and Expressiveness using Higher-Order Models

Models are very closely related to anonymous functions. We will see later that the models are in fact encoded as lambda abstractions, with special care taken to flow connections. Since models are first class citizens, reuse and expressive modeling can make use of

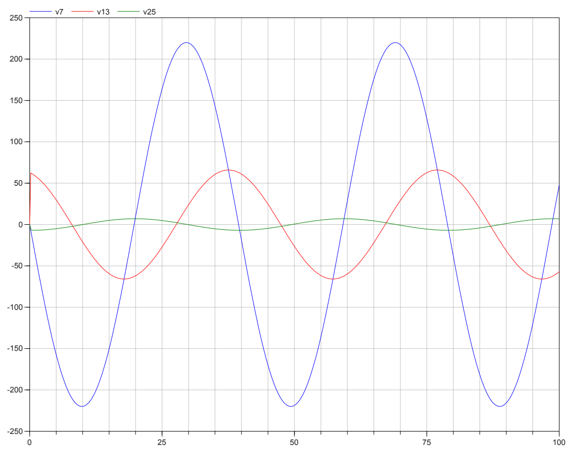


Figure 2: *Plot of simulation result of the simple circuit. The largest curve shows the voltage source, the second largest the voltage drop over the inductor, and the smallest one the voltage drop over the capacitor.*

higher-order models, i.e., models and functions can take models as arguments and return new models.

For example, let us assume that we want to create a new model, which connects two Resistors in parallel. This model can be defined as follows, with resistance values 10 and 100:

```
    def ParallelResistor = model(p,n)
    {
        Resistor(p,n,10);
        Resistor(p,n,100);
    };
```

This simple definition defines a new model named `ParallelResistor`, which composes two resistor instances. Hence, a new model can be defined by reusing other models in an hierarchical structure.

However, can we not generalize this and create a generic way for composing models? Assume that we want to create a model based on composing a few existing models in series. However, we do not want to do it from scratch, e.g., create a model where two resistors are composed in series and then yet another model for an inductor and a capacitor in series. Consider the following function, which takes two models `M1` and `M2` as input, plus an attribute value for the model, such as the resistance or the inductance.

```
def makeSerial = func(M1,val1,M2,val2){
    model(pin,pout){
        def w = Wire();
        M1(pin,w,val1);
        M2(w,pout,val2);
    }
};
```

The function `makeSerial` creates instances of the models `M1` and `M2`, and connects them together using the wire `w`. The left side of `M1` is connected to the new anonymous model's port `pin`, and the second port of `M2` is connected to `pout`. The generic function then returns this new model.

An example where this function is used is given in the following circuit:

```
def Circuit2 = model()
{
    def w1 = Wire();
    def w2 = Wire();
    def ResInd = makeSerial(Resistor, 100, Inductor, 0.1);
    def CapRes = makeSerial(Capacitor, 0.01, Resistor, 200);
    ResInd(w1,w2);
    CapRes(w1,w2);
    Ground(w2);
    VSourceAC(w1,w2,5);
};
```

Here, `makeSerial` defines a new model called `ResInd`, by composing a `Resistor` and an `Inductor`. In the same way model `CapRes` is defined, by composing a `Capacitor` and another `Resistor`.

Note that models can also be parameterized and specialized using traditional concepts in functional programming, e.g., by using currying.

3 Flow Lambda Calculus

In this section, the new connection semantics of flow variables is presented, by extending the untyped lambda calculus with a number of terms, values, and rules. We call this extended version of the lambda-calculus for *flow lambda-calculus*, denoted $\tilde{\lambda}$-calculus.

3.1 Abstract Syntax

Consider the abstract syntax of the $\tilde{\lambda}$-calculus listed in Figure 3. Besides the standard terms lambda abstraction, application, and identifier, a number of terms have been added.

The equation term $t_1 \texttt{=} t_2$ expresses a differential or algebraic equation. The conjunction term $t_1 \wedge t_2$ is used for composing equations into a tree, forming an equation system.

The term `var(`t`)` constructs a new *variable location* (potential variable) in the *variable store*, σ. The creation of flow variables in this store is described in Section 3.2. The store consists of a mapping from a *variable store location* l to a value, $\sigma : \text{VLoc} \rightarrow_{\text{fin}} \text{Value}$. When creating models with systems of equations, variables are often unknown before simulation. An unknown variable is a mapping from a variable store location l to the unknown term ϵ.

The most essential part in this calculus is the definition and treatment of flow nodes together with the flow store. During evaluation, the flow nodes are combined into a tree, which is stored in the flow store ϕ. This tree controls the sum to zero equations that are going to be part of the equation system. The flow store is a finite map from a *flow store location*, f, to a specific node in the tree. Nodes in the tree can be colored to be

r	$\in \mathbb{R}$		Real number
x	$\in$ Ident		Identifier
l	$\in$ VLoc		Variable Store location
f	$\in$ FLoc		Flow Store location
σ	$\in$ VStore = VLoc $\rightarrow_{\text{fin}}$ Value		Variable Store
ϕ	$\in$ FStore = FLoc $\rightarrow_{\text{fin}}$ FNode		Flow Store
n	$\in$ FNode		**Flow nodes**
	$n ::=$		
		$\mathcal{N}_{\mathcal{B}}(t, n)$	Black node
	$\vert$	$\mathcal{N}_{\mathcal{W}}(t, n)$	White node
	$\vert$	$\mathcal{N}_{\mathcal{E}}$	Empty node
t	$\in$ Term		**Terms**
	$t ::=$		
		$\lambda x.t$	Lambda abstraction
	$\vert$	$t_1 t_2$	Application
	$\vert$	r	Real number
	$\vert$	x	Identifier
	$\vert$	t_1=t_2	Equation
	$\vert$	$t_1 \wedge t_2$	Conjunction
	$\vert$	`var`(t)	Variable constructor
	$\vert$	`flow()`	Flow node constructor
	$\vert$	`fork`(t)	Fork connection
	$\vert$	l	Variable Store location
	$\vert$	f	Flow Store location
	$\vert$	ϵ	Unknown
v	$\in$ Value		**Values**
v	$::=$		
		$\lambda x.t \mid r \mid v_1$=$v_2$	
	$\vert$	$v_1 \wedge v_2 \mid l \mid f \mid \epsilon$	

Figure 3: *Abstract Syntax for $\tilde{\lambda}$–calculus.*

either black or white, which is represented in the abstract syntax by terminals $\mathcal{N}_{\mathcal{B}}(t, n)$ and $\mathcal{N}_{\mathcal{W}}(t, n)$. After evaluation, the black nodes represent the sum to zero equations. The white nodes are used during the construct of the tree, but are not representing any equations. Variables, sometimes referred to as *flow variables*, which are used in the sum to zero equations, are created in the variable store σ, and referred to in the flow nodes located in the flow store ϕ.

New nodes are added to the flow store by evaluation of the term `flow()`. Recall the definition of `Wire` in Section 2, which consisted of a tuple with terms `var`(t) and `flow()` as elements. A new flow node is created when this tuple is evaluated.

The last new term is `fork`(t). This is the essential term used for flow connections. It is an internal term, which does not need to be created explicitly by the user of the language. Instead, this term can be hidden and created implicitly by other more convenient constructs. Section 4 describes in more detail how this simplification transformation is performed.

The described $\tilde{\lambda}$-calculus is defined using call-by-value evaluation order. Hence, the set Value $\subseteq$ Term, is used for determining when a term has been evaluated to a value.

3.2 Operational Semantics

The computation rules for the operational semantics are stated in Figure 4, and the congruence rules in Figure 6. The syntax and semantics of the rules are according to standard small-step operational-semantics with premises above the line and the conclusion below. Each rule contains triples, where each triple consist of three elements, separated by bars '|', where the first element is the term, the second the variable store σ, and the last one the flow store, ϕ.

To avoid misinterpretation of the semantics, some notations need clarification. Capture-avoiding substitution is expressed using syntax $[x \mapsto t_1]t_2$, meaning the term obtained by replacing all free occurrences of identifier x in t_2 by t_1. Similar syntax is also used for store updates, where the notation $[f \mapsto n]\phi$ means the resulting flow store that maps f to n together with all other mappings from location to flow node in ϕ. Flow stores are extended using the notation $(\phi, l \mapsto n)$, meaning the flow store ϕ extended with the mapping from l to n, where $l \notin dom(\phi)$. Updates in variable stores are expressed with the corresponding notation, i.e., $(\sigma, l \mapsto v)$. Moreover, in the usual way, rules with more spe-

$$(\lambda x.t)v \mid \sigma \mid \phi \longrightarrow [x \mapsto v]t \mid \sigma \mid \phi \qquad \text{(E-APPABS)}$$

$$\frac{l \notin dom(\sigma)}{\texttt{var(}v\texttt{)} \mid \sigma \mid \phi \longrightarrow l \mid (\sigma, l \mapsto v) \mid \phi} \qquad \text{(E-VAR-CON)}$$

$$\frac{f \notin dom(\phi)}{\texttt{flow()} \mid \sigma \mid \phi \longrightarrow f \mid \sigma \mid (\phi, f \mapsto \mathcal{N}_{\mathcal{B}}(0, \mathcal{N}_{\mathcal{E}}))} \qquad \text{(E-FLOW-CON)}$$

$$\frac{\begin{array}{c} \mathcal{N}_{\mathcal{B}}(t_1, n_2) = \phi(f) \\ l' \notin dom(\sigma) \qquad f' \notin dom(\phi) \\ \phi' = ([f \mapsto \mathcal{N}_{\mathcal{B}}(t_1, \mathcal{N}_{\mathcal{W}}(l', n_2))]\phi, (f' \mapsto \mathcal{N}_{\mathcal{W}}(l', \mathcal{N}_{\mathcal{E}}))) \end{array}}{\texttt{fork(}f\texttt{)} \mid \sigma \mid \phi \longrightarrow f' \mid (\sigma, l' \mapsto \epsilon) \mid \phi'} \qquad \text{(E-FORK-BLACK)}$$

$$\frac{\begin{array}{c} \mathcal{N}_{\mathcal{W}}(t_1, \_) = \phi(f) \\ l' \notin dom(\sigma) \qquad f' \notin dom(\phi) \\ \phi' = ([f \mapsto \mathcal{N}_{\mathcal{B}}(t_1, \mathcal{N}_{\mathcal{W}}(l', \mathcal{N}_{\mathcal{E}}))]\phi, (f' \mapsto \mathcal{N}_{\mathcal{W}}(l', \mathcal{N}_{\mathcal{E}}))) \end{array}}{\texttt{fork(}f\texttt{)} \mid \sigma \mid \phi \longrightarrow f \mid (\sigma, l' \mapsto \epsilon) \mid \phi'} \qquad \text{(E-FORK-WHITE)}$$

$$\frac{v \notin \text{FLoc}}{\texttt{fork(}v\texttt{)} \mid \sigma \mid \phi \longrightarrow v \mid \sigma \mid \phi} \qquad \text{(E-FORK-RM)}$$

Figure 4: *Computation rules of the operational semantics for the $\tilde{\lambda}$–calculus.*

cific terms in patterns are selected first, e.g., for a term t_1t_2, where $t_1 \in$ Value $\subseteq$ Term, rule (E-APP2) in Figure 6 is selected in favor of (E-APP1).

The most interesting rules which differ from standard untyped lambda-calculus are the last four rules in Figure 4. An example of the application of these rules is given in Figure 5. At the first step, a `flow()` term is evaluated using rule (E-FLOW-CON). This rule creates an unused flow location in the flow store ($f \notin dom(\phi)$), maps the location to a new black node, extends the flow store ϕ with this mapping, and returns the new flow store location. The left element in the new black node is a zero value of type real. Figure 5 shows the graph representation of the flow tree. The dashed arrow states that the input flow is zero to the black node. This node corresponds to a sum to zero equation, which has not yet any outgoing flow variables (represented by edges). In the third column in Figure 5, the current state of the flow-store is shown after evaluation of the term in column one. The fact that the black node does not have any outgoing edges is shown with the empty node $\mathcal{N}_\mathcal{E}$ in the second element.

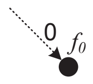
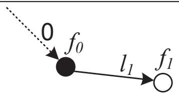
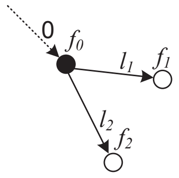
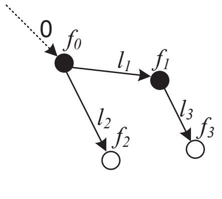
Term	Graph Representation	Flow Store	Var Store
`flow()` $\longrightarrow f_0$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{E}))$	
`fork(`f_0`)` $\longrightarrow f_1$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E})))$ $(f_1 \mapsto \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))$	$(l_1 \mapsto \epsilon)$
`fork(`f_0`)` $\longrightarrow f_2$		$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))))$ $(f_1 \mapsto \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))$ $(f_2 \mapsto \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{E}))$	$(l_1 \mapsto \epsilon)$ $(l_2 \mapsto \epsilon)$
`fork(`f_1`)` $\longrightarrow f_3$	 0 f_0 l_1 f_1 l_2 f_2 l_3 f_3	$(f_0 \mapsto \mathcal{N}_\mathcal{B}(0, \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{W}(l_1, \mathcal{N}_\mathcal{E}))))$ $(f_1 \mapsto \mathcal{N}_\mathcal{B}(l_1, \mathcal{N}_\mathcal{W}(l_3, \mathcal{N}_\mathcal{E})))$ $(f_2 \mapsto \mathcal{N}_\mathcal{W}(l_2, \mathcal{N}_\mathcal{E}))$ $(f_3 \mapsto \mathcal{N}_\mathcal{W}(l_3, \mathcal{N}_\mathcal{E}))$	$(l_1 \mapsto \epsilon)$ $(l_2 \mapsto \epsilon)$ $(l_3 \mapsto \epsilon)$

Figure 5: *Example of the fork command and respresentations in the flow store and the variable store.*

At the second step in the example, node f_0 is forked using rule (E-FORK-BLACK). This rule is chosen in favor of (E-FORK-WHITE), since $\phi(f_0)$ represents in this case a black node. The second and third premise in this rule create both a new variable location (a flow variable) and a new flow store location. As illustrated in the graph representation, a new white node is created. In the forth premise, a new ϕ' is bound, representing the store where location f_0 is updated and a new mapping from f_1 to the new white node is added.

In the third step, f_0 is forked again. In this case another white node is created and an edge is assigned between the black node and the new white node.

Finally, step four forks the node located by f_1. Since this node is a white-node (before evaluation of $\texttt{fork(}f_1\texttt{)}$), rule (E-FORK-WHITE) applies. The main difference in this rule compared to (E-FORK-BLACK) is that the color of the node pointed to by f_1 is changed from white to black. This means that this fork operation both generated a new sum to zero equation (the black node) and added a flow variable l_3.

After evaluation of the given example, the flow store contain four mappings, where two of them maps to black nodes, and two maps to white ones. All locations pointing to a black node will generate a sum to zero equation. The equation is generated by let-

$$\frac{t_1 \mid \sigma \mid \phi \longrightarrow t_1' \mid \sigma' \mid \phi'}{t_1 t_2 \mid \sigma \mid \phi \longrightarrow t_1' t_2 \mid \sigma' \mid \phi'} \quad \text{(E-APP1)}$$

$$\frac{t_2 \mid \sigma \mid \phi \longrightarrow t_2' \mid \sigma' \mid \phi'}{v_1 t_2 \mid \sigma \mid \phi \longrightarrow v_1 t_2' \mid \sigma' \mid \phi'} \quad \text{(E-APP2)}$$

$$\frac{t \mid \sigma \mid \phi \longrightarrow t' \mid \sigma' \mid \phi'}{\texttt{var(}t\texttt{)} \mid \sigma \mid \phi \longrightarrow \texttt{var(}t'\texttt{)} \mid \sigma' \mid \phi'} \quad \text{(E-VAR)}$$

$$\frac{t_1 \mid \sigma \mid \phi \longrightarrow t_1' \mid \sigma' \mid \phi'}{t_1\texttt{=}t_2 \mid \sigma \mid \phi \longrightarrow t_1'\texttt{=}t_2 \mid \sigma' \mid \phi'} \quad \text{(E-EQ1)}$$

$$\frac{t_2 \mid \sigma \mid \phi \longrightarrow t_2' \mid \sigma' \mid \phi'}{v_1\texttt{=}t_2 \mid \sigma \mid \phi \longrightarrow v_1\texttt{=}t_2' \mid \sigma' \mid \phi'} \quad \text{(E-EQ2)}$$

$$\frac{t_1 \mid \sigma \mid \phi \longrightarrow t_1' \mid \sigma' \mid \phi'}{t_1 \wedge t_2 \mid \sigma \mid \phi \longrightarrow t_1' \wedge t_2 \mid \sigma' \mid \phi'} \quad \text{(E-CONJ1)}$$

$$\frac{t_2 \mid \sigma \mid \phi \longrightarrow t_2' \mid \sigma' \mid \phi'}{v_1 \wedge t_2 \mid \sigma \mid \phi \longrightarrow v_1 \wedge t_2' \mid \sigma' \mid \phi'} \quad \text{(E-CONJ2)}$$

$$\frac{t \mid \sigma \mid \phi \longrightarrow t' \mid \sigma' \mid \phi'}{\texttt{fork(}t\texttt{)} \mid \sigma \mid \phi \longrightarrow \texttt{fork(}t'\texttt{)} \mid \sigma' \mid \phi'} \quad \text{(E-FORK)}$$

Figure 6: *Congruence rules of the operational semantics for the $\tilde{\lambda}$–calculus.*

ting the left side of the equation be the first element of the black node, e.g., in node $\mathcal{N}_{\mathcal{B}}(0, \mathcal{N}_{\mathcal{W}}(l_1, \mathcal{N}_{\mathcal{E}}))$, zero will be on the left hand side of the equation. The right hand side consist of the sum of white nodes term values given in element two of the black node. In the example given in Figure 5, the sum to zero equations for the final value of the flow store would be:

$$0 = l_2 + l_1 \tag{1}$$
$$l_1 = l_3 \tag{2}$$

Implicit dereferencing of locations is assumed in the above equations. These equations together with resulting equations after evaluation forms the final equation system. The variables in the equation system correspond to all locations created in the variable store. Note that this store now contains both potential variables created using the term `var(`t`)` and flow variables generated due to forking both black and white nodes.

The congruence rules in Figure 6 are less interesting, but equally important to the semantics. We have chosen to write out all the rules explicitly for completeness, even if there exist simpler and more compact ways of describing these kinds of rules. It should be noted that the congruence rules for equations (E-EQ1) and (E-EQ2), and the rules for conjunction (E-CONJ1) and (E-CONJ2) are stated with two terms to show the evaluation order.

We choose to describe the semantics with small-step-semantics, since it has been shown to exist efficient ways of proving type safety of a language using the progress and preservation theorems [90], if the language is extended with a static type system.

4 Modeling Kernel Language

To enable realistic modeling capabilities, the $\tilde{\lambda}$-calculus needs to be extended with more convenient constructs for modeling. The language presented in this section, called modeling kernel language (MKL) is then used for demonstrating modeling capabilities in Section 2.

4.1 Abstract Syntax

The extra terms and syntactic categories for constructs of the extended language, are listed in Figure 7.

Several of the introduced terms are used for making the language more expressive. For example, a new syntactic category of *patterns* is introduced. The current minimal language supports identifier and tuple patterns, but the language could easily be enriched with other constructs such as records and variants.

Another term for functional abstraction, `func` p $\{t\}$ has been added to distinguish it from the lambda abstraction given in the $\tilde{\lambda}$-calculus. The main difference is that `func` p $\{t\}$ includes a pattern as its formal parameter, while the lambda absraction $\lambda x.t$ used an identifier as formal parameter.

The most important term in the MKL is the `model`-term. The purpose with this term is to create an abstraction mechanism for equation-systems using a functional modeling

bop	$\in$ Bop $= \{+, -, *, /\}$		Binary operations
p	$\in$ Pattern		**Pattern**
	$p ::=$		
		x	Identifier pattern
	$\vert$	`(`$p_i{}^{i\in 1..n}$`)`	Tuple pattern
t	$\in$ Term		**Terms**
	$t ::=$		
		`(`$t_i{}^{i\in 1..n}$`)`	Tuple
	$\vert$	`func` p `{`t`}`	Function abstraction with pattern
	$\vert$	`model` p `{`t`}`	Model abstraction with pattern
	$\vert$	t_1 bop t_2	Binary operations
	$\vert$	`-`t	Uniary negation
	$\vert$	`der(`t`)`	Derivative
	$\vert$	`sin(`t`)`	Sine
	$\vert$	`cos(`t`)`	Cosine
	$\vert$	`time`	Global simulation time
v	$\in$ Value		**Values**
v	$::=$		
		`(`$v_i{}^{i\in 1..n}$`)` $\vert$ `func` p `{`t`}`	
	$\vert$	`-`v $\vert$ `der(`v`)` $\vert$ `sin(`v`)`	
	$\vert$	`cos(`v`)` $\vert$ v_1 bop v_2 $\vert$ `time`	

Figure 7: *Abstract syntax of the kernel language MKL, which represents extensions to the syntax given in Figure 3.*

style, and at the same time hide the existence of the fork semantics, which is needed for correct flow semantics.

The other terms, e.g., binary operations, time derivative operation, Sine function etc., are needed to be able to create relevant models. Some of these terms could also have been implemented as library functions (e.g., Sine and Cosine), but are here part of the language for presentation purpose.

4.2 Operational Semantics

The new evaluation rules for MKL are given in Figure 8. Besides these semantic rules, some syntactic sugar is also added. For example, the `def` construct is transformed into a combination of lambda abstraction and application terms. We will not discuss this syntactic transformation any further, since it is not important in regards to the flow connection semantics.

The functional application rule (E-APPABS-MATCH) states ordinary function application, but with pattern matching. The matching rules are expressed with a separate set of inference rules, where (M-IDENT) is used for identifier patterns and (M-TUPLE) for tuple patterns.

New evaluation rules:

$$(\texttt{func}\ p\ \{t\})v \mid \sigma \mid \phi \longrightarrow match(p,v)t \mid \sigma \mid \phi \qquad \text{(E-APPABS-MATCH)}$$

$$(\texttt{model}\ p\ \{t\})v \mid \sigma \mid \phi \longrightarrow (\texttt{func}\ p\ \{t\})(\texttt{fork(}v\texttt{)}) \mid \sigma \mid \phi \qquad \text{(E-APPMODEL)}$$

$$\texttt{fork((}t^{i\in 1..n}\texttt{))} \mid \sigma \mid \phi \longrightarrow \texttt{(fork(}t_i\texttt{)}^{i\in 1..n}\texttt{)} \mid \sigma \mid \phi \qquad \text{(E-FORKTUPLE)}$$

$$\frac{t_j \mid \sigma \mid \phi \longrightarrow t'_j \mid \sigma' \mid \phi' \qquad t_{tmp} = \texttt{(}{v_i}^{i\in 1..j-1}, t'_j, {t_k}^{k\in j+1..n}\texttt{)}}{\texttt{(}{v_i}^{i\in 1..j-1}, t_j, {t_k}^{k\in j+1..n}\texttt{)} \mid \sigma \mid \phi \longrightarrow t_{tmp} \mid \sigma' \mid \phi'} \qquad \text{(E-TUPLE)}$$

Matching rules:

$$match(x,v) = [x \mapsto v] \qquad \text{(M-IDENT)}$$

$$\frac{\text{for each } i \qquad match(p_i, v_i) = \rho_i}{match(\texttt{(}{p_i}^{i\in 1..n}\texttt{)}, \texttt{(}{v_i}^{i\in 1..n}\texttt{)}) = \rho_1 \circ \cdots \circ \rho_n} \qquad \text{(M-TUPLE)}$$

Figure 8: *Additional semantic rules for the kernel language.*

The most important rule of the new rules is (E-APPMODEL), which matches an application, where the first term is a model. From the definition of `model` p $\{t\}$, we can see that it is almost the same as a functional abstraction, but if we take a closer look at rule (E-APPMODEL), we note that the model is transformed into a function abstraction (a lambda abstraction with pattern), together with a `fork(`v`)` term on the second part of the application term. This construct is the key element of hiding the `fork` construct from the user. The intuition is that each time a connection should be stated between model instances, the wires (connections) need to be forked to form correct flow trees.

Finally, there is one rule (E-FORKTUPLE), which propagates the fork term into a tuple's elements, and a new congruence rule for evaluating a tuple's elements.

5 Prototype Implementation and Evaluation

To evaluate the described language semantics, a prototype implementation was constructed, where the semantic rules were directly translation into OCaml source code. The implementation is not intended for performance evaluation, but to verify the correctness of the given rules.

There are certain properties of the given semantics that we want to prove correct, but this is left to future research. However, it is not obvious how we can prove that it actually models certain properties physically correct in a domain. One alternative would be to prove properties relating to e.g., Modelica and the $\tilde{\lambda}$-calculus. However, since there does not exist any formal semantics of Modelica, which is small enough to reason about, we see this as a difficult strategy to follow.

Instead, the prototype implementation is used for verifying that relevant physical models can indeed be simulated and that they generate approximately the same simulation result. In this prototype implementation, the elaboration procedure transforms a model definition (e.g., the circuit in Section 2) to a flat set of equations. This latter representation can be converted to a flat Modelica file, which we are using for simulating the system. A number of test models were created in both Modelica and in MKL and the simulation result was compared. The purposes of these verification tests are:

- To verify that the prototype can generate equation systems that are solvable.
- To verify that the simulation result correspond to the simulation of equivalent Modelica model.

Tests have been performed on a number of models with positive result. However, it should be noted that the correctness of the current semantics is not verified comprehensively enough. Furthermore, certain proves of correctness must also be conducted in future work.

6 Related Work

The most closely related work to our flow connection semantics is the connection semantics described in the specification of the Modelica language [61]. In Modelica, connections between components (model instances) are declared by using `connect`-equations. For example, consider the following Modelica source code, which expresses the same model `Circuit`, as described in Section 2.

```
model Circuit
  Resistor  R1(R=10);
  Capacitor C(C=0.01);
  Resistor  R2(R=100);
  Inductor  L(L=0.1);
  VsourceAC AC;
  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;
```

From a modeling perspective, connections between components are in Modelica expressed by stating one `connect`-equation between each connector (port). On the contrary, in MKL, a wire is declared, which is then connected by using the name of the wire to express the connection. From a modeling point of view, different users may have different preferences and options on what is simpler and more clear than the other. There are differences regarding modeling capabilities, but it need further analysis to conclude anything about clarity and expressiveness. However, we believe that the $\tilde{\lambda}$-calculus semantics is cleaner due to its declarative nature, which enables better ability to reason about the semantics.

Currently, it does not exist any clean small formal semantics of the Modelica language. There exist specification attempts to specify the whole language using natural semantics [47, 48]. However, this resulted in a very large formal specification, which was very hard to reason about.

Other hybrid languages, such as χ has formal operational semantics defined [88]. However, until this date, the χ language do not yet support the concept of flow connections.

A similar idea of using functional abstraction for modeling of acausal physical models were outlined by Nilsson et. al. [64]. This paradigm, which they call *functional hybrid modeling (FHM)* introduces the concept of *first-class relations on signals and switch constructs*. The signal relations `sigrel` used in the examples in the article have similarities with our model notation, but since the work by Nilsson et.al [64] does not contain any formal semantics, it is hard to analyze the exact similarities. One major difference is that Nilssons et. al.'s work does not incorporate the flow connection semantics into the semantic framework.

To the best of our knowledge, there are no previous published work of a formal semantics of encoding the flow connection semantics in the lambda calculus.

7 Conclusions

We have in this paper described a novel approach of encoding the physical flow connection semantics into the untyped lambda-calculus, using small-step operational semantics. A minimal calculus, called *flow lambda calculus*, denoted $\tilde{\lambda}$-calculus was defined. Based on this calculus, the syntax and semantics was extended to give better modeling capabilities. This language, called modeling kernel language (MKL), was demonstrated with a couple of examples. A prototype implementation of the language was implemented as an interpreter, and some models were simulated and compared with models created in the Modelica language.

Acknowledgments

I would like to thank Peter Fritzson and Björn Lisper for many helpful and constructive comments on drafts of this paper.

This research work was funded by CUGS (the National Graduate School in Computer Science, Sweden), by SSF under the VISIMOD II project, and by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag, New York, USA, 1996.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2nd edition.

[3] Eric Allen, David Chase, Victor Luchangco, Jan-Willem Maessen, and Jr. Guy L. Steele. Object-Oriented Units of Measurement. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 384–403, Vancouver, BC, Canada, 2004. ACM Press.

[4] Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, December 1994.

[5] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[6] 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.

[7] David Broman. Lossless data compression - methods for achieving better performance in a Wireless VPN. Master's thesis, Linköping University, 2001. LiTH-ISY-EX-3159.

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

[9] 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.

[10] 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. Linköping University Electronic Press.

[11] 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.

[12] 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.

[13] Peter Bunus and Peter Fritzson. Automated Static Analysis of Equation-Based Components. *SIMULATION*, 80(7–8):321–245, 2004.

[14] Luca Cardelli. Type Systems. In *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, second edition, 2004.

[15] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[16] François E. Cellier. *Continuous System Modeling*. Springer-Verlag, New York, USA, 1991.

[17] 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.

[18] 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.

[19] 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.

[20] Ole-Johan Dahl and Kristen Nygaard. SIMULA: an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[21] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol. Version 1.1, 2006. RFC 4346.

[22] Iain S. Duff. Algorithm 575: Permutations for a Zero-Free Diagonal [F1]. *ACM Transactions on Mathematical Software*, 7(3):387–390, 1981.

[23] Iain. S. Duff and John K. Reid. Algorithm 529: Permutations To Block Triangular Form [F1]. *ACM Transactions on Mathematical Software*, 4(2):189–192, 1978.

[24] Dynasim. Dymola - Dynamic Modeling Laboratory (Dynasim AB). `http://www.dynasim.se/` [Last accessed: November 8, 2007].

[25] Hilding Elmqvist. *A Structured Model Language for Large Continuous Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, May 1978.

[26] 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.

[27] 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.

[28] Georgina Fábián. *A Language and Simulator for Hybrid Systems*. PhD thesis, Institute for Programming research and Algorithmics, Technische Universiteit Eindhoven, Netherlands, Netherlands, 1999.

[29] Dag Fritzson, Jonas Ståhl, and Iakov Nakhimovski. Transmission line co-simulation of rolling bearing applications. In *In Proceedings of the 48th Conference on Simulation and Modelling (SIMS'07)*, Gothenburg, Sweden, 2007. Linköping University Electronic Press.

[30] Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. Wiley-IEEE Press, New York, USA, 2004.

[31] Peter Fritzson. *Developing Efficient Language Implementations from Structural and Natural Semantics - Draft Version 0.97*. 2006. Book draft available from: `http://www.ida.liu.se/~pelab/rml/`.

[32] 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.

[33] 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 *IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany, 2006.

[34] 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*, Hamburg, Germany, 2005.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York, USA, 1995.

[36] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.

[37] IEEE 1706.1 Working Group. *IEEE Std 1076.1-1999, IEEE Standard VHDL Analog and Mixed-Signal Extensions*. IEEE Press, New York, USA, 1999.

[38] Paul N. Hilfinger. An ADA Package for Dimensional Analysis. *ACM Transactions on Programming Languages and Systems*, 10(2):189–203, 1988.

[39] Iain S. Duff. On Algorithms for Obtaining a Maximum Transversal. *ACM Transactions on Mathematical Software*, 7(3):315–330, 1981.

[40] 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.

[41] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[42] ISO/IEC. *ISO/ IEC 14882 : Programming language C++*. ANSI, New York, USA, 1998.

[43] ITI. SimulationX. `http://www.iti.de/` [Last accessed: November 8, 2007].

[44] Johan Åkesson. *Languages and Tools for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund Institute of Technology, Sweden, November 2007.

[45] Gilles Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, Passau, Germany, 1987. Springer-Verlag.

[46] Andrew Kennedy. *Programming Languages and Dimensions*. PhD thesis, St. Catharine's College, University of Cambridge, UK, UK, 1996.

[47] David Kågedal. A Natural Semantics specification for the equation-based modeling languge Modelica. Master's thesis, Linköping University, 1998.

[48] David Kågedal and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proceedings of the Summer Computer Simulation Conference*, 1998.

[49] Petter Krus. Modelling of Mechanical Systems Using Rigid Bodies and Transmission Line Joints. *Transactions of ASME. Journal of Dynamic Systems Measurement and Control*, 1999.

[50] Petter Krus, Arne Jansson, Jan-Ove Palmberg, and Kenneth Weddfelt. Distributed Simulation of Hydromechanical Systems, 1990. Presented at Third Bath International Fluid Power Workshop.

[51] S. Lehtinen and C. Lonvick. The Secure Shell (SSH) Protocol Assigned Numbers, 2006. RFC 4250.

[52] MathCore. MathModelica System Designer: Model based design of multi-engineering systems. `http://www.mathcore.com/products/mathmodelica/` [Last accessed: November 8, 2007].

[53] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/` [Last accessed: November 8, 2007].

[54] Jakob Mauss. Modelica Instance Creation. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.

[55] Joaquin Miller and Jishnu Mukerji. MDA Guide Version 1.0.1, 2003. Available from: `http://www.omg.org`.

[56] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[57] Robin Milner, Mads Tofte, Robert Harper, and David MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[58] John C. Mitchell and Krzysztof Apt. *Concepts in Programming Languages*. Cambridge University Press, 2003.

[59] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Version 1*, September 1997. Available from: `http://www.modelica.org`.

[60] 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`.

[61] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Language Specification Version 3.0*, 2007. Available from: `http://www.modelica.org`.

[62] Iakov Nakhimovski. *Contribution to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 2006.

[63] IEEE Standards Information Network. *IEEE 100 The Authoritative Dictionary of IEEE Standards Terms*. IEEE Press, New York, USA, 2000.

[64] 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.

[65] 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, 2007. Linköping University Electronic Press.

[66] Object Management Group. *Unified Modeling Language: Infrastructure version 2.1.1*, February 2007. Available from: `http://www.omg.org`.

[67] Object Management Group. *Unified Modeling Language: Superstructure version 2.1.1*, February 2007. Available from: `http://www.omg.org`.

[68] 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):611–633, 1996.

[69] OpenModelica. Project. `http://www.ida.liu.se/~pelab/modelica/OpenModelica.html` [Last accessed: November 11, 2007].

[70] Terence Parr. ANTLR Parser Generator. `http://www.antlr.org/` [Last accessed: November 8, 2007].

[71] Daniel Persson. Dimensional Analysis and Inference for gPROMS. Master's thesis, Department of Computer Science and Engineering, Mälardalen University, Sweden, 2003.

[72] Larry L. Peterson and Bruce S. Davie. *Computer Networks - A Systems Approach*. Morgan Kaufmann Publishers, USA, second edition, 2000.

[73] Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Department of Computer and Information Science, Linköping University, Sweden, 1995.

[74] 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.

[75] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[76] Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Department of Computer Science, University of Aarhus, 1981.

[77] Mikael Sandberg, Daniel Persson, and Björn Lisper. Automatic Dimensional Consistency Checking for Simulation Specifications. In *SIMS 2003*, September 2003.

[78] Alexander Siemers and Dag Fritzson. A Meta-Modeling Environment for Mechanical System Co-Simulation. In *In Proceedings of the 48th Conference on Simulation and Modelling (SIMS'07)*, Gothenburg, Sweden, 2007. Linköping University Electronic Press.

[79] Alexander Siemers, Iakov Nakhimovski, and Dag Fritzson. Meta-modelling of Mechanical Systems with Transmission Line Joints in Modelica. In *Proceedings of the Fourth International Modelica Conference*, pages 177–182, Hamburg, Germany, 2005.

[80] 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.

[81] MSC Software. MD Adams - overview. `http://www.mscsoftware.com/` [Last accessed: November 8, 2007].

[82] Lars Erik Stacke and Dag Fritzson. Dynamic behaviour of rolling bearings: simulations and experiments. *Proceedings of the Institution of Mechanical Engineers, Part J: Journal of Engineering Tribology*, 215(6):499–508, 2001.

[83] Lars Erik Stacke, Dag Fritzson, and Patrik Nordling. BEAST – a rolling bearing simulation tool. *Proceedings of the Institution of Mechanical Engineers, Part K: Journal of Multi-body Dynamics*, 213(2):63–71, 1999.

[84] Arthur G. Stephenson, Lia S. LaPiana, Daniel R. Mulville, Peter J. Rutledge, Frank H. Bauer, David Folta, Greg A. Dukeman, Robert Sackheim, and Peter Norvig. Mars Climate Orbiter Mishap Investigation Board Phase 1 Report. Technical report, NASA, 1999. Available from: `ftp://ftp.hq.nasa.gov/pub/pao/reports/1999/MCO_report.pdf` [Last accessed: November 8, 2007].

[85] 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.

[86] Don Syme. Proving Java Type Soundness. *Lecture Notes in Computer Science*, 1523:83, 1999.

[87] Zerksis D. Umrigar. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices*, 29(9):135–139, 1994.

[88] 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.

[89] D.A. van Beek and J. E. Rooda. Languages and applications in hybrid modelling and simulation: Positioning of Chi. *Control Engineering Practice*, 8:81–91, 2000.

[90] Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

[91] Dirk Zimmer. Enhancing Modelica towards variable structure systems. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 61–70, Berlin, Germany, 2007. Linköping University Electronic Press.