Thesis Proposal

Type Safety of Equation-Based Object-Oriented Languages

David Broman

Department of Computer and Information Science
Linköping University, Sweden
davbr@ida.liu.se

Abstract

During the past two decades, a new kind of object-oriented language based on differential-algebraic equations has emerged. Examples of such languages are Modelica, gPROMS, and VHDL-AMS. This kind of language, which we call equation-based object-oriented (EOO), enables new possibilities of modeling and simulation of complex dynamic physical systems. However, the unusual language semantics results in new challenges regarding static detection of model errors. In this thesis, we are investigating the use of type systems for static detection of such errors, as well as defining the language semantics in terms of a small kernel language. The formal semantics of such a kernel language is to be defined and the soundness of the type system is to be proven.

1. Introduction

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 [22], using block diagrams and causal connections have dominated the area for 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 efforts [13]. One such language is *Modelica* [24], which is an attempt to unify concepts and notation from several research projects and industrial initiatives. Other examples are gPROMS [26] and VHDL-AMS [11]. We call this kind of language equation-based object-oriented.

1.1 Outline

The remaining part of the introduction will give a fundamental overview of EOO languages when used for modeling and simulation, which is necessary for the understanding of the research problem outlined in this proposal.

In Section 2 the general problem area is outlined and motivations for its relevance are given. Section 3 elaborates on different approaches to attack the problem area and reviews related work. Section 4 begins by stating our hypotheses on how to approach the overall problem. This is followed by delimitations and more detailed descriptions of the problems implied by the hypotheses. The section ends with expected contributions of our work. Section 5 describes the proposed research method, by stating needed working steps including motivations on why these steps are necessary to justify our hypotheses. Section 6 outlines our plan to complete this thesis and gives the status of the current work, including a short summary of current publications. Finally, Section 7 concludes the proposal.

1.2 What is an EOO Language?

All kinds of object-oriented (OO) languages have some form of *object*. Many mainstream OO languages are class-based (e.g. Java and C++), meaning that an object is created as an instance of a *class*. Nevertheless, there are many concepts related to object-oriented languages and there is no clear consensus what actually defines the core concepts of OO languages [2].

In a traditional OO language, the behavior of an object is implemented using *methods* and data is exchanged between objects using message passing or method invocation. The fundamental difference in what we define to be an EOO language is that the behavior of an object is described by *equations*.

EOO languages are primarily used for capturing the behavior of physical systems, i.e., systems in the real world are modeled using equations grouped into classes and objects. Hence, instead of using the term *program*, we will use the word *model* to describe the executable artifact in an EOO language.

To conclude, we define the concept of EOO language as follows:

Definition 1 (EOO language). *Equation-based Object- Oriented (EOO) languages provide the following fundamental concepts:*

- ***Class*** *- a blueprint for creating objects*
- ***Object*** *- can be created according to classes*
- ***Subtyping*** *- if an object A supports the same interface as an object B, then A can be used in the context where B is expected.*
- ***Inheritance*** *- the ability to define a new class by reusing behavior from another class.*
- ***Equations*** *- defines invariant relation between objects.*

Equations can have several different forms. If a model contains *differential equations*, using time derivatives $\frac{dx}{dt}$, it is said to be a *dynamic model*, which is time-dependent. Conversely, if a model only contains *algebraic equations*, such as $x^2 + y^2 = z^2$, it is a *static model*.

In many domains the resulting equation system contains both ordinary differential equations and algebraic equations. This kind of equation system is called Differential-Algebraic Equation (DAE)-system [20].

1.3 Modeling and Simulation with Modelica

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

The first part of this section describes the most fundamental concepts and constructs in an EOO language when used for M&S. We will primarily use Modelica as the target language for our discussions, since it is an open standard with a growing and active community. However, we believe that the concepts and problems are applicable to other related languages as well.

The second part of this section describes the compilation process, where a model is taken as input and a 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. The model can both have a textual representation (left side) and a graphical representation (right side). Tools, such as Dymola [12] and MathModelica [21], make it possible to modify both these representations concurrently and relatively consistently.

Hierarchical Composition. Instances of classes (in Modelica defined with keyword `model`) can be hierarchical composed. For example in Figure 2, model `Inductor` is defined, while in Figure 1 model `Circuit` holds a element named `L`, which is an instance of class `Inductor`.

Continuous-time vs. Discrete-time. If a model only have variables that evolves 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 [22], 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 [13]. 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 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. 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 the Kirchhoff's current law saying that the current 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 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 effecting the other parts of the model.

```
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;
```

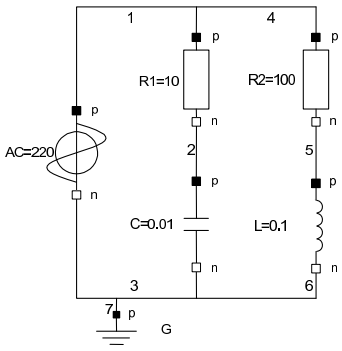

Figure 1. Modelica model of a simple electrical circuit.

```
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 2. Source code of the `Inductor` model and its base class `TwoPin`.

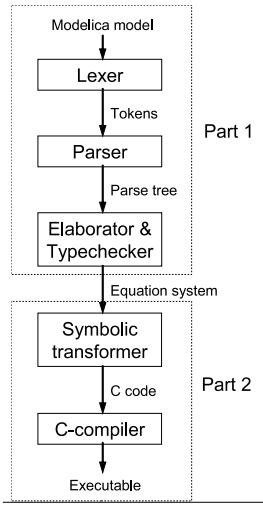

Figure 3. The structure of a Modelica compiler.

Modelica is a large and complex language, consisting of many more constructs, such as inner-outer components, arrays, matrices, expandable connectors etc. For a more comprehensive overview, see [14].

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 3. In the first part, scanning and parsing results in a abstract syntax tree. The Abstract Syntax Tree (AST) is then type-checked and *elaborated* into a *flat system of equations*. This gives us the following definitions:

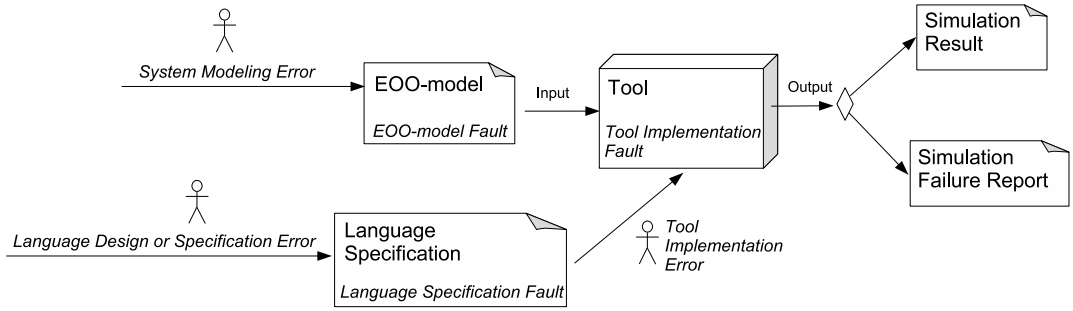

Figure 4. Relations between possible errors and faults in M&S-environment.

Definition 2 (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 (Elaboration). *Elaboration is the task of producing a flat system of equations from the AST of a model.*

In the second part, different symbolic manipulation 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 [29], which is used for solving the equation system. Finally, this executable is executed, which produces the simulation result.

2. Problem Area

EOO modeling and simulation 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, it does not imply that this is the correct result.

We will in the following section outline the overall problems and challenges regarding error handling in EOO languages and its environment.

2.1 Detecting Errors - Isolating Faults

By following the terminology defined in IEEE Standard 100 [25], 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 has gone wrong.

People make mistakes, i.e., commit errors when modeling a system or creating a software. This can result in incorrect simulation results, or even 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 sound 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 produce 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 in an equation system containing 20000 equations, but 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?

2.2 Errors and Faults in an M&S-Environment

There are many different sources of errors in an M&S environment. Consider Figure 4, 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 occur 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 contains a *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 wrong 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 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 [33]

Language Design and Specification Errors

Almost all commonly used languages evolve over time, which results 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 [24] 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.

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 result from a tool, it is obviously even more challenging to detect and isolate the faults during the tool and language development life-cycles.

3. Related Work

Error handling in an M&S environment is a large and complex subject that involves several scientific disciplines, e.g., computer science, scientific computing, electrical and mechanical engineering. In this section, we will briefly survey three areas which are particular interesting from a computer science perspective.

3.1 Precise Language Specifications

There exist several languages which fall into the category of EOO languages used for modeling and simulation of physical systems. Besides Modelica, such languages are e.g. gPROMS [26] and VHDL-AMS [11].

Most of these languages have specifications where the syntax is formally specified using some variant of BNF, while the elaboration and simulation semantics is informally specified using plain text.

This holds also for the Modelica specification, where even the explicit notation of types are indirectly specified, due to the fact that the language make use of a structural type system [4]. Early versions of the language was partially specified using natural semantics in a compiler generation system called Relational Meta Language (RML) [28]. This work [19] has during the last 9 years evolved into the OpenModelica project [15], with the goal to be a complete Modelica compiler, but not a formal specification. Lately, efforts have been made to specify parts of the elaboration process [23] using an algorithmic approach, excluding the type system.

gPROMS [26] does not have any official language specification, since it was commercialized as a product in the year 1997 [31]. However, Barton's PhD thesis [3] describes an early version of the language informally.

VHDL-AMS is designed to be a superset of VHDL, to support simulation of continuous-time and hybrid systems [11]. The language specification, which has an informally described semantics, became an IEEE standard in 1999 [16].

3.2 Well-Constrained Models

A necessary but not sufficient requirement of an EOO model is that it has the same number of unknowns as equation after elaboration. It is trivial to detect this after elaboration, but challenging to isolate in which class the missing or extra equation should be located. Bunus PhD thesis [6] suggests a static debugging approach to this problem, where equations are annotated during elaboration, and then the under-, over-, and well-constrained parts of the equation system are identified. The tool will then gives suggestions of equations that should be added or removed.

3.3 Dimension- and Unit-checking

Dimension and unit checking is far from a new research area. Many library-based approaches exist for imperative programming languages, such as a package approach for Ada [17] and a template approach in C++ [34]. In Kennedy's thesis [18], 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 [1].

Besides the work on gPROMS [27, 32], few attempts has 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.

4. Thesis Statement

In the following section, the research hypotheses of the thesis are given, delimitations are stated, and some of the problems and questions resulting from the hypotheses are discussed.

4.1 Research Hypotheses

Theory about types, type systems, static type checking, and programming languages is a well established and mature research area, where significant results have been established during the last decades [8, 9, 30]. However, this theory has been sparsely used in EOO related languages, especially for detecting and isolating faults in models. This observation leads to our first hypothesis of this thesis proposal.

HYPOTHESIS 1. *By making use of a type system and static type checking, we can detect physical unit and constraint errors in an EOO model at an early stage in the compilation-simulation process, and thus precisely isolate the fault(s).*

Hence, we believe that by making use of types, we can attack the first two challenge areas presented in Section 2.1. However, this still does not give any guarantees for absence of errors. For that reason, we state the following:

HYPOTHESIS 2. *By proving type safety of the EOO language, we can by applying static type checking on a model guarantee that no faults arising from unit or constraint errors exist in the model.*

As stated earlier, Modelica has become a large and complex language, where the semantics is informally described. To be able to prove type soundness (type safety), the language semantics must be specified formally using operational semantics [30, 35]. However, to formally specify the entire Modelica language seems to be a very hard problem.

HYPOTHESIS 3. *By designing a kernel language that models the core concepts of the elaboration phase of Modelica, where the elaboration semantics and the type system are specified formally using operational semantics, it is possible to prove the absence of unit and constraint faults for a model in the kernel language.*

Since the above hypotheses only gives guarantees for models defined in the kernel language, we state the following last hypothesis.

HYPOTHESIS 4. *By transforming a Modelica model into the kernel language (where the utilized constructs in the model can be expressed in the kernel language), we can guarantee the absence of unit and constraint faults in the Modelica model by type-checking the transformed model in the kernel language.*

However, the above hypothesis assumes that the transformation of the model is correct.

HYPOTHESIS 5. *The kernel language is expressive enough to model the essence of the Modelica semantics and still small enough to make it possible to reason about. Furthermore, the transformation rules from Modelica to the kernel language can be concisely and clearly expressed.*

Failure of this last hypothesis does not necessarily mean that the core language is not expressive enough. Instead, we argue that this can show weaknesses and inconsistency in the Modelica language semantics. Hence, our approach can also enable improved language design of the Modelica language.

Due to the wide scope and strong claims of our hypotheses, the following delimitations are given to narrow down the scope of the research proposal for the thesis.

4.2 Delimitations

- Only continuous-time language constructs are considered, i.e., hybrid simulation is excluded from the primary study.
- Only structurally non-singular models can be detected and faults isolated at the model level. Numerically non-singular systems cannot be detected until simulation time.
- Several advanced (and questionable) features of the Modelica language may be excluded from the kernel language, such as expandable connectors.
- Only the semantics of the elaboration phase is intended to be included in the kernel language, i.e. the simulation semantics is excluded.

4.3 Problems and Outstanding Questions

From the hypotheses stated in the previous section, we will here elaborate on the problems implied by the hypotheses.

Defining the Kernel Language

The first main problem of this work is to define a minimal kernel language that is small enough to make it possible to reason about and perform proofs upon, and still expressive and complete enough to capture the essence of Modelica's elaboration semantics.

One design problem is related to the choice of terms in the kernel language. For example, should there exist different terms for modification, redeclaration, inheritance or should it be one construct that models all these concepts? How do we define inner and outer components, that require instance hierarchy scoping, while other constructs are lexically scoped? Can all similar constructs such as `connector`, `model`, `class`, and `record` be unified into one term in the kernel language. Which constructs are part of the type system, and which are part of the untyped language (e.g. `replaceable`)?

Other issues relate to the problem of specifying the semantics using operational semantics, for example, how do we specify the connect semantics with `flow` variables in a consistent way, without the need of too many rules?

Furthermore, it is a challenge to design the static type system so that it is not too conservative, i.e. that the type checker rules out too many legal terms.

There are many questions to be answered and decisions to be taken to be able to model the rather unusual semantics of the elaboration phase in a compact and consistent way.

Extending the Type System to Detect Errors

According to our hypotheses, we intend to extend the types of the kernel language to be able to detect constraint errors, i.e. if the model is over- or under-constrained during type checking. The question is what we should extend the type with? What information is needed to be carried by the types of object and classes? Is it possible to guarantee more than that a model has the same number of equations as unknowns, e.g. to guarantee that a model is structurally non-singular [7]? If we can detect at the class level that a model is for example under-constrained, can we then state where in the model the fault is located?

Much of the theory concerning unit checking is already established, but the difficulty depends on the requirements that are put on the system. Should it be possible to define new units in the language? Do we limit the dimensions to be represented by integer? Should parametric polymorphism be allowed over dimensions and types? Our work in unit and dimensional checking has not yet been started. Hence, these requirements are still to be specified.

4.4 Expected Contributions

The main contributes of this thesis are expected to be:

- A minimal and expressive formally specified kernel language that models the essence of the elaboration process in an EOO language.
- An approach to detect and isolate constraint errors at the class level, without the need for elaboration.
- An industrially suitable solution to guarantee the absence of unit errors in Modelica models.
- The first (to the best of our knowledge) type soundness proof of the elaboration process in an EOO language.
- A better theoretical foundation of the semantics of EOO languages, that aid improved language design of EOO languages in general and Modelica in particular.
- Improve the understanding of types in the Modelica language.

5. Method

In the following section we describe the method used to justify our hypotheses.

5.1 Steps

Our method can be divided into the following five separate steps which are to be iterated until the hypotheses are considered to be reasonable justified.

Step 1 - Define Kernel Language. Develop the minimal and expressive kernel language with a clear separation between the untyped dynamic semantics and the static type system.

Step 2 - Extend the Type System. Extend the type system to handle constraint and unit / dimension errors.

Step 3 - Prove Type Safety. Prove the type safety of the extended language using the progress and preservation theorem [30, 35].

Step 4 - Implement Prototype. Implement a prototype of the kernel language for both the static and dynamic semantics. Implement the transformation from a Modelica model into the kernel language's AST. The output from the prototype should be a flat system of equations that can then be simulated in a standard Modelica tool.

Step 5 - Validate Prototype. This is the final and most critical step to be able to justify our hypotheses. The aim is to select a number of relevant Modelica models from the industry and then inject faults. In the positive test cases, it is checked that:

1. The model passes the kernel language's type checker.
2. The generated flat system of equations is simulated by another Modelica tool and the simulation result is compared to a simulation of the original Modelica model.

In all the negative test cases, the type checker should reject the model and give a response of where the fault is located.

5.2 Method Validation and Criticism

This experimental approach of justifying the hypotheses are naturally very sensitive to the number of test cases, number of test models, and the relevance and quality of the models. Consequently, failure to do these choices right would result in less convincing research results.

However, we claim that the method to justify hypotheses 1 must have a more inductive then deductive characteristic [10]. This due to the fact of the subjective statement "precisely isolate the fault(s)". The soundness of hypothesis 2 and 3 is proven for the kernel language in step 3 the method. Hypothesis 4 follows by proving the soundness of the kernel language. However, the challenge is to not make it too conservative, i.e., that valid Modelica models are rejected.

6. Status and Plan

In the following section, a short summery of the status of current work is given and the plan for future work is outlined.

6.1 Status of Current Work

Our initial work focused on the understanding of the elaboration semantics and the type system of Modelica. It was shown that a clear definition of what a type really is in Modelica was missing in the specification. Hence, this work elaborates on the type concept in Modelica and a concrete syntax of types was proposed [4].

Our second area of work concerned the problem of detecting and isolating constraint faults early in the elaboration phase [5]. The concept, which we call *structural constraint delta*, can determine if a model is under- or over-constrained without the need to elaborate the model to a flat system of equations. In contrary to work by Bunus & Fritzson [7], the isolation of the fault is not performed on the system of equation, but at the type level. The approach makes use of static type checking and consists of a type inference algorithm. A prototype has been implemented for a subset of the Modelica language, and successfully validated on several small examples.

This work was a first attempt to attack the problem of under- and over-constrained system of equations using a type system. However, in contrary to this thesis proposal, no formal semantics was used. Consequently, no type soundness proof was given.

6.2 Plan for Future Work

Currently, our work is focused on defining the untyped part of the kernel language. This should then be extended by a type system followed by proof of its type soundness. Our aim is to present a licentiate thesis containing this work on the kernel language together with the two earlier published papers in the year of 2007.

The next goal is to incorporate unit checking into the kernel language, with a first prototype in the early 2008. The aim is further to incorporate and validate the concept of structural constraint delta in the formal semantics. Finally, the plan is to defend the PhD thesis at the end of 2009.

7. Conclusion

We have described the background and problem area of detecting errors and isolating faults in an equation-based object-oriented language used in a modeling and simulation environment.

The single most important expected contribution of our work would be a minimal and expressive formally specified kernel language that models the essence of the elaboration step in an EOO language. With this language we expect to show that it is possible guarantee that certain kind of faults are always possible to detect and isolate in a model.

We believe that this work will in the future be able to support larger portions of EOO languages in general and the Modelica language in particular, so that ultimately the whole language can be specified and proven for type safety.

References

[1] 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, New York, USA, 2004. ACM Press.

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

[3] Paul Inigo Barton. *The Modelling and Simulation of Combined Discrete/Continuous Processes.* PhD thesis, University of London, 1992.

[4] David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria, 2006.

[5] 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)*, Portland, Oregon, USA, 2006. ACM Press.

[6] Peter Bunus. *Debugging Techniques for Equation-Based Languages.* PhD thesis, Linköping University, 2004.

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

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

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

[10] Alan Chalmers. *What is this thing called Science?* Open University Press, Berkshire, United Kingdom, 1999.

[11] 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.

[12] Dynasim. Dymola - Dynamic Modeling Laboratory with Modelica (Dynasim AB). `http://www.dynasim.se/` [Last accessed: 8 March 2007].

[13] 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.

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

[15] 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.

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

[17] Paul N. Hilfinger. An ADA Package for Dimensional Analysis. *ACM Trans. Program. Lang. Syst.*, 10(2):189–203, 1988.

[18] Andrew Kennedy. *Programming Languages and Dimensions.* PhD thesis, University of Cambridge, 1996.

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

[20] Peter Kunkel and Volker Mehrmann. *Differential-Algebraic Equations Analysis and Numerical Solution.* European Mathematical Society, Züric, Switzerland, 2006.

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

[22] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. `http://www.mathworks.com/products/simulink/` [Last accessed: 6 March 2007].

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

[24] 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` [Last accessed: 29 March 2006].

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

[26] 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.

[27] Daniel Persson. Dimensional Analysis and Inference for gPROMS. Master's thesis, Mälardalen University, 2003.

[28] Mikael Pettersson. *Compiling Natural Semantics.* PhD thesis, Linköping University, 1995.

[29] 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.

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

[31] PSE. E-mail correspondence with Process Systems Enterprise Limited, July 7, 2006. Company webpage: `http://www.psenterprise.com/`.

[32] Mikael Sandberg, Daniel Persson, and Björn Lisper. Automatic Dimensional Consistency Checking for Simulation Specifications. In *SIMS 2003*, page 6, September 2003.

[33] 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: 10 November 2005].

[34] Zerksis D. Umrigar. Fully static dimensional analysis with C++. *SIGPLAN Not.*, 29(9):135–139, 1994.

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