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

by

David Broman

December 7, 2007
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 Etiopia, 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

I Introduction  

1 Background 3  
1.1 Modeling and Simulation  3  
1.2 Equation-Based Object-Oriented Languages  5  
1.3 Fundamentals of Modelica  7  

2 Problem Area 11  
2.1 Safety Aspects in EOO Languages and Environments  11  
2.2 Security Aspects of Modeling and Simulation  13  

3 Paper Overview 15  
3.1 Research Questions  15  
3.1.1 Semantics of the Modelica Language  15  
3.1.2 Early Detection of Constraint Errors  16  
3.1.3 Formal Operational Semantics of EOO languages  16  
3.1.4 Secure Simulation  16  
3.2 List of Papers  17  
3.3 Research Methods  18  
3.4 Contributions  19  
3.5 Related Work  20  
3.6 Paper Errata  21  

4 Concluding Remarks 23  
4.1 Conclusions  23  
4.2 Future Research  24
Contents

II Papers 27

A Types in the Modelica Language 29
1 Introduction ......................................... 32
2 Types, Subtyping and Inheritance .................. 32
  2.1 Language Safety and Type Systems ................ 33
  2.2 Subtyping ........................................ 35
  2.3 Inheritance ........................................ 35
  2.4 Structural and Nominal Type Systems ............. 37
3 Polymorphism ....................................... 40
  3.1 Subtype Polymorphism ............................. 40
  3.2 Parametric Polymorphism ........................... 41
  3.3 Ad-hoc Polymorphism ............................... 42
4 Modelica Types ..................................... 43
  4.1 Concrete Syntax of Types ........................... 44
  4.2 Prefixes in Types .................................. 47
  4.3 Completeness of the Type Syntax .................. 50
5 Conclusion .......................................... 50

B Determining Over- and Under-Constrained Systems of Equations 53
using Structural Constraint Delta
1 Introduction ......................................... 56
  1.1 Constraint Checking of Separately Compiled Components .. 56
  1.2 Error Detection and Debugging ........................ 56
  1.3 Contributions ....................................... 57
  1.4 Outline ........................................... 57
2 Equation-Based Modeling in Modelica .................. 58
  2.1 Modelica Model of an Electric Circuit ................ 58
  2.2 Connector Classes ................................... 59
  2.3 Base Classes and Inheritance ........................ 59
  2.4 Modification and Redeclaration ....................... 60
  2.5 Acausal Modeling and Dynamic Systems ............... 60
3 The Modelica Compiler ................................ 61
  3.1 Elaboration and Type Checking ....................... 61
  3.2 Symbolic Transformation and Code Generation .......... 62
  3.3 Separate Compilation ................................ 63
  3.4 Concluding Remarks ................................ 64
4 Featherweight Modelica ................................ 65
  4.1 Syntax and Semantics ................................ 65
  4.2 Type-Equivalence and Subtyping ...................... 65
5 The Approach of Structural Constraint Delta ............ 68
  5.1 Algorithms for Computing $C_\Delta$ and $E_\Delta$ .......... 69
  5.2 Extending the Type System with $C_\Delta$ ............... 75
6 Prototype Implementation ............................. 76
  6.1 Constraint Checking of Separately Compiled Components .. 76
  6.2 Error Detection and Debugging ...................... 78
| 2.1  | A Simple Electrical Circuit                        | 127 |
| 2.2  | Connections, Variables, and Flow Nodes             | 128 |
| 2.3  | Models and Equation Systems                        | 129 |
| 2.4  | Reuse and Expressiveness using Higher-Order Models | 130 |
| 3    | Flow Lambda Calculus                               | 132 |
| 3.1  | Abstract Syntax                                   | 132 |
| 3.2  | Operational Semantics                             | 134 |
| 4    | Modeling Kernel Language                           | 137 |
| 4.1  | Abstract Syntax                                   | 137 |
| 4.2  | Operational Semantics                             | 138 |
| 5    | Prototype Implementation and Evaluation            | 139 |
| 6    | Related Work                                       | 140 |
| 7    | Conclusions                                       | 141 |

Bibliography 143
Part I

Introduction
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 introductionary 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 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.

\footnote{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.}

3
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

\[
\begin{align*}
& f(t, \dot{x}(t), x(t), y(t), u(t), p) = 0 \\
& t \quad \text{time} \\
& \dot{x}(t) \quad \text{vector of differentiated state variables} \\
& x(t) \quad \text{vector of state variables} \\
& y(t) \quad \text{vector of algebraic variables} \\
& u(t) \quad \text{vector of input variables} \\
& p \quad \text{vector of parameters and constants}
\end{align*}
\]

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 programming 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]. 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
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.

---

**Figure 1.1: Modelica model of an electrical circuit.**
1.3 Fundamentals of Modelica

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 2. 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.
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

---

**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.
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**


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


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


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


**Paper E: Flow Lambda Calculus for Declarative Physical Connection Semantics**

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.
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 $C^\Delta$. 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 readability 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-safety 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.