Towards a Formalized Modelica Subset[†]

Lucas Satabin[1] Jean-Louis Colaço[1] Olivier Andrieu[1] Bruno Pagano[1]

[1]Esterel Technologies/ANSYS SBU, France, firstname.lastname@ansys.com

Abstract

The ever growing requirement for safety in embedded systems, together with the willingness of having a modelling language to describe both the physics and the software that controls it makes Modelica an interesting candidate to design, simulate and implement complex systems. Originally designed to address multi-physics, since its version 3.3 Modelica integrates constructions to describe discrete controllers. Now the question of using Modelica to design critical embedded software arises.

In this paper we address the problem of defining a practical Modelica subset that can be entirely formalized and we sketch the formalization of this subset with the concrete example of static name resolution. This work should serve as a basis to define a suitable language that can be used to both simulate systems and generate embedded critical code.

Keywords: embedded systems, safety, code generation, formalization, name resolution

1 Introduction

Designing a complete programming language is a heavy task that involves many different aspects. The more features it contains, the more interactions between them are to be considered to ensure its correctness.

Modelica is an object-oriented language that was designed to simulate multi-physics systems. It is quite rich with a lot of constructs that are both static and dynamic. To make it a useful language, having a consistent behavior in its different implementations is a key point that can only be reached if it has a well documented and non-ambiguous semantics.

Moreover, the Modelica specification version 3.3 introduced new synchronous features that make it usable to design discrete controllers. It becomes tempting to use these features of Modelica to both simulate the physics with the controller and generate code for the controller, so that the same model is used for both activities.

Embedding code into critical systems (such as airplanes) requires some guarantees on the language and tools used for their development; the most important ones are: *determinism* and *absence of ambiguities*. For example, implicit and undefined behaviors are problematic in such settings and would lead to additionnal verification activities (e.g. tests or reviews) to satisfy the certification objectives. Hence, the need for a programming language with unambiguous semantics appears clearly if one wants to use it in the development of safety-critical software.

Formalizing the language is a good way to ensure its correctness and analyze the safety issues it could raise.

In the scope of the CertMod project[1], we worked on formalizing the static semantics of a Modelica subset as a basis for a qualified code generator development. In the remainder of this paper we use the terms *qualification* and *certification* as defined in DO-330 (2011): "Tool qualification is the process necessary to obtain certification credit for a tool." The current paper relates part of the results we produced during this project, which aims to provide a complete specification that can be used to develop a qualified code generator for Modelica. We focus on the basis elements identified in the scope of the CertMod project.

The contributions of this paper are the following:

- the definition of a practical subset of Modelica that can be formalized and used in a safety-critical context,
- a framework to formalize the various static aspects of Modelica and
- a formalization of static name resolution in Modelica.

The remainder of this paper is structured as follows: section 2 is a review of existing related work. Section 3 outlines the Modelica subset that is considered. Section 4 defines the formalization framework that will be used together with the notations. Section 5 depicts the name resolution within the formalism. Section 6 details the open points and future work.

[†]This work has been partially supported by the European Commission within the framework of the Clean Sky CertMod project with call identifier SP1-JTI-CS-2012-01.

[1]http://cordis.europa.eu/project/rcn/111584_en.pdf

2 Related Work

Modelica Association (2012) is the reference document for Modelica specification; it describes in natural language in a pretty free style all the constructs of the language and their behaviours in different contexts. The description of a given construct can be scattered over the entire specification document. This makes it hard to ensure that an implementation entirely respects it. Formalizing the language requires to be systematic in the description, in the sense of identifying the different aspects of correctness (naming, typing, clocking, ...) and for each of them going through each construct and define its correction condition with respect to this aspect. One of the first benefits of a formalization is to provide an organization of the different concerns. It was already identified in Broman et al. (2006) that the Modelica specification could benefit from a more formal definition. The language has grown complex and a lot of constructs interact with each other, hence it has become hard to reason about Modelica models. Another benefit of a formal description is to reduce the possible interpretations to the intended one; this goal is reached by the use of mathematical and well defined notations.

Modelica was not designed with safety-critical embedded controllers in mind. This means that some language features are either not relevant or not defined appropriately for such applications. This was discussed in Thiele et al. (2012) where a Modelica sub- and superset was sketched to address safety requirements. Our subset is based on the one identified in this work, but we decided to define a strict subset and no superset of Modelica. This decision to have a strict subset is motivated by the willingness to seamlessly integrate with existing implementations. No change is required between the model being simulated and the one generating the actual code.

Implementations compliance rapidly arises when several implementations of Modelica exist and different behaviors are observed. For example, **protected** elements in OpenModelica[2] may be accessed with the dot-notation whereas Dymola[3] does not allow to access **protected** classes. Also, the specification may be incomplete on some points and implementations must interpret it. For instance, defining the scope in which redeclaration as modifications takes place is subject to controversy[4].

```
class A
  replaceable class R end R;
end A;

class S type T = Real; end S;

class B
  extends A(redeclare class R = S);
  extends C;
end B;

class C
  class S type T = Integer; end S;
end C;
```

According to the specification, it is unclear what `B.R.T` represents, whether it is `C.S.T` (i.e. `Integer`) or `.S.T` (i.e. `Real`). Even though the various implementations agree on this particular ambiguity, it is still problematic in the context of a certification process, because the specification is the reference, not the implementation.

To address such problems, test suites can help disambiguating situations. A Modelica compliance test suite[5] is being developed that aims to validate various implementations of Modelica. Such test suites are useful to validate a compiler but can become hard to maintain up-to-date over time. In any case, these tests need oracles to validate implementation outputs and these oracles must be defined by the language specification. This is particularly important in a certification process such as DO-178C (2011), which requires to have test oracles based upon the specification. The typical approach used for software development is to write requirement based specification documents and test oracles are written using these requirements. This process allows for a clear traceability between requirements and test cases.

As of today, if the goal were to implement a qualified code generator for Modelica, the specification from Modelica Association (2012) coupled with a test suite would probably not satisfy a certification authority requirements.

In the industry, languages used to write embedded controllers are not all formalized. For instance, the C programming language is standardized[6] but implementations of certain constructs diverge depending on the compiler or the target platform. In the embedded software world, some rules and constraints are widely accepted and used to define a subset of C that aims to be safer. These guidelines are known under the name MISRA C[7].

The ultimate step in the formalization direction is having a formally described language and formally proven compiler, which gives a comprehensive formal proof that each transformation in the compiler preserves the semantics of the input program. The most advanced work in this area is incarnated by the CompCert C compiler Leroy (2009).

Finally, in the model-based approach to embedded software development, Scade 6 is the industrial dialect of the dataflow language Lustre, Halbwachs et al. (1991), extended with state machines, Colaço et al. (2005).

Since the latest major evolution of the language called

[2] `https://openmodelica.org/`

[3] `http://www.3ds.com/products-services/catia/products/dymola`

[4] For instance `https://trac.modelica.org/Modelica/ticket/1680`

[5] `https://github.com/modelica-compliance`

[6] For example by the ISO/IEC 9899:1999 aka. C99 standard.

[7] `http://www.misra.org.uk/`

Scade 6, the entire language static semantics is formally defined by various type systems that cover all constructs of the language. This work follows the approach chosen in the design of Lucyd Synchrone Pouzet (2006). This formalization is the basis of the Scade 6 certified compiler implementation. The present work is based on the very same idea and aims to provide a similar formalism level for a Modelica subset.

3 A Practical Modelica Subset

As we mentioned in the introduction, Modelica has a lot of constructs. Historically designed to model multi-physics systems with continuous time, it gains only recently the ability to describe synchronous controllers. In the scope of qualified embedded controllers development, only these synchronous features are of interest. Moreover, continuous time features are hard to formally describe and lots of behaviors depend on the solver at runtime. That is why we made the choice to formally describe a Modelica subset instead of taking the complete language.

To be of any practical use, the subset must be as complete as possible so that its expressiveness is not sacrificed for the sake of the formalization simplicity.

A first subset was described in Thiele et al. (2012), which was quite conservative. For example, import clauses where excluded from this subset. This kind of restriction can rapidly become annoying when dealing with existing libraries that make heavy use of import clauses (including the Modelica standard library. Our work is based on this subset with some additions to make it a more realistic subset.

3.1 Declarations

In the Modelica specification the language is defined as an *EBNF*[8] but syntactically allows for incorrect constructs. For instance, the EBNF does not prevent one from writing

```
function F
  input Integer i;
  output Integer o;
  equation
    o = i * 3;
end F;
```

This kind of class declaration is illegal (Modelica Association, 2012, section 12.2) as functions may not have equations but only statements in an **algorithm** section. However, this declaration is syntactically allowed (Modelica Association, 2012, section 4.5).
In comparison, the subset aims to syntactically enforce as many constraints as possible. Syntactically enforcing constraints allows for less normalization steps and checks after parsing, and makes the formalism simpler.

[8]Extended Backus-Naur Form

We added more syntactical restrictions on declarations. For example a package can only be defined inside a package and not inside other specialized classes. The same constraint exists for functions, which can only be declared in packages. Having a function declared in something else than a package makes it parameterized by all components of the class it is declared in. This restriction was thought as a way to improve modularity. In the following, we do not reason about the flattened model but about the structured input models. Checks that are described can be done in a modular way (i.e. without effective computation of the flattened model). Modifications in classes makes modular reasoning more complex. Also, we see functions as pure functions (Modelica Association, 2012, section 12.3) in the sense that they are side-effect free, and thus must not depend on the context of instantiation.

Modelica also allows many of type prefixes, or modifiers, for components. They are not all present in the subset. For instance, **inner** and **outer** components are not included, as they introduce an implicit binding that makes it hard to reason about. We will discuss this in section 6.

Declarations in Modelica can also be redeclared in inheriting classes. This features makes it possible to change the behavior of an inherited component by replacing it with another component. Although it is not forbidden by certification processes such as DO-178C (2011) and its object-oriented extension DO-332 (2011), we identify this feature as dangerous. Replacing or redeclaring components in this context requires more checks and validation to be performed to ensure that the global behavior and invariant of the inherited model are respected. This feature is not included in the subset, however parameter modifications are.

3.2 Equations

The selected subset contains basically all kind of equations that are meaningful in the context of synchronous models. It means the subset accepts these equations:

- simple equations that are flow definitions.
- **if**-equations
- clocked **when**-clauses
- **connect**-clauses

The only missing equation kind are **for**-equations. Their general form as defined in (Modelica Association, 2012, section 8.3.2) may introduce patterns that cannot be statically verified. Even though the expression the loop iterates over is required to be a parameter, it still allows to multiply define some cells of a vector or to leave some other cells non-initialized. However, adding **for**-equations that reduce to a *map* operator will be considered in future developments.

3.3 Expressions

The restrictions on the expressions are essentially the same as in Thiele et al. (2012). All continuous-time related operators are not included as they do not make sense in this context. On the other hand, most of the expressions related to synchronous features are included.

The aim of this paper is not to describe the subset entirely, and the complete grammar could not fit here. For the complete, refer to Satabin et al. (2015).

4 Formalization framework

The main contribution of this paper is to define a framework that can be used to formalize various aspects of the Modelica language. In programming language theory, it is used to distinguish two aspects of a language semantics:

- the *static semantics*, which corresponds to a certain (language dependent) level of correctness of syntactically correct programs required before execution, this aspect is statically checked at compile-time (i.e. without execution) and
- the *dynamic semantics*, which describes the behavior of the programs that are both syntactically and statically correct.

This separation reduces the set of programs to be considered by the dynamic semantics, in which one can assume that all the static aspects are respected.

In this work we focused on the static semantics of Modelica, which encloses:

- Static name lookup (Modelica Association, 2012, section 5.3).
- Type checking (Modelica Association, 2012, chapters 6 to 14).
- Clock checking (Modelica Association, 2012, chapter 16).

For each of these aspects we defined a dedicated system of inference rules, derived from the Modelica specification. The formalism used is based on works such as Igarashi et al. (2001), Igarashi (1999) for the object-oriented and type part or Forget et al. (2008) for the clock checking.

The Modelica syntax is rich and each construct may have several shapes. While writing a formalization it is more readable to have only one shape for each construct. That is why, the first step before formalizing is the normalization of declarations. [9]

[9]Note that this normalization must preserve correctness i.e. an incorrect program cannot normalize into a correct one and reciprocally.

4.1 Component Clauses

Components in Modelica are declared with component clauses. One such clause can declare several components of different array types. Moreover, clauses are grouped into public and protected sections which defines the visibility of each component declared in this section. Even though these syntactic constructs are allowed in our subset, component clauses are normalized so that:

- each clause declares exactly one component;
- each clause has a visibility, written ν, which corresponds to the section it is declared in;
- each clause has a set of modifiers (with restrictions discussed in section 3) written μ. If a declaration has no modifiers μ is the empty set, written $\emptyset$;
- each array subscript appears after the component name.

The normalization of component clauses is depicted in figure 1 where :

- $\mathtt{c},\ \mathtt{c}_1,\ \ldots,\ \mathtt{c}_i$ represent component declarations with potential array subscripts ;
- $\mathtt{T}$ represents a type identifier with potential array subscripts and
- $\mathtt{t}$ is a type identifier.

Hence, a component declaration is written $\nu\ \mu\ \mathtt{T}\ \mathtt{c}$.

We will also use lists of components in the following which will be written $\overline{\nu\ \mu\ \mathtt{T}\ \mathtt{c}}$. This notation is a shortcut for $\nu_1\ \mu_1\ \mathtt{T}_1\ \mathtt{c}_1,\ \ldots,\ \nu_n\ \mu_n\ \mathtt{T}_n\ \mathtt{c}_n$ for some $n \in \mathbb{N}^*$

4.2 Short Class Definitions

Modelica allows for so-called short class definition (Modelica Association, 2012, section 4.5.1). It is presented as syntactic sugar for simplified standard class definitions which does not introduce a new scope. Our subset allows for such declarations only for **`type`** and **`connector`**. The normalizing function *rewriteShort* is given in figure 2.

The component name λ is a fresh name which is generated during rewriting. Referring to a component whose type is declared with short class definition is equivalent to accessing the λ component. Enumerations are not rewritten because their only possible shape is with the short class definition. In the following, special rules will be written to handle them.

$$\textit{normDecl}(\nu\ \mu\ \texttt{t}\ [\texttt{n}_1,\ \ldots,\ \texttt{n}_p]\ \texttt{c;}) = \nu\ \mu\ \texttt{t}\ \texttt{c}[\texttt{n}_1,\ \ldots,\ \texttt{n}_p]\texttt{;}$$
$$\textit{normDecl}(\nu\ \mu\ \texttt{T}\ \texttt{c}_1,\ \ldots,\ \texttt{c}_q\texttt{;}) = \textit{normDecl}(\nu\ \mu\ \texttt{T}\ \texttt{c}_1\texttt{;})\ \ldots\ \textit{normDecl}(\nu\ \mu\ \texttt{T}\ \texttt{c}_q\texttt{;})$$
$$\textit{normDecl}(\nu\ \mu\ \texttt{t}\ \texttt{c;}) = \nu\ \mu\ \texttt{t}\ \texttt{c;}$$

Figure 1. Normalization of component clauses

$$\textit{rewriteShort}(\textbf{connector}\ \texttt{C} = \mu\ \texttt{T}) = \textbf{connector}\ \texttt{C}\ \mu\ \texttt{T}\ \lambda\texttt{;}\ \textbf{end}\ \texttt{C}$$
$$\textit{rewriteShort}(\textbf{type}\ \texttt{C} = \mu\ \texttt{T}) = \textbf{type}\ \texttt{C}\ \mu\ \texttt{T}\ \lambda\texttt{;}\ \textbf{end}\ \texttt{C}$$
$$\textit{rewriteShort}(\textbf{type}\ \texttt{C} = \mu\ \texttt{T[n]}) = \textbf{type}\ \texttt{C}\ \mu\ \texttt{T}\ \lambda\texttt{[n];}\ \textbf{end}\ \texttt{C}$$

where each occurrence of λ is fresh.

Figure 2. Normalization of short class definitions

4.3 Names

Components in Modelica are referred to by either simple names of the form `C` or by composite names of the form `A.B.C` (Modelica Association, 2012, chapter 5). These composite names, or paths, can be absolute, in which case they start with an dot, as in `.A.B.C`. To handle all paths uniformly in the upcoming formalization, we introduce the root package name, written $\star$. Absolute paths are thus written $\star$`.A.B.C` and all paths have the same shape.

In the following, we will differentiate between absolute resolved paths and unresolved paths (which can be either relative or absolute). For the sake of readability we will use notation $\underline{\mathbf{P}}$ for absolute paths of the form $\star$`.A.B.C` and $\underline{\texttt{P}}$ for unresolved paths of the form `A.B.C`.

4.4 Class Table

A Modelica model usually contains several classes organized into packages. These classes are stored in a table, written *CT*, that maps absolute class paths to their definition. A same path can only refer to at most one class definition. Construction of *CT* is done by walking through the syntactic structure of the model and by adding each encountered class definition name prefixed by its enclosing package path. This construction may fail if two classes are located at the same path. If it succeeds, all classes of the model are present in this table. The function *dom* is used to check whether an absolute path is an existing class with the notation `A.B.C` $\in$ *dom*(*CT*).
In the remainder of this paper, we consider that *CT* was successfully built.

After the short class definition rewriting that was discussed previously we can see classes in *CT* as the sets of components that are syntactically declared in them. For example, let's consider the class `C` below:

```
class C;
  Integer C1;
  parameter Boolean C2;
end C;
```

Conceptually this class is equivalent to the set of component declarations `C1` and `C2`, written {`Integer C1`; **`parameter`** `Boolean C2`}. We will use the notation `Integer C1` $\in$ *CT*(`A.B.C`) as a way to express the fact that a component is declared in a class.

4.5 Specialized Classes

The Modelica specification defines several specialized classes (Modelica Association, 2012, section 4.6). Most of the time, the specialized class kind does not matter, and they all are treated the same way and we will use the notation **`ckind`** to denote any specialized class kind. However sometimes the kind of specialized class is relevant to check some restrictions or allow some extensions. To this end, we define a function named *kind*, depicted in figure 3, that, given a class absolute path, returns the kind of specialized class it represents.

$$\textit{kind}(\star) = \textbf{package}$$
$$\textit{kind}(\underline{\mathbf{C}}) = \textbf{ckind}\ \text{if}\ \textit{CT}(\underline{\mathbf{C}}) = \nu\ \textbf{ckind}\ \texttt{C}\ldots\textbf{end}\ \texttt{C}$$
$$\textit{kind}(\underline{\mathbf{C}}) = \textbf{type}\ \text{if}\ \textit{CT}(\underline{\mathbf{C}}) = \nu\ \textbf{type} = \textbf{enumeration}(\ldots)$$

Figure 3. Specialized class kind

5 Name Resolution

As part of the formalization of Modelica's static semantics, the first aspect to consider is the name resolution. It is crucial in the sense that there must exists no ambiguity on what is referred to when a name is used in a model and neither correction can be decided nor compilation done without linking referenced names to the definition of the identified entity. Modelica has several features that are involved in this step and several rules that must be respected. It has modularization features, such as packages and visibility, that are to be taken into account.

In this section we propose a formalism for name resolution in our subset discussed in 3. It is written as a bunch of inference rules, each of which will be linked to the sections in the Modelica specification it was derived from.

5.1 Import Clauses

Classes and components can be imported in other classes to shorten the name that are referred to. There are four kinds of import clauses in Modelica (Modelica Association, 2012, section 13.2.1):

1. **`import`** `A.B.C` where `C` becomes visible in the lexical scope of the **`import`** clause.

2. **`import`** `A.B.{C, D, E}` where `C`, `D` and `E` become visible in the lexical scope of the **`import`** clause

3. **`import`** `A.B.*` where all elements defined in `A.B` become visible in the lexical scope of the **`import`** clause.

4. **`import`** `D = A.B.C` where `A.B.C` becomes visible with name `D` in the scope of the **`import`** clause.

Clauses of the second form can be reduced to case one by duplicating **`import`** clauses as many times as there are imported elements and will be treated as such in the following. In case three the import clause is said to be *unqualified* and has lower priority than other import clauses that are said to be *qualified* (Modelica Association, 2012, section 5.3.1). Case four allows to introduce a different local name for imported elements that otherwise would conflict.

Imported names are always fully qualified names (Modelica Association, 2012, section 13.2.1.1). It means that if one writes **`import`** `A` in Modelica, it will be treated as **`import`** `★.A`. In other words, only absolute composite names are imported.

We define the *imports* function that will be used in the following to get the list of unresolved **`import`** from a resolved path. Each import returned by this function is the pair containing the import name and the imported path. In the case of unqualified imports, the empty set symbol $\emptyset$ is returned instead of a name.

$$\mathit{imports}\,(\underline{\mathbf{C}}) = \{\ \mathit{namePath}(\mathtt{imp})\ |\mathtt{imp} \in \mathit{CT}\,(\underline{\mathbf{C}})\ \}$$

where function *namePath* is defined by:

$$\begin{aligned}\mathit{namePath}\,(\mathbf{import}\ \underline{\mathtt{A}}.\mathtt{B}) &= (\mathtt{B},\ \underline{\mathtt{A}}.\mathtt{B})\\ \mathit{namePath}\,(\mathbf{import}\ \mathtt{C} = \underline{\mathtt{A}}.\mathtt{B}) &= (\mathtt{C},\ \underline{\mathtt{A}}.\mathtt{B})\\ \mathit{namePath}\,(\mathbf{import}\ \underline{\mathtt{A}}.\star) &= (\emptyset,\ \underline{\mathtt{A}})\end{aligned}$$

5.2 Inheritance

Modelica is an object-oriented language that allows for multiple inheritance (Modelica Association, 2012, section 7.1.1). A class may contain as many **`extends`** clauses as wanted in any order. We define the function *extends* which returns the list of unresolved extended path of a resolved path.

$$\mathit{extends}\,(\underline{\mathbf{C}}) = \{\ \underline{\mathtt{X}}\ |\ \mathbf{extends}\ \underline{\mathtt{X}} \in \mathit{CT}\,(\underline{\mathbf{C}})\ \}$$

5.3 Visibility

Modelica defines two level of visibility: **`public`** and **`protected`**. The **`protected`** visibility means that the element cannot be accessed via the dot notation (Modelica Association, 2012, section 4.1). Visibility appears in several kinds of clauses: **`extends`** clauses, component clauses and class definition. An **`extends`** clause may be protected, which means that all inherited components and classes are considered **`protected`** from the inheriting class (Modelica Association, 2012, section 7.1.2).

When resolving names, we would need to check that a name is visible when accessing it with the dot notation (Modelica Association, 2012, section 4.1).

5.4 Static Name Lookup

Based on the previous definitions, we can define name lookup in our Modelica subset. It starts with the static name lookup, where all the classes and their component names are resolved. The complete set of rules are depicted in figure 4.

Judgements of these rules must be read as follows:

- $\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\bullet}{\Rightarrow} \underline{\mathbf{D}}$ means "the simple name `C` seen from $\underline{\mathbf{P}}$ is resolved to path $\underline{\mathbf{D}}$."

- $\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Rightarrow} \underline{\mathbf{D}}$ means "the simple name `C` seen from $\underline{\mathbf{P}}$ is resolved to path $\underline{\mathbf{D}}$ by only using named elements of $\underline{\mathbf{P}}$ or its super classes."

- $\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Uparrow}$ means "the simple name `C` seen from $\underline{\mathbf{P}}$ cannot be resolved by only using named elements of $\underline{\mathbf{P}}$ or its super classes."

- $\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\bullet}{\Uparrow}$ means "the simple name `C` seen from $\underline{\mathbf{P}}$ cannot be resolved by using named elements of $\underline{\mathbf{P}}$ or its super classes nor import clauses of $\underline{\mathbf{P}}$."

Conceptually, the class path on the left of the $\vdash$ symbols gives the scope of the lookup and unambiguously describes where the search must start.

Several aspects of the static name lookup are of interest in this formalization. First, visibility is not taken into account. The reason why and impacts will be discussed in section 6. Then, we can see that few rules are needed

$$\textsc{N-Root}\ \frac{}{\vdash \star \overset{\circ}{\Rightarrow} \star} \qquad \textsc{N-Self}\ \frac{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Rightarrow} \underline{\mathbf{D}}}{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\bullet}{\Rightarrow} \underline{\mathbf{D}}} \qquad \textsc{N-InCT}\ \frac{\underline{\mathbf{P}}.\mathtt{C} \in \mathit{dom}(\mathit{CT}) \qquad \nu\ \mu\ \underline{\mathtt{T}}\ \mathtt{C} \notin \mathit{CT}(\underline{\mathbf{P}})}{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Rightarrow} \underline{\mathbf{P}}.\mathtt{C}}$$

$$\textsc{N-Comp}\ \frac{\underline{\mathbf{P}}.\mathtt{C} \notin \mathit{dom}(\mathit{CT}) \qquad \nu\ \mu\ \underline{\mathtt{T}}\ \mathtt{C} \in \mathit{CT}(\underline{\mathbf{P}})}{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Rightarrow} \underline{\mathbf{P}}.\mathtt{C}}$$

$$\textsc{N-Super}\ \frac{\begin{array}{c}\underline{\mathbf{C}}.\mathtt{D} \notin \mathit{dom}(\mathit{CT}) \qquad \nu\ \mu\ \mathtt{X}\ \mathtt{D} \notin \mathit{CT}(\underline{\mathbf{P}}) \\ \underline{\mathtt{X}} \in \mathit{extends}(\underline{\mathbf{C}}),\ (\underline{\mathbf{C}} \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{Y}} \wedge \underline{\mathbf{Y}} \vdash \mathtt{D} \overset{\circ}{\Rightarrow} \underline{\mathbf{T}}) \\ \forall\ \underline{\mathtt{Z}} \in \mathit{extends}(\underline{\mathbf{C}}),\ \underline{\mathbf{C}} \vdash \underline{\mathtt{Z}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{W}} \wedge (\underline{\mathbf{W}} \vdash \mathtt{D} \overset{\circ}{\Rightarrow} \underline{\mathbf{T}} \vee \underline{\mathbf{W}} \vdash \mathtt{D} \overset{\circ}{\Uparrow})\end{array}}{\underline{\mathbf{C}} \vdash \mathtt{D} \overset{\circ}{\Rightarrow} \underline{\mathbf{T}}}$$

$$\textsc{N-NoSelf}\ \frac{\underline{\mathbf{P}}.\mathtt{C} \notin \mathit{dom}(\mathit{CT}) \qquad \nu\ \mu\ \mathtt{X}\ \mathtt{C} \notin \mathit{CT}(\underline{\mathbf{P}}) \qquad \forall\ \underline{\mathtt{X}} \in \mathit{extends}(\underline{\mathbf{P}}),\ \underline{\mathbf{P}} \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{Y}} \wedge \underline{\mathbf{Y}} \vdash \mathtt{C} \overset{\circ}{\Uparrow}}{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Uparrow}}$$

$$\textsc{N-ImportQual}\ \frac{\begin{array}{c}\underline{\mathbf{C}} \vdash \mathtt{D} \overset{\circ}{\Uparrow} \\ (\mathtt{D},\ \underline{\mathtt{X}}.\mathtt{Y}) \in \mathit{imports}(\underline{\mathbf{C}}) \qquad \star \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{P}} \qquad \underline{\mathbf{P}} \vdash \mathtt{Y} \overset{\circ}{\Rightarrow} \underline{\mathbf{E}} \qquad \mathit{kind}(\underline{\mathbf{P}}) = \mathbf{package}\end{array}}{\underline{\mathbf{C}} \vdash \mathtt{D} \overset{\bullet}{\Rightarrow} \underline{\mathbf{E}}}$$

$$\textsc{N-ImportUnqual}\ \frac{\begin{array}{c}\underline{\mathbf{C}} \vdash \mathtt{D} \overset{\circ}{\Uparrow} \qquad (\mathtt{D},\ \_) \notin \mathit{imports}(\underline{\mathbf{C}}) \\ (\emptyset,\ \underline{\mathtt{X}}) \in \mathit{imports}(\underline{\mathbf{C}}) \qquad \star \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{P}} \qquad \underline{\mathbf{P}} \vdash \mathtt{D} \overset{\circ}{\Rightarrow} \underline{\mathbf{E}} \qquad \mathit{kind}(\underline{\mathbf{P}}) = \mathbf{package}\end{array}}{\underline{\mathbf{C}} \vdash \mathtt{D} \overset{\bullet}{\Rightarrow} \underline{\mathbf{E}}}$$

$$\textsc{N-NoImport}\ \frac{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\circ}{\Uparrow} \qquad (\mathtt{D},\ \_) \notin \mathit{imports}(\underline{\mathbf{C}}) \qquad \forall\ (\emptyset,\ \underline{\mathtt{X}}) \in \mathit{imports}(\underline{\mathbf{P}}),\ \underline{\mathbf{P}} \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{Y}} \wedge \underline{\mathbf{Y}} \vdash \mathtt{C} \overset{\circ}{\Uparrow}}{\underline{\mathbf{P}} \vdash \mathtt{C} \overset{\bullet}{\Uparrow}}$$

$$\textsc{N-Encl}\ \frac{\mathit{CT}(\underline{\mathbf{P}}) = \mathbf{ckind}\ \mathtt{P}\ \mathbf{...end}\ \mathtt{P} \qquad \underline{\mathbf{P}}.\mathtt{C} \vdash \mathtt{D} \overset{\bullet}{\Uparrow} \qquad \underline{\mathbf{P}} \vdash \mathtt{D} \overset{\bullet}{\Rightarrow} \underline{\mathbf{E}}}{\underline{\mathbf{P}}.\mathtt{C} \vdash \mathtt{D} \overset{\bullet}{\Rightarrow} \underline{\mathbf{E}}}$$

$$\textsc{N-Encaps}\ \frac{\mathit{CT}(\underline{\mathbf{P}}) = \mathbf{encapsulated\ ckind}\ \mathtt{P}\ \mathbf{...end}\ \mathtt{P} \qquad \underline{\mathbf{P}}.\mathtt{C} \vdash \mathtt{D} \overset{\bullet}{\Uparrow} \qquad \star \vdash \mathtt{C} \overset{\bullet}{\Rightarrow} \underline{\mathbf{E}}}{\underline{\mathbf{P}}.\mathtt{C} \vdash \mathtt{D} \overset{\bullet}{\Rightarrow} \underline{\mathbf{E}}}$$

$$\textsc{N-Dot}\ \frac{\underline{\mathbf{P}} \vdash \underline{\mathtt{C}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{D}} \qquad \underline{\mathbf{D}} \in \mathit{dom}(\mathit{CT}) \qquad \underline{\mathbf{D}} \vdash \mathtt{E} \overset{\circ}{\Rightarrow} \underline{\mathbf{F}} \qquad \mathit{kind}(\underline{\mathbf{D}}) = \mathbf{package}}{\underline{\mathbf{P}} \vdash \underline{\mathtt{C}}.\mathtt{E} \overset{\bullet}{\Rightarrow} \underline{\mathbf{F}}}$$

Figure 4. Static Name Lookup

to formally describe the entire static lookup semantics. It represents aspects that are scattered in the specification, all put together here in a unified framework.

N-Root indicates that the root package name $\star$ always resolves to itself. It is the base case when looking up in the enclosing classes. All predefined types (such as `Integer`) and predefined functions (such as `abs`) are considered to be defined in the root package.

N-Self simply states that if a simple name can be resolved with the named elements of a class or its super classes, then it is resolved. It is kind of a weakening rule for name resolution since the premise gives a stronger information than the conclusion.

The base rules N-InCT and N-Comp look up for a simple name in the current class. They state that if a simple name is declared in the current class, then it resolves to this name augmented with the path of the current class. The same name cannot be defined twice in the same class (Modelica Association, 2012, section 5.6.3).

The rule N-Super treats the case where a simple name is defined in inherited classes. A same name `C` can be inherited multiple times if and only if all inherited elements with name `C` are exactly identical (Modelica Association, 2012, section 7.1). In this rule *identical* means that the resolved path is the same for all inherited elements with name `C`.

If the rules we saw so far do not apply to resolve a simple name `C`, then we conclude that it is not defined in the current class. This is what the rule N-NoSelf means.

In such a case, the rules N-ImportQual and N-ImportUnqual may be applied, to lookup for the simple name in the import clauses. The former rule looks up for the name in qualified imports, while the latter one looks up in unqualified import if no qualified import clause allowed to resolve the name (Modelica Association, 2012, section 5.3.1). Imported paths are resolved starting in the root package (Modelica Association, 2012, section 13.2.1.1) and names can only be imported from packages (Modelica Association, 2012, section 13.2.1.2). Our subset allows packages to be defined only packages, that is why it is sufficient to check that only the last element of the path is a package.

If none of the import-related rules described in the previous paragraph applies to resolve a simple name `C`, then we conclude that it is not defined in the current class, nor is it imported. This is the meaning of rule N-NoImport.

Only in this case, the simple name must be resolved by looking up in the enclosing classes. Two different cases may apply at this point depending on the definition of the current class. If the current class is declared **`encapsulated`**, rule N-Encaps applies and the name is looked up in the root package (Modelica Association, 2012, section 5.3.1). In the case the class is not **`encapsulated`**, rule N-Encl applies and the name is looked up in the directly enclosing class.

Finally, the last case deals with composite names. The set of rule deals only with static name resolution, which means that composite names corresponding to component accesses of class instances are not treated here. We will discuss such cases in section 5.5. Static resolution of composite names is only allowed for names defined in packages, as stated by rule N-Dot. The last name in the path is looked up among elements defined or inherited in the package resolved so far (Modelica Association, 2012, section 5.3.2).

5.5 Component Lookup

In the previous section we covered the static name lookup only. This is, the resolution of class names in packages and component names in packages. Composite names that access components inside components require some typing information to be resolved. They indeed require to be aware of the structure of the component to decide what component the name represents. This structure is only known once all static names are resolved. We can then gather the list of components in a component using the *components* function depicted in figure 5. The *extendComponents* function allows to retrieve all inherited components.

Components of a resolved class are all the components defined in this class or inherited. Resolving accesses to components is done by the type checking. The type system is beyond the scope of this paper, but the typing rule T-Dot which describes component access in a component would look like this.

$$\text{T-Dot}\ \frac{\texttt{n}:\ \underline{\mathbf{T}} \qquad \nu\ \mu\ \underline{\mathbf{C}}\ \texttt{c} \in \mathit{components}\,(\underline{\mathbf{T}})}{\texttt{n.c}:\ \underline{\mathbf{C}}}$$

It reads as: if a simple name `n` has a resolved type $\underline{\mathbf{T}}$, then we can resolve and type `n.c` if `c` is a component of $\underline{\mathbf{T}}$ with type $\underline{\mathbf{C}}$. Of course this rule is just a sketch and more concepts are taken into account by the real type system.

5.6 Class Resolution

A class in Modelica is said to be resolved if several constraints are respected:

- All component types can be resolved ;
- All **`import`**-clauses can be resolved ;
- All **`extends`**-clauses can be resolved ;
- All components defined in a class must have names distinct from inherited components. In Modelica component may have the same name if they are syntactically equal (Modelica Association, 2012, section 7.1). The specification recommends to emit a warning in this case, but we decided to forbid it, as it does not bring anything to define twice components that are exactly the same, and most probably it is a symptom of model design problem ;

$$\mathit{extendComponents}(\underline{\mathtt{C}}) = \bigcup_{\underline{\mathtt{D}} \in \mathit{extends}(\underline{\mathtt{C}})} \{\ \nu_x\ \mu_x\ \underline{\mathtt{X}}\ \mathrm{x} \mid \nu_x\ \mu_x\ \underline{\mathtt{X}}\ \mathrm{x} \in \mathit{components}(\underline{\mathtt{D}})\ \}$$

$$\mathit{components}(\underline{\mathtt{C}}) = \begin{cases} \{\ \mathtt{\textbf{public}}\ \underline{\mathtt{C}}\ \mathrm{E}_i \mid \mathrm{i} \in [1..n]\ \} & \text{if } \mathit{CT}(\underline{\mathtt{C}}) = \mathtt{\textbf{enumeration}}(\mathrm{E}_1,\ \ldots,\ \mathrm{E}_n) \\ \{\ \nu\ \mu\ \underline{\mathtt{X}}\ \mathrm{x} \mid \nu\ \mu\ \underline{\mathtt{X}}\ \mathrm{x} \in \mathit{CT}(\underline{\mathtt{C}})\ \} \cup \mathit{extendComponents}(\underline{\mathtt{C}}) & \text{otherwise.} \end{cases}$$

Figure 5. Component Lookup Function

- All qualified **`import`**-clauses to distinct names ;
- All unqualified **`import`**-clauses bring distinct names into scope ;

Rule N-Class in figure 6 gives the rule that ensures that all names are resolved in a class. And that all constraints defined above are respected.

6 Discussion and Future Work

In the context of the CertMod project, we also formalized type checking and clock checking of models based on similar rules. This work is the basis that we used to write a complete Modelica front-end that performs all static checks we described on input models. It can also be used to write or verify oracles in a compliance test suite, and then test the model checker against these oracles.

Some restrictions present in the current subset could be removed, and some omissions could be added. For instance, we did not take visibility of elements into account. This was motivated by our tests on various Modelica implementations which did not agree with neither the specification nor between each other. Adding visibility to the name resolution rules would be quite easy though. Only rules N-Import and N-Dot would need to take this visibility into account. The *extends* function would also require to return the visibility of the extends clause. Similarly the sketched T-Dot rule would require that ν is **`public`**.

Other constructs that were not taken into account in name resolution rules in this paper are **`inner`**/**`outer`** declarations (Modelica Association, 2012, section 5.4). These constructs introduce an implicit name, inherited from an enclosing class. They represent a handy way of having global parameters in a model that we do not bother passing explicitly to each part requiring it. Adding these constructs to the subset would require to add rules to resolve **`outer`** names. These rules would be quite complex, considering the restrictions and constraints that exists on them. The rules must represent the fact that the closest **`inner`** component with the same name is selected when the class is instantiated.

Redeclarations in inheriting classes are also not included in our subset. Redeclaring classes allows for having a class name denoting a completely different path in a sub-class than in the inherited one. Remember the lookup scope problem for redeclarations we discussed in section 2. Formalizing redeclarations would definitively help clarify the situation by having a non ambiguous way of describing the lookup scope. However, the resulting rules would be quite complex because for each name one should lookup for the current redeclaration, if any.

This added complexity makes it harder to read and understand the rule, but is symptomatic of an intrinsic complexity in the language construct. As a rule of thumb, the fact that a construct introduces complexity in the formalism can be seen as a hint whether the construct is legit or not. A too high complexity reflects a construct that will be hard to understand for modelers, and to implement correctly by tool providers.

7 Conclusion

In this work, we described all the rules related to name resolution as described in the Modelica specification. It was interesting and enlightening to compile the rules and constraints that appear at various places in the specification into a single place. It also allowed us to detect some features that may be problematic to write a qualified code generator for Modelica. For example, in the rules depicted in figure 4, the involved concepts are usual in object-oriented languages. However, the unqualified import clause lookup described by rule N-ImportUnqual implies a priority in name lookup that would require more validation activities to be used. The mix with qualified imports makes it also harder for the modeler to determine which element is selected. The safest way to deal with this problem would be to avoid unqualified imports all together, and to exclude them from the subset. Moreover, the encapsulated concept appears to be quite exotic and would also require extra checks to be performed as it introduces some irregularities in the lookup algorithm. Language features must ensure the highest possible level of safety, and restricting some constructs can benefit to developers. Expressiveness is important in a language but for safety-critical software development, safety and non ambiguity is even more important.

The considered subset presented here only includes discrete synchronous features of Modelica and the formalization only deals with static aspects of this subset. Adding the dynamic semantics of the subset appears to be an important step to take to achieve a comprehensive formal description of the language. Such a semantics would describe how a model behaves when it is instantiated and how the generated code must behave as well. This can be used to write oracles in the test suite and then

$$\textsc{N-Class}\ \frac{\begin{array}{c}
\forall\ \nu\ \mu\ \mathtt{X}\ \mathtt{x}\ \in\ \mathit{CT}\,(\underline{\mathbf{C}})\,,\ \underline{\mathbf{C}} \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \_ \\
\forall\ (\_,\ \underline{\mathtt{X}})\ \in\ \mathit{imports}\,(\underline{\mathbf{C}})\,,\ \underline{\mathbf{C}} \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \_ \\
\forall\, \underline{\mathtt{X}}\ \in \mathit{extends}(\underline{\mathbf{C}}),\ \left(\underline{\mathbf{C}} \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{X}} \wedge \forall\, \nu_c\, \mu_c\, \mathtt{C}\ \ \mathtt{c}\ \in \mathit{CT}(\underline{\mathbf{C}}), \underline{\mathbf{X}} \vdash \mathtt{c} \overset{\circ}{\Uparrow}\right) \\
\forall\ ((\mathtt{N},\ \underline{\mathtt{X}}),\ (\mathtt{N},\ \underline{\mathtt{Y}}))\ \in\ \mathit{imports}(\underline{\mathbf{C}}) \times \mathit{imports}(\underline{\mathbf{C}}),\ \left(\star \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{D}} \wedge \star \vdash \underline{\mathtt{Y}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{D}}\right) \\
\forall\ ((\emptyset,\ \underline{\mathtt{X}}),(\emptyset,\ \underline{\mathtt{Y}}))\ \in\ \mathit{imports}(\underline{\mathbf{C}}) \times \mathit{imports}(\underline{\mathbf{C}}),\ \left(\begin{array}{ll}
 & \star \vdash \underline{\mathtt{X}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{A}} \\
\wedge & \star \vdash \underline{\mathtt{Y}} \overset{\bullet}{\Rightarrow} \underline{\mathbf{B}} \\
\wedge & \underline{\mathbf{A}} \vdash \mathtt{C} \overset{\circ}{\Rightarrow} \_ \implies \underline{\mathbf{B}} \vdash \mathtt{C} \overset{\circ}{\Uparrow}
\end{array}\right)
\end{array}}{\underline{\mathbf{C}}}$$

Figure 6. Class Resolution

validate simulators as well as code generators and check that they agree on the behavior through the test suite.

A complete formal semantics of a language brings also the possibility to write proofs on the language. This is useful to ensure that the type-system is sound and that the language has a deterministic behavior. Reaching this point naturally requires a lot more work to be done, and the continuous part of Modelica would be quite problematic to semantically describe.

The presented work is a first small step toward having a formally described version of Modelica. Although we only covered a small part of the various aspects of the language, it sets up a framework for a more comprehensive formalization. It already brings some clarity where rules written in English may be misinterpreted. It is also a comprehensive, concise and non-ambiguous way to describe these rules. We believe that it is a huge step forward and that it can help clarifying things when it is hard to interpret the specification. We also believe that this work can help in writing the next versions of the Modelica specification. Not necessarily does it mean that this exact formalism must be included in it, but having this way of describing behaviors in mind helps writing more comprehensive and rigorous specification.

Acknowledgments

We would like to particularly thank Martin Otter, Bernhard Thiele and Daniel Schlabe for their precious help and their answers during this work. We also want to thank Marc Pouzet for his comments on the clock calculus we developed in this project.

References

David Broman, Peter Fritzson, and Sébastien Furic. Types in the modelica language. In *Proceedings of the Fifth International Modelica Conference*, 2006.

Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *EMSOFT'05*, September 2005.

DO-178C. DO-178C Software Considerations in Airborne Systems and Equipment Certification, December 2011.

DO-330. DO-330 Software Tool Qualification Considerations, December 2011.

DO-332. DO-332 Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A, December 2011.

Julien Forget, Frédéric Boniol, David Lesens, and Claire Pagetti. A multi-periodic synchronous data-flow language. In *HASE 2008. 11th IEEE*. IEEE, 2008.

Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language lustre. In *Proceedings of the IEEE*, 1991.

Atsushi Igarashi. *Formalizing Advanced Class Mechanisms*. PhD thesis, University of Tokyo, 1999.

Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, May 2001.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7), 2009.

Modelica Association. Modelica – A Unified Object-Oriented Language for Systems Modeling, version 3.3. `http://modelica.org`, May 2012.

Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

Lucas Satabin, Olivier Andrieu, Bruno Pagano, and Jean-Louis Colaço. Formalization of A Modelica Subset for Safety-Critical Software Development. Technical report, Esterel Technologies, 2015.

Bernhard Thiele, Stefan-Alexander Schneider, and Pierre R Mai. A Modelica Sub-and Superset for Safety-Relevant Control Applications. In *Proceedings of the Ninth International Modelica Conference*, 2012.