I. Troch, F. Breitenecker, eds. ISBN 978-3-901608-35-3

GROWING AN EQUATION-BASED OBJECT-ORIENTED MODELING LANGUAGE

David Broman
Linköping University, Linköping, Sweden

Corresponding author: David Broman, Department of Computer and Information Science,
Linköping University, SE-581 83 Linköping, Sweden, davbr@ida.liu.se

Abstract. Equation-based object-oriented (EOO) modeling languages are typically rather complex. Such languages can unfortunately not be designed correctly once and for all, not least because all requirements and use cases are not known initially, and may never be known completely. Hence, there is a need to plan for modeling languages to grow in a sound manner. This paper discusses and analyzes how EOO languages in general can be designed for growth, and in particular how this relates to the evolution of the Modelica language. Different ways of growth are categorized and various stakeholders' perspectives are discussed regarding what is important when growing a language.

1 Introduction

General purpose *programming languages*, e.g., C++, Java, and Haskell, can all be used to solve a wide variety of problems. These languages are all Turing complete, i.e., they all possess the same ability to compute the result for a given problem [15]. The main differences between them are their *expressive* power to state problems, their implementation's *performance* of computation, and their ability to guarantee *safe* programs with absence of faults.

Equation-based *modeling languages* (e.g., Modelica [8, 16], gPROMS [1, 19], and VHDL-AMS [6, 11]) are different from such programming languages. Using this kind of modeling language, what is described is not a program, but a mathematical model describing an abstraction of a physical system. The dynamics of the system is normally described using differential algebraic equations (DAEs), which can be hierarchically composed in an object-oriented fashion. In contrast to most programming languages, models expressed in these modeling languages describe the problem (the model), not how it should be solved or analyzed. Instead, the tasks of e.g., simulation and/or optimization are left to the implementation of language translation/execution and/or optimization tools.

In the hypothetical ideal case, a language can be defined once and subsequently for all future fulfill all demands a user might require regarding expressiveness, performance, and safety. Unfortunately, this is never the case. Language theory is one of the core areas within computer science, and history has shown that language design is a very difficult task and that there is no simple solution to design a language that covers all problem domains at once. In a famous speech by Guy L. Steele, he discusses the essence of designing a language for the future [21, 23]:

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

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

This paper discusses and analyzes how EOO languages in general can be designed for growth, and in particular how this relates to the evolution of the Modelica language. This paper does not present any technical contribution, but a systematic categorization of how a language can grow. More precisely, the work is presented as follows:

- Fundamental properties of language theory from computer science are summarized by examples using the Modelica language. The aim is to give the mathematical modeling community tools to understand and discuss properties such as abstract syntax and language semantics (Section 2).
- The design space of how an EOO language can grow is outlined using a new matrix model that categorizes different ways of growth (Section 3).
- The trade-offs for different ways of growth are discussed and analyzed from various stakeholder' perspective (Section 4).

2 The concepts of syntax and semantics

This section briefly outlines the fundamental language concepts of syntax and semantics, by giving concrete examples in the Modelica language.

2.1 Concrete syntax

In a language definition, the concrete syntax describes how *terminals* (e.g., keywords such as `if` and `model`, identifiers, and number literals) are composed. This is described using a *grammar*.

For example, the following Modelica `if`-expression can be *parsed* according to a defined syntax described by a grammar.

```
if x > 1 then x else 0
```

The output from parsing is a *parse tree* that includes all non-terminals and terminals. Figure 1a depicts the parse tree, including all terminal symbols.

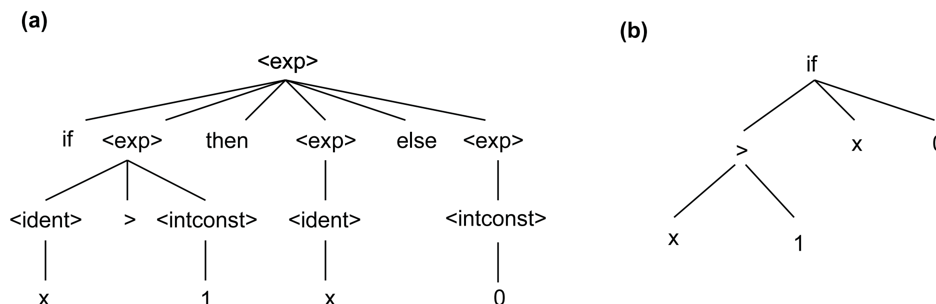

Figure 1: (a) shows the parse tree and (b) the corresponding abstract syntax tree (AST).

2.2 Abstract syntax

Inside a compiler, it is of no relevance to store all terminals within the tree. Hence, the output from the parsing (also called syntactic analysis) is generally an *abstract syntax tree (AST)*. The corresponding AST for the above `if`-expression is given in Figure 1b. Note how some terminals are removed.

2.3 Semantics

While the syntax only describes the structure of the program, the actual *meaning* of the program or model is defined by the *semantics*. When describing how the meaning of a program should be interpreted, the particular keywords or operators symbols are of no importance. For example, whether an `if`-expression has a concrete syntax similar to Pascal

```
if x > 1 then x else 0
```

or to a C conditional statement

```
if(x > 1){return x;} else {return 0;}
```

is unimportant for the semantics. Hence, semantic descriptions should be described based on the abstract syntax and not the concrete syntax. A typical implementation of an EOO language, when used for modeling and simulation, is outlined in Figure 2.

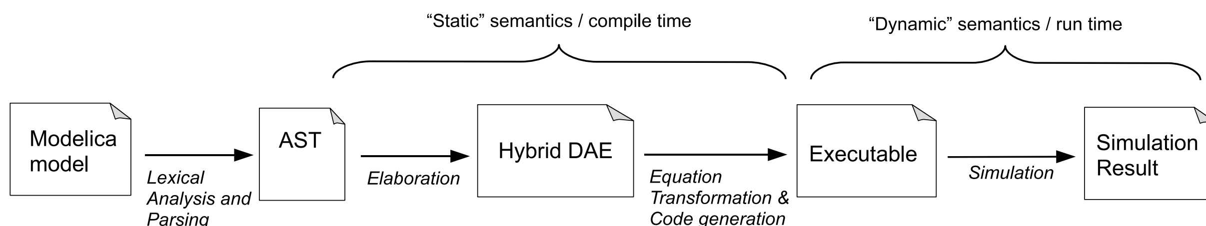

Figure 2: Outline of a typical compilation and simulation process for a Modelica language tool.

In the first phase, a hierarchically composed acausal model is parsed into an abstract syntax tree, which is then *elaborated* (also called flattened or instantiated) into a hybrid DAE, describing both continuous-time behavior (DAEs) and discrete-time behavior (e.g., when-equations). The third phase performs *equation transformations and code generation*, which produces executable target code. When this code is executed, the actual simulation (i.e., execution) of the model takes place, which produces a simulation result. In the most common implementations of the EOO language Modelcia, e.g., Dymola [7] or OpenModelica [9], the first three phases occur during compile time and the simulation can be viewed as the run-time. However, this is not a necessary requirement of EOO

languages in general, especially not if the language supports structurally dynamic systems (e.g., Sol [24], FHM [17], or MOSILAB [18]).

In programming language theory it is common to talk about static and dynamic semantics. Usually, the static semantics is referred to the meaning (transformation, type checking) performed during compile time, and the dynamic semantics the meaning (checking, execution) during run-time. In Modelica, the distinction is not that simple. However, for simplicity, we keep this convention in this paper.

Semantics can be described using formal mathematical methods or by using informal natural language. The former approach is precise and less ambiguous, but complicated to define for large languages. The latter approach is used in the current Modelica specification. See [2] for further discussions of this topic.

3 Different ways of growing a language

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

3.1 The ways of growth matrix

The relationship between syntax and semantics regarding language growth is illustrated in Figure 3.

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

Figure 3: Categorization of different ways of growth depending on whether the language is extended by syntax and/or semantics.

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

3.2 Growth by adding new language features

The most obvious one is given in the upper left corner of Figure 3, i.e., extending both the syntax and the semantics. This is the ordinary way of adding a new language feature, where the new language construct is added to the syntax grammar and the new semantics for this construct is defined.

For example, lookup of variables in Modelica can be according to lexical scope and scope defined by instance hierarchy. The latter was added by defining new syntax where variables could be defined to be `inner` or `outer`. For example, consider Figure 4:

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

Figure 4: Example of the rules for using inner/outer.

Inside model `N` two instances of model `M` are created, namely `m1` and `m2`. Besides the syntactic extension needed for this language feature extension, the meaning of a variable declared as `inner` and `outer` must be defined.

Both the dynamic and the static semantics must be defined. The dynamic semantics can be seen as the meaning of the actual scoping. From the specification [16], the definition is:

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

Hence, `N.x`, `N.m1.x`, and `N.m2.x` are the same variable. The static semantics define the type system, e.g., if `N.x` is an integer but `M.x` is a real, a conflict exists.

3.3 Growth by adding syntactic sugar

Another approach of growing a language is to extend the syntax, but to leave the semantics as it is (the upper right corner of Figure 3). This way of extending a language is often referred to as adding *syntactic sugar*. What does this mean?

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

Let us explain the idea with a concrete example. Consider the four Modelica models `M1a`, `M1b`, `M1c`, and `M1d` given in Figure 5.

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

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

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

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

Figure 5: Four almost identical Modelica models stating a simple initial value problem.

All four models state a simple initial value problem, with a slight difference in their definitions. Are all these models stating exactly the same problem, i.e., are the models's meaning the same?

The simulation result for the first three models are the same, but model `M1d`, does not compile. In model `M1a` and `M1c` the `start` attribute states that the initial condition for `x` is 5 at time 0. In Modelica specification version 2.0, the procedure for specifying initial conditions were changed, and the ability to add a `initial equation` section was added [13]. Examples `M1b` and `M1d` both show that the initial equation `x = 5` is used instead of the `start` attribute. Hence, the same meaning for initial conditions can be specified in different syntactic ways. Would it not be possible to just specify the meaning of one form, and then add the other form as syntactic sugar?

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

Now, consider Figure 6, which shows three potential ways of modeling a steady state initialization:

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

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

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

Figure 6: Steady state initialization modeled in three different ways.

Model `M2c` is using initial equations for modeling the steady state, which is one of the motivations of introducing initial equations. However, as can be seen in the first two models, it can also be modeled by using the start attribute together with a new algebraic variable `dx`. Models `M2b` and `M2c` always give a steady state initialization, but `M2a` depends on if the tool chooses to use the start value of `dx` as initial value (which turned out to not be the case in the tested Dymola [7] environment).

With the current design of Modelica, it is not trivial to define for example the `start` attribute as syntactic sugar of initial equations. However, if it was possible, one can argue that the *meaning* of the start attribute would be easier

to grasp, both for an end user and a compiler engineer. As shown in Figure 6, it is also possible to define steady state initialization, by using the start attribute together with ordinary equations.

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

3.4 Growth by new meanings of annotations or built-in functions

A somewhat more unusual way to grow a language is to extend the semantics *without* changing the syntax (the lower left corner of Figure 3). Hence, this approach changes the meaning of programs without the need to update the grammar for the concrete syntax of the language. How is this possible?

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

Another approach is to have a uniform syntax, where new keywords can be added, i.e., new keywords can be used without altering the grammar. This approach is used in some programming languages inside comments, i.e., data that is normally ignored by the lexical analysis. For example, the javadoc format in Java [10] that gives structured information for automatic code documentation uses ordinary comments (`/** comment */`) with an extra star in the beginning. Hence, there is another syntax defined inside the comment, but this syntax is not changed; only new keywords are added.

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

In the latest specification, a number of annotations are standardized (i.e., the meaning (semantics) of them are specified). However, vendor tools are free to add their own annotations, as long as their names start with the company's name. For example, Figure 7 shows an example of a vendor specific graphical annotation:

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

Figure 7: A vendor specific annotation for a circle [16].

3.5 Growth by new user defined abstractions

The lower right corner of Figure 3 assumes that neither the syntax nor the semantics is extended. How is it then possible to grow the language at all?

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

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

A key point, also emphasized in Steele's speech [21], is that new definitions defined in libraries should look like primitives in the language itself. Hence, in the ideal case, a user of a language should not be able to distinguish if the language has been extended with new functionality via a library definition, or by changes in the language specification. One early programming language that achieved this is LISP [22]. In this language, new definitions defined by users look like primitives and all primitives look like definitions by users. Hence, LISP is a language truly built for growth by its users.

In Modelica, it is possible to define functions which are similar to built-in primitives, e.g., it is possible to define the `sin(x)` function for the real type. However, since the language is still missing capabilities such as operator overloading, function overloading, and higher-order functions [3], many extensions look quite different compared to primitive constructs. For example, if a library is implementing complex numbers or rational numbers without such language constructs, a simple multiplication of two constant complex numbers could look like

```
complexMul(Complex(re=2,im=3),Complex(re=1,im=2));
```

instead of the more natural mathematical notation:

$$(2+3i)*(1,2i) \tag{1}$$

Even though a functional description such as the one above for `complexMul` gives the same semantic result as the one that looks like primitive constructs, it prevents the user from growing the language by himself/herself.

3.6 Restricting the language

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

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

Our previous work on determining over- and under-constrained models by extending the static type system of an EOO language [4] is an example of such an approach. Parts of these ideas have later also been included in the latest Modelica specification [16]. This is one of the major changes in Modelica version 3.0, where Modelica models are required to be balanced, i.e., to have the same number of equations as unknowns. A more detailed overview and rationale of balanced models in Modelica is given in [20].

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

4 The right way to grow

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

4.1 Stakeholders of an object-oriented equation-based modeling language

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

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

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

4.2 Language designers' perspective

Unfortunately, language designers tend to want their language to be able to handle everything. One of their main challenges are not what to put in the language, but what to actually leave out. If a language is designed by one person or a small group, these individual(s) need to judge, test and take all decisions by themselves. This may lead to a concise design, but there is a considerable risk that important input from other stakeholder's, such as end users becomes limited. On the other hand, if the language is designed by a community with a committee, input comes from many sources. However, there is a substantial risk that the different parties involved will lead to many

compromises that can make the language large and complex. The latter approach with a design community and committee is the path that the design of Modelica has followed.

When many parties are involved in the language the risk is that new "features" are continuously added to the specification, i.e., the upper left corner of the matrix in Figure 3 where both the syntax and semantics are changed. However, if fewer people are involved in the process, the language may be designed with a more well-defined core semantics and large parts are defined by using the approach with syntactic sugar. This is the way that for example Standard ML is defined [14]. This way of defining a language is hard and challenging, but can if done right lead to a less ambiguous specification. See [2] for further discussions regarding this topic.

There are some issues that the language designers face that are a bit different for a modeling language such as Modelica, compared to other general purpose programming languages. For example, in most programming languages, functions such as `sin(x)` would not be a built-in function as in Modelica, but a function defined in a library (e.g., in a default library called the preclude in Haskell). However, since the Modelica compiler need to perform symbolic manipulation, e.g., differentiation of the `sin(x)` function, special handling of such functions is important.

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

4.3 End users' perspective

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

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

End users will of course also be able to solve new problems and use existing models in different ways. Even though the Modelica language is primarily designed for simulation, there are several other kinds of analysis that are important, such as applications for automatic control [5] and optimization problems [12].

4.4 Library users' perspective

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

Library users may have conflicting interests with both language designers and tool vendors, since complications and details about the language is not the primary focus for the user. Hence, library users are typically stakeholders who want to continuously expand and add new complex features into the language, so that it becomes more expressive for their needs (adding both new syntax and semantics).

4.5 Tool vendors' perspective

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

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

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

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

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

5 Conclusions

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

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

- *Growth by adding new language features*. Always changing both the syntax and the semantics is the most drastic kind of change of a language and should be minimized or avoided, especially for mature and widely used languages. The stakeholders that are most negatively affected of such changes are language designers and tool vendors, while library users might be the ones that push most for such extensions.
- *Growth by adding syntactic sugar*. Extending only the syntax by using syntactic sugar and at the same time keeping a core semantics is one of the preferable approaches to language growth. It gives both a precise language definition for the tool vendors as well as an understandable language for the user.
- *Growth by new meanings of annotations or built-in functions*. Growth by only changing the semantics and not the syntax might first seem to be a very attractive approach, especially for language designers since few changes are needed in the specification. However, it can also be dangerous, e.g., in cases where many tool dependent annotations might make different tools incompatible.
- *Growth by new user defined abstractions*. Finally, growth by user defined abstractions, i.e., neither the syntax nor the semantics are changed, is the preferable approach in the long term. However, it is far from obvious how to achieve this, especially in such a young language research area as equation-based object-oriented languages.

Acknowledgments

I would like to thank Thomas Schön and Peter Fritzson for many useful comments on this paper. This research was funded by Vinnova under the NETPROG Safe and Secure Modeling and Simulation on the GRID project and by CUGS (National Graduate School in Computer Science).

6 References

[1] P. I. Barton. *The Modelling and Simulation of Combined Discrete/Continuous Processes*. PhD thesis, Department of Chemical Engineering, Imperial Collage of Science, Technology and Medicine, London, UK, 1992.

[2] D. Broman and P. Fritzson. Abstract Syntax Can Make the Definition of Modelica Less Abstract. In *Proceedings of the 1st International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 111–126, Berlin, Germany, 2007. LIU Electronic Press.

[3] D. Broman and P. Fritzson. Higher-Order Acausal Models. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 59–69, Paphos, Cyprus, 2008. LIU Electronic Press.

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

[5] F. Casella, F. Donida, and M. Lovera. Beyond Simulation: Computer Aided Control System Design Using Equation-Based Object Oriented Modelling for the Next Decade. In *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools*, pages 35–45, Paphos, Cyprus, 2008. LIU Electronic Press.

[6] E. Christen and K. 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.

[7] Dynasim. Dymola - Dynamic Modeling Laboratory (Dynasim AB). `http://www.dynasim.se/` [Last accessed: April 30, 2008].

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

[9] P. Fritzson, P. Aronsson, A. Pop, H. Lundvall, K. Nyström, L. Saldamli, D. Broman, and A. 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. See also the OpenModelica Project. www.openmodelica.org [Last accessed: Dec 30, 2008].

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.

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

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

[13] S. E. Mattsson, H. Elmqvist, M. Otter, and H. Olsson. Initialization of Hybrid Differential-Algebraic Equations in Modelica 2.0. In *Proceedings of the 2nd International Modelica Conference*, pages 9–15, Oberpfaffenhofen, Germany, 2003.

[14] R. Milner, M. Tofte, R. Harper, and D. MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.

[15] J. C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.

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

[17] H. Nilsson, J. Peterson, and P. Hudak. Functional Hybrid Modeling. In *Practical Aspects of Declarative Languages : 5th International Symposium, PADL 2003*, volume 2562 of *LNCS*, pages 376–390, New Orleans, Lousiana, USA, Jan. 2003. Springer-Verlag.

[18] C. Nytsch-Geusen et. al. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.

[19] M. Oh and C. C. Pantelides. A modelling and Simulation Language for Combined Lumped and Distributed Parameter Systems. *Computers and Chemical Engineering*, 20(6–7):611–633, 1996.

[20] H. Olsson, M. Otter, S. E. Mattsson, and H. Elmqvist. Balanced Models in Modelica 3.0 for Increased Model Quality. In *Proceedings of the 6th International Modelica Conference*, pages 21–33, Bielefeld, Germany, 2008.

[21] G. L. Steele. Growing a Language. Videotape (54 minutes) of a talk at the *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, University Video Communications, 1998.

[22] G. L. Steele. *Common LISP. The Language*. Digital Press, 2nd edition, 1990.

[23] G. L. Steele. Growing a Language. *Higher-Order and Symbolic Computation*, 12:221–236, 1999.

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