Abstract Syntax Can Make the Definition of Modelica Less Abstract

David Broman and Peter Fritzson

Department of Computer and Information Science
Linköping University, Sweden

{davbr,petfr}@ida.liu.se

Abstract. Modelica is an open standardized language used for modeling and simulation of complex physical systems. The language specification defines a formal concrete syntax, but the semantics is informally described using natural language. The latter makes the language hard to interpret, maintain and reason about, which affect both tool development and language evolution. Even if a completely formal semantics of the Modelica language can be seen as a natural goal, it is a well-known fact that defining understandable and concise formal semantics specifications for large and complex languages is a very hard problem. In this paper, we will discuss different aspects of formulating a Modelica specification; both in terms of *what* should be specified and *how* it can be done. Moreover, we will further argue that a "middle-way" strategy can make the specification both clearer and easier to reason about. A proposal is outlined, where the current informally specified semantics is complemented with several grammars, specifying intermediate representations of abstract syntax. We believe that this kind of evolutionary strategy is easier to gain acceptance for, and is more realistic in the short-term, than a revolutionary approach of using a fully formal semantics definition of the language.

1 Introduction

Modelica is an open standard language aimed primarily at modeling and simulation of complex physical systems. The first language specification 1.0 [19] was released in September 1997. Since then, the current specification 2.2[20] has evolved to be large and complex with many constructs.

During these past ten years, the user community has grown fairly large and the Modelica Standard Library has evolved to include several physical domains. The dominating Modelica tool has for a long time been the commercial tool Dymola [4]. However, during recent years, alternative tools have emerged; both open source (OpenModelica [7, 21]) and commercial environments (e.g., MathModelica System Designer [16], MOSILAB [5], and SimulationX[12]).

The rapidly growing user community and increasing number of tool vendors augment the demand of the language specification being precise so that different

tools will be compatible. Hence, the Modelica Association, who is responsible for the language specification, has defined the goals for the next language version both to make the specification clearer and to simplify the language itself.

1.1 Specification of the Modelica Simulation process

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

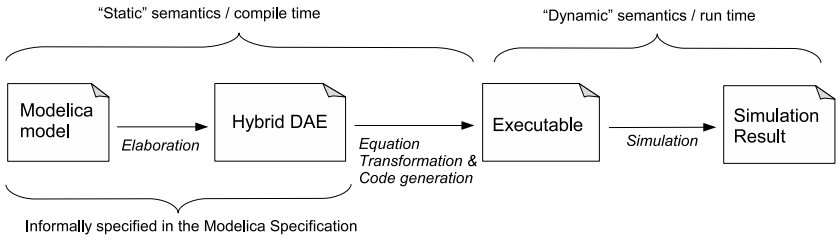

Fig. 1. Overview of a typical Modelica compilation and simulation process.

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

In the current specification 2.2 [20], the concrete syntax is stated formally using Extended Backus-Naur From (EBNF), but only the semantics of the first part of the process is informally described using natural language backed up with concrete source code examples.

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

1.2 Unambiguous and Understandable Language Specification

The natural goal of a language specification is to be *unambiguous*, so that tool implementors interpret the specification in exactly the same way. At the same time, it is important that the specification is *easy to understand* for the intended

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

audience. Unfortunately, it is not that easy to meet both of these goals when describing a large and complex modeling language such as Modelica. There are several specification approaches with different pros and cons. Hence, the overall problem is to find an approach that satisfies the specification goals in the best possible way.

If the language is described using *formal semantics*, e.g., structured operational semantics [25], the language semantics is precise and can in some cases be proved to have certain properties, such as type safety [24, 26]. However, to understand and interpret a formal language specification require a rigorous theoretical computer science knowledge. Furthermore, even if great effort has been spent during the last decades in formalizing mainstream programing languages, only a few, e.g., Standard ML [18], are actually fully formally specified. Accordingly, it turns out to be a very hard task to specify an understandable and concise formal specification of an existing complex language.

Alternatively, if the language semantics is described using *natural languages*, e.g., plain English text describing the semantics, it might be easy for software engineers to understand the specification. Many languages are described in this way, for example Java [9], C++ [11], and Modelica [20]. However, ease of understanding does not imply that different individuals interpret the specification in the same way. It is a well known fact that it is very hard to write unambiguous natural language specifications, and perhaps even harder to verify their consistency.

1.3 Previous Specification Attempts

Several previous attempts have been made to formalize and improve the specification of the Modelica language. The most obvious one is the further development of the official language specification itself, conducted by the Modelica Association. The work on the next language specification includes substantial restructuring and a more detailed description of the semantics of the language. However, it is not planned to include any formal descriptions, apart from an appendix containing one possible definition of Modelica abstract syntax.

Natural Semantics. Already in 1998 Kågedal and Fritzson [14, 15], created a formal specification for a subset of the Modelica language, influenced by the language specification examples described in the 1997 version of [6]. The specification was using *Natural Semantics* [13] and the executable specification language Relational Meta Language (RML) [22]. This work influenced the design of the language and the official Modelica specification. The executable specification has gradually evolved and is now the code basis for the OpenModelica project[21]. In 2006, the code base was converted from RML to Meta-Modelica[8] with the purpose of making it more accessible for software engineers in the Modelica community. Hence, today the project is more intended to be a complete implementation of the language than a specification itself. One lesson learned from this specification project was that for an almost complete specification of

an early Modelica language version, the formal specification became hard to get an overview of, since it grew to be very large.

Elaboration. Jakob Mauss has made several contributions to formally describe the elaboration process (called *instance creation* in his work) of a subset of Modelica, i.e., the translation process from a Modelica model into a system of equations. The published work [17] describes an algorithmic specification approach, which focuses on Modelica's complex lookup rules and modification semantics; including redeclaration of classes and components. Semantics for describing restrictions on validity of a model, such as types, restricted classes, and most prefixes are not considered. It exists also a refined version of this work, which uses a more compact notation. However, this work is still unpublished.

Modelica Types. In our previous work on types in the Modelica language[2], we concluded that the type concept is only implicitly defined in the Modelica language specification. In that work, we proposed a concrete syntax of specifying Modelica types and gave a suggestion for constraining information of element prefixes in the types. Furthermore, it was emphasized that Modelica has a *structural type system*, which implies that a *class* and a *type* are two separate language concepts. In this paper, we will not cover types, even though parts of a specification can also be described using type rules.

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

1.4 Abstract Syntax as a Middle-Way Strategy

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

When a compiler parses a model, the result is normally stored internally as an *Abstract Syntax Tree (AST)*. Hence, one particular model results in a specific AST, which can be seen as an instance of the language's abstract syntax. The abstract syntax can be specified using a *context-free grammar*, and an AST can also have a corresponding textual representation.

The internal representation of an AST is often seen as a tool implementation issue, and not as something that is defined in a language specification. Nevertheless, in this paper we propose that the intermediate representations between the transformation steps (recall Fig. 1) should be described by specifying its abstract syntax.

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

By following this *evolutionary* strategy, the semantic description may then be gradually more described using techniques such as Syntax-Directed Translation Schemes (SDT)[1] or different forms of operational semantics. However, as earlier described, this is not straight forward when considering the whole Modelica language. The main purposes of including abstract syntax definitions in the specification can be summarized to be:

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

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

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

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

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

2 Specifying the Modelica Specification

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

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

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

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

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

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

In the introduction section of the Modelica specification 2.2 [20], it is stated that the scope of the specification is to define the semantics of the translation to a flat Hybrid DAE and that it does not define the result of a simulation. A mathematical notation of the hybrid DAE is given, but no precise and complete output is defined.

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

So, the questions arise: what is actually the transformation process? What is the expected result of the execution? We would argue that the answer to these questions would differ depending on who you ask, since the current specification is open for interpretation. In this subsection, we give our view of a typical Modelica transformation process.

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

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

Equation Transformation and Code Generation The Hybrid DAE is simplified and transformed (index reduction, generation of Block Lower Triangular form (BLT)). Finally, target code is generated (typically C-code), which is linked together with a numerical solver, such as DASSL[23].

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

Static vs. Dynamic. In the example above, it was assumed that the process was *compiled* and not *interpreted*. This is not a specification requirement, even if it is common that tools are implemented as compilers. The definitions of static and dynamic semantics are often confusing in relation to compile-time and simulation-time. Some people will argue that the dynamic semantics is only the simulation sub-process and that the elaboration and equation transformation as well as the code generation phases are the static semantics. If the tool is implemented as an interpreter, the distinction becomes less clear. In such a case, it is natural to view all three processes as the dynamic semantics. Even if this is only a matter of definitions, it becomes significantly important when reasoning about type checking and separate compilation.

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

2.2 Rejection Aspects - *What* is actually a Valid Modelica Model?

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

Besides specifying the translation semantics of a model, a language specification typically describes which models that should be treated as valid, and which

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

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

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

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

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

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

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

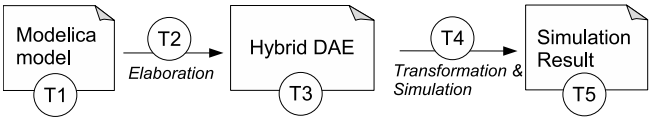

Fig. 2. Possible checking-time during the process

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

The checking at point `T4` takes place during simulation of the model. This is what many would refer to as *dynamic checking*, since it is performed during runtime. Errors which can occur here are for example numerical singularities after events or array out-of-bound errors. Since Modelica does not have an exception

handling mechanism, it is implicitly assumed that the tool exits with an error statement. Checking point T3 is performed after the elaboration phase. This can for example concern the control that the number of equations equals the number of unknowns.

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

Following this argumentation, then T1 would represent *static type checking*, i.e., the types in the language are checked *before* elaboration. This reasoning is analogous to dynamic checking in languages such as PHP and Common LISP, compared to static type checking in Haskell, Standard ML, or Java. Even if the Modelica specification does not currently support this kind of static checking, it has a major impact on the ability to detect and isolate for example over- and under-constrained systems of equations[3] or to enable separate compilation.

2.3 Specification Approaches - *How* can we state what it's all about?

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

As evaluation criteria, it is natural to use the specification goals of *understandability*[2] and *unambiguity*. Furthermore, it is also of interest to estimate the *expressiveness* of the approach, i.e., how much of the intended specification task can be covered by the approach.

In the following table, a number of possible specification approaches are listed, with our judgements of the evaluation criteria.
A natural language specification can be understandable and expressive, depending on the size and quality of the text, but easily leads as we have discussed earlier to ambiguous specifications. Using a formal type system together with formal semantics [24] is here seen as having low understandability, since it requires high technical training. It is however very precise and fairly expressive.

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

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

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

Table 1. Possible specification approaches with estimated evaluation criteria.

We have also, for the sake of completeness, included related approaches such as the use of a test suite and reference implementation. The approach to use a test suite as a specification can be an interesting complement to abstract syntax and informal semantics. However, it is very important to state which description that has precedence if ambiguities are discovered. Finally, a reference implementation can also be seen as a specification, even if it is hard to get an good overview and reason about it.

3 An Abstract Syntax Specification Approach

In the following section we will go into more details about the proposal to use abstract syntax as part of the Modelica specification. Initially, the different abstract syntax representations are outlined in relation to the transformation process described in Section 2.1, followed by a discussion about the specification and representation of the syntax. Finally a small example of abstract syntax grammar is given and discussed.

3.1 Specifying the Elaboration Process

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

Consider Fig. 3, where the elaboration process is shown with surrounding ASTs. The first step in the process is the ordinary scanning and parsing step,

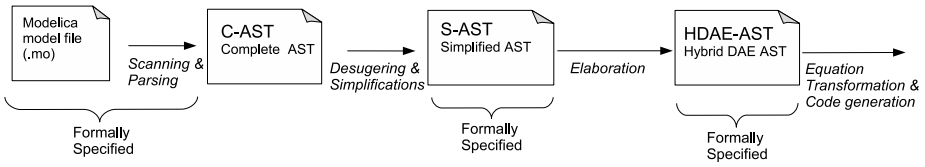

Fig. 3. Modelica's compilation process divided into intermediate representations in the form of abstract syntax trees (ASTs).

which is formally defined in the specification using lexical definitions and concrete syntax definitions using Extended BNF.

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

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

- *Desugaring* : The process of removing so called *syntactic sugar*, which is a convenient syntactic extension for the modeling engineer, but with no direct implication on the semantics. Example of such desugaring of a model is to collect all equation sections into one list, since the Modelica syntax allows several algorithm and equation sections to be defined in a model.

- *Canonical Transformations* Minor transformations and operations that help the S-AST to be a canonical form which is more suitable as input to the elaboration process. For example assigning correct prefixes to subelements (e.g., Section 3.2.2.1 in [20]).

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

The S-AST can be seen as a simplified internal language analogously to the *bare* language of Standard ML[18]. However, initially, we do not see a similar short and precise way of specifying the transformation from C-AST to S-AST, as the transformation rules are given in the Standard ML specification.

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

great help for the reader to actually know what the output is, not just assume it.

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

3.2 Specifying the Abstract Syntax

The specification of the syntax must be described using some kind of *grammar*, or data type construct in a language such as in Haskell, Standard ML, or MetaModelica [8].

The syntax can be specified using a *context-free* grammar, e.g. in Backus-Naur Form (BNF). However, we propose a more abstract definition of a grammar, where certain *meta-variables* range over names and identifiers. The notation has to some extent similarities to and is inspired by the abstract syntax definition of Featherweight Java[10].

For example, by stating that a meta variable R_r ranges over *names* (identifiers with possible dot-notation) referencing a `record`, we have introduced a contextual dependency in the grammar. The grammar declaratively states the requirement that this name must after lookup be a record, without stating *how* the name lookup should be performed. The latter must of course also be described in the specification, but in this way the different issues are separated. Consequently, this grammar is not intended to be used directly by a parser generator tool such as Yacc, but as a high-level specification which is less open for interpretation.

3.3 The Structure of an Abstract Syntax

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

When specifying a simple functional languages, it is common that the grammar of the abstract syntax only has one non-terminal, namely a *term* [24]. Hence, all evaluation semantics is performed on this node type only, and all terms can be nested into each other. This gives a very expressive language, but the constraining rules ensuring the validity of an input program must be given in another form. This form is normally a formal *type system*, describing allowed terms.

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

Somewhere inbetween those two extremes is for example the `SCODE` representation used in the earlier RML specification[14] and the current OpenModelica implementation.

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

Fig. 4. Example of a grammar for the connector non-terminal.

For the specification purpose, we suggest to use the most verbose alternative, i.e. the second alternative using many non-terminals. The rational for this choice is basically that this more restrictive form gives more information about what the actual input and output of the elaboration processes are.

3.4 A Connector S-AST Example with Meta-Variables

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

Consider Fig. 4, where an example of the non-terminal for a `connector` is listed using a variant of Extended BNF[3]. As usual, alternatives are seprated using

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

the '|' symbol, and curly brackets ({...}) denote that the enclosing elements can be repeated zero or more times.

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

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

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

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

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

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

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

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

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

4 Conclusion

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

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

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

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

Acknowledgments

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

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

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition.
2. David Broman, Peter Fritzson, and Sébastien Furic. Types in the Modelica Language. In *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria, 2006.
3. 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.
4. Dynasim. Dymola - Dynamic Modeling Laboratory with Modelica (Dynasim AB). `http://www.dynasim.se/` [Last accessed: 22 June 2007].
5. Christoph Nytsch-Geusen et. al. MOSILAB: Development of a Modelica based generic simulation tool supporting model structural dynamics. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.
6. Peter Fritzson. *Developing Efficient Language Implementations from Structural and Natural Semantics - Draft Version 0.97*. 2006. Book draft available from: `http://www.ida.liu.se/~pelab/rml/`.

7. 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.
8. Peter Fritzson, Adrian Pop, and Peter Aronsson. Towards Comprehensive Meta-Modeling and Meta-Programming Capabilities in Modelica. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.
9. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.
10. Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
11. ISO/IEC. *ISO/ IEC 14882 : Programming language C++*. ANSI, New York, USA, 1998.
12. ITI. SimulationX. `http://www.iti.de/` [Last accessed: 22 June 2007].
13. Gilles Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, 1987. Springer-Verlag.
14. David Kågedal. A Natural Semantics specification for the equation-based modeling languge Modelica. Master's thesis, Linköping University, 1998.
15. David Kågedal and Peter Fritzson. Generating a Modelica Compiler from Natural Semantics Specifications. In *Proceedings of the Summer Computer Simulation Conference*, 1998.
16. MathCore. MathModelica System Designer: Model based design of multi-engineering systems. `http://www.mathcore.com/products/mathmodelica/` [Last accessed: 8 March 2007].
17. Jakob Mauss. Modelica Instance Creation. In *Proceedings of the 4th International Modelica Conference*, Hamburg, Germany, 2005.
18. Robin Milner, Mads Tofte, Robert Harper, and David MacQuee. *The Definition of Standard ML - Revised*. The MIT Press, 1997.
19. Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling - Version 1*, September 1997. Available from: `http://www.modelica.org`.
20. 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`.
21. OpenModelica. Project. `http://www.ida.liu.se/~pelab/modelica/OpenModelica.html` [Last accessed: 22 June 2007].
22. Mikael Pettersson. *Compiling Natural Semantics*. PhD thesis, Linköping University, 1995.
23. 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.
24. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
25. Gordon D. Plotkin. A Structural Approach to Operational Semantics. Technical report, Dept. of Computer Science, University of Aarhus, 1981.
26. Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.