A Reference-Based Parameterization Scheme for Equation-Based Object-Oriented Modeling Languages

Dr. Dirk Zimmer

German Aerospace Center, Institute of Robotics and Mechatronics
Münchnerstr. 20, D-82234 Oberpfaffenhofen (email: dirk.zimmer@dlr.de).

Abstract: This paper presents a reference-based parameterization scheme for equation-based, object-oriented modeling languages such as Modelica. It is demonstrated, how simple language constructs can be designed that enable a general and powerful parameterization of models. Furthermore, the computational process required for this parameterization scheme is outlined. To validate our concepts, an experimental language has been developed and implemented. It is called Hornblower and it represents an attempt to embrace the core ideas of Modelica while reorganizing the higher-level modeling tasks that have evolved during time.

Keywords: object-oriented modeling, language design, model parameterization

1. INTRODUCTION

An ever increasing complexity of equation-based modeling has led to the creation of increasingly larger systems that contain a vast number of components. Such systems raise the demand for elaborated parameterization schemes. It is no longer sufficient to regard parameters simply as real numbers or values of any other base-type. Instead, whole components or classes of components become subject to parameterization. For instance, the gear-box of a vehicle needs to be changed, or the class of medium-models within a fluid system shall be replaced by another one.

In Modelica, one of the most prominent object-oriented modelling languages, several language constructs have been introduced in order to support such advanced parameterization tasks. Unfortunately, they revealed to be conceptually flawed and made the language more complex and difficult to maintain than originally intended.

The publication "Towards Improved Class Parameterization and Class Generation in Modelica" (Zimmer, 2010) outlines the current deficiencies and contains a concrete proposal for a redesign. By analyzing the current use of component parameters and class parameters, we exposed that one reason for the inadequate complexity is that a completely different set of syntax elements is used for setting component parameters than for normal conventional parameters.

The following line presents the syntax for setting a normal parameter. We define the length of a vehicle model at its declaration:

```
Vehicle car1(length = 5.2);
```

If the gearbox is a parameter we have to use a different syntax though:

```
Vehicle car1(redeclare AutomGear gear);
```

It is not possible to state this parameterization using the conventional syntax for parameter setting such as:

```
Vehicle car1 (gear = AutomGear() );
```

The reason for this is that in Modelica components or classes of components do not have first class status (Burstall, 2000) and thus cannot be part of a normal expression. This, unfortunately, implies many restrictions. For example, it is not possible to make a component parameter dependent on a Boolean expression.

The proposed changes in (Zimmer, 2010) aimed therefore to raise the status of component declarations to first class in such a way that the following statement would be valid Modelica code:

```
Vehicle car1 (gear = if c== "A" then
AutomGear() else ManualGear() );
```

This was outlined by a number of examples and a detailed listing of required grammar changes. The suggested improvements remained, however, on a conceptual level since they have not been validated by a corresponding test-implementation.

Meanwhile, such a prototype implementation has become available by the design effort of the experimental language Hornblower. This language represents an attempt to embrace the core ideas of Modelica while reorganizing the higher-level modeling tasks that have evolved during time.

In contrast to Modelica, Hornblower is based on a reference-based parameterization scheme. This enables to handle whole components or classes of models as simple parameters in a simple unified concept. In addition to simplicity, the concept of Hornblower incorporates many further particular advantages such as:

- Models, model classes and even annotations are elements of first-class status (Burstall, 2000).
- The handling of attributes or annotations can be integrated into the process of parameter evaluation with virtually no effort.
- Sharing objects as for the case of world models is now easily possible. Complicated concepts such as inner/outer (Fritzson, 2004) can be abandoned.
- Large parameter arrays can be handled more efficiently in the translation process.

This paper will present the project Hornblower with its parameterization scheme. To this end, we present a staged compilation scheme and then look at one of the compiler stages in more detail. This includes the design of the language constructs as well as the computational realization of the compilation process. Finally an example is presented.

2. STAGED COMPILATION OF HORNBLOWER

Modeling of physical systems is a multi-faceted task. A suitable modelling language has therefore to meet several requirements. Three major objectives can be identified:

- Declare variables and equations that relate them in order to build up a component consisting in a system of differential-algebraic equations (*Low Level: Component Modelling*).
- Compose systems out of generic components and parameterize them so that they fit the concrete task (*Medium Level: System Modelling*).
- Create whole modelling libraries and enable code reuse (*High Level: Library Development*).

For each of these objectives, the modeller requires separate means in the language. These means will then be processed by separate stages in the compilation scheme.

Figure 1 outlines the processing of the Hornblower language. It is divided into a compilation scheme with multiple stages. The first stage is the parsing of the language. It is considered to be trivial. The last stage is the actual generation of simulation code. This is beyond the scope of this project. Instead, we take on the three centre stages in this document that directly correspond to the three main objectives:

These are:

- Elaboration (Library Development)
- Instantiation (System Modelling)
- Flattening (Component Modelling)

Like Modelica, Hornblower is a declarative language. The central idea of a declarative language is that the modeller states what he wants and can refrain from describing how to achieve this. The burden of the computational realization is taken away from him by a suitable algorithm in processing scheme of the language.

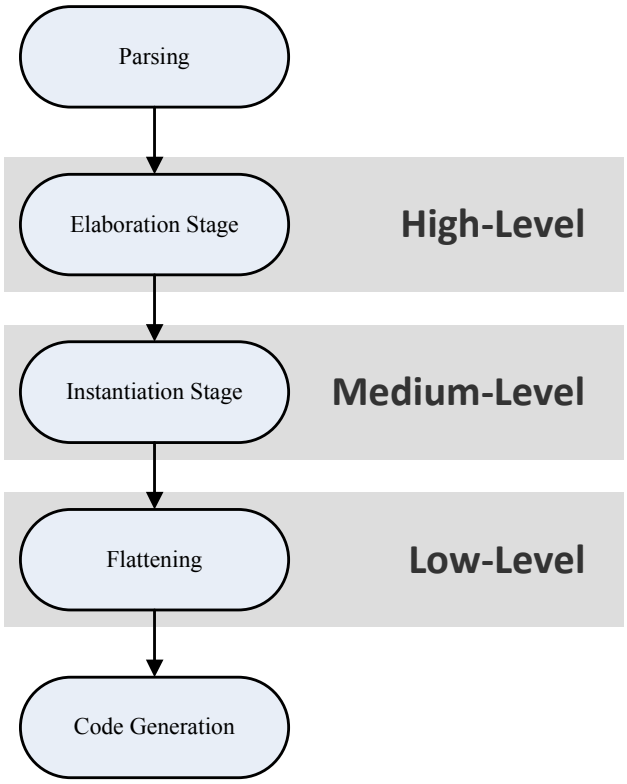

Fig. 1. Compilation Stages of Hornblower.

Since each processing stage in Hornblower has its own, clearly separated algorithm for compilation, the language essentially represents a combination of three (layered) declarative sub-languages.

In concrete (but simplified) terms this means:

- The modeler may define classes and relate them with other definitions of classes (by import statements or extension). The algorithm of the elaboration stage then determines the appropriate order of class generation.
- The modeler may declare components and assign values to parameters. The algorithm of the instantiation stage then determines the correct order of instantiation and parameter evaluation.
- The modeler creates a hierarchic system, built out of several components. The algorithm of the flattening stage then separates equations and data-structures and compiles a global set of equations. In addition, structural parameters are separated from conventional parameters.

This document centers on the parameterization scheme and hence we will exclusively concern ourselves with the instantiation stage in the following sections.

3. INSTANTIATION STAGE

3.1 Objectives

The instantiation stage concerns the medium-level constructs in Hornblower. Its goal is to perform all processes regarding the instantiation of classes and the evaluation of parameters. Let us define the main modelling objectives that correspond to the instantiation stage:

- The modeller shall have means to declare components and to compose its subsystem from them.
- The modeller shall have means to set parameters for his components.
- These parameters might influence the structure or just the numeric solution of the system.
- The modeller shall have means to exchange sub-components or classes like a parameter.

3.2 Hornblower Basics.

The following two model definitions represent typical Hornblower code:

```
define model Body
  variable tau as Real;
  variable phi as Real;
  parameter I := 1.0 as Real;
implementation
  variable w as Real;
  w = der(phi);
  tau = I*der(w);
end;

define model Engine
  variable tau as Real;
  variable phi as Real;
  parameter T := 1.0 as Real;
implementation
  tau = (1+sin(phi))*T;
end;
```

The definition of a model in Hornblower consists in two sections. The header section contains the declaration of variables, parameters, and components. The implementation section completes the model by providing equations and further private declarations.

Variables represent time-varying values of a specific static type description following the keyword **as**. Values can be equated by the operator "=".

Parameters and components represent references to objects or values and are of dynamic type. This type is inferred if not explicitly stated. References can be assigned by the operator ":=". The precise difference between parameters and componentes is outlined in section 3.4.

3.3 Declaration and assignment of components

We want to compose a system model of a rotating body driven by an engine. To this end, we want to declare objects of the two classes. The most direct way of doing this, is by an anonymous declaration such as:

```
Body(I := 2.0);

Engine();
```

The anonymous declaration consists in the class name followed by tuple for the parameters. The tuple is mandatory and denoted as modifier. It might, however, be empty.

The declared object may now be assigned to a component declaration. In this way, the total system can be conveniently constructed:

```
define model FlyWheel
  component B := Body(I := 2.0);
  component E := Engine();
implementation
  E.tau + B.tau = 0;
  E.phi = B.phi;
end;
```

The assignment of components is reference-based. This means that if an object is assigned to an identifier, no copy of the object is being created and passed as value but instead the identifier holds now a reference to the object. In this way, two component declarations may point to the same object.

```
define model FlyWheel
  component B := Body(I := 2.0);
  component sameBody := B;
  component E := Engine();
implementation
  E.tau + sameBody.tau = 0;
  E.phi = sameBody.phi;
end;
```

In the example above, `B` is the original instance and `sameBody` is just an alias to it. All designators in the model equations are resolved and refer now to the main instance. The assignment `sameBody := B` represent that there is an alias for `B`.

3.4 Assigning parameters

In the examples above, we could already observe the use of parameters. In fact, parameters are essentially components. There are only a few differences:

- A component declaration must contain an assignment. For a parameter declaration, this is optional.
- Parameters can be assigned in the modifier tuple of an anonymous declaration. Components must not.
- In case a parameter is assigned in the modifier, its original assignment is removed.

The example of section 3.2 shows one example of such a parameterization process. The real number 2.0 is assigned to the parameter `I` in `Body`. The corresponding assignment is now outside the Body object at the system level.

Since parameters are simply components, and model instances can be passed by reference, it is now naturally possible to use parameters also for higher-level modelling task as the exchange of a component. In the following example, the engine is transformed into a parameter of the fly-wheel model:

```
define model FlyWheel
  component B := Body(I := 2.0);
  parameter E as Engine;
implementation
  E.tau + B.tau = 0;
  E.phi = B.phi;
end;
```

The engine model can now be assigned from outside:

```
define model System
  component F := FlyWheel(E:=Engine(T:=1.5))
end;
```

Since Hornblower supports anonymous declaration of components and assigns a fist-class status to them, more complex parameterization can easily be stated :

```
FlyWheel(E:=if Cond then Eng1() else Eng2);
```

3.5 Outer Models

Since components can now be passed around as parameters and in this way, a component can be shared by many sub-models, a very simple concept enables the use of "outer" models.

Using outer models represents a common modelling technique in Modelica (Fritzson, 2004) that is used for global or semi-global models like a gravity model, a signal broadcaster, etc. In Hornblower, this modelling technique is enabled by defining the default value of a parameter as **`outer`**. Let us look at the following example of a Body model that accesses a gravity model outside its scope:

```
define model Body1D
  variable f as Real;
  variable s as Real;
  parameter M := 1.0 as Real;
  parameter w := outer worldModel as World1D;
implementation
  variable v as Real;
  v = der(s);
  f = M*(der(v)-w.gravity);
end;
```

The keyword **`outer`** before the parameter expression indicates that the expression is not resolved within the model but in the scope of its declaration. Hence, when the component `Body1D` is used in a system, the system must declare a component `worldModel` of type `World1D` (Of course, this declaration can be again an outer declaration).

```
define model System
  component B := Body1D();
  component worldModel := World1D();
end;
```

Alternatively, it is still possible to reassign the parameter:

```
define model System
  component B := Body1D(w:=World1D());
end;
```

In this way, outer models can be implemented for virtually no extra cost. In contrast to prior solutions, naming conventions are not necessary and the namespace is not polluted by global entities.

3.6 *Attributes*

Since the parameterization in Hornblower is generic and flexible, it can be used for more than just the main parameters of a model. Practical modelling experience tells us that there is a need to contain a lot more information in a model than just the variables and the equations (Zimmer, 2008). This information concerns for example:

- Hints for the compiler on how to perform index-reduction.
- Additional information for a type (physical units and boundaries).
- Uncertainty modelling.
- The graphic representation of a class, a component, or a system.
- Documentation of a class

In Modelica 3, these issues are typically handled by attributes or annotations. These are either vendor specific or become part of the specification. In Hornblower, it may be appropriate to regard this information as parameters that are loosely attached to their objects. In this case, the normal instantiation process can be used for the handling of annotations, without any large effort.

However, what is meant by "loosely attached parameters" ? These are parameters that can be set but do not have to be set. This is possible since such parameters are ensured to always have a reference to a default object. To this end, these parameters are defined as attributes:

```
define attribute Extent
  parameter  r := {0,0,10,10} as Real[4]
  parameter angle := 0 as Integer
end;
```

An attribute definition is a model definition with no implementation and only parameters in its header section. All declared parameters must be provided with a default

reference. The attribute from above can be used to describe the extent of a component in the GUI of a modelling window. Let us apply this attribute to the component Res2

```
component Res1 := Resistor(R:=100);
component Res2 := Resistor(R:=100)
@Extent(r:= {20,20,40,40});
```

It is possible to access this attribute by using the attribute access operator `@`

```
Res2@Extent.r[2]
```

But the attribute can be also read for `Res1`. The following statement is a valid reading access.

```
Res1@Extent.r[2]
```

Since no "`Extent`" attribute has been attached to `Res1`, the default value of the attribute definition is returned. In this way a safe-reading access is guaranteed for attributes. Another option is to attach an attribute to the corresponding class definition of `Resistor`:

```
define model Resistor
  @Extent(r:= {100,100,60,60});
  [...]
end;
```

In this case, the value is determined by the attribute in the class definition. So attributes are like parameters but with different rules for reading and writing.

For writing the following rule holds:

- An attribute can only be attached by the object's creation (declaration). The attachment is done by the operator `@`.

For reading the following rules hold:

- Attributes can always be read and used like normal parameters.
- If the desired attribute is not attached to the component, the attribute is taken from the corresponding class definition of the object.
- If there is no such attribute attached to the class definition, the default attribute is taken.

Now it is clear, what is meant by loosely attached parameters: These are parameters that can be written if desired, but that always can be read.

By interpreting attributes as annotations, it is possible to reduce the complexity of the modelling language drastically. First, attributes do not need to be included in the language specification (as in Modelica 3). Instead they can be part of the standard library. Second, the user is now enabled to formulate attributes by himself. Third, attributes can be made dependent on other parameters just like normal parameters. All this leads to a simpler language with higher expressiveness.

4. INSTANTIATION PROCESS

Let us now regard the computational realization of the instantiation process. The goal of the instantiation is

- to represent each instance of a model (or port) by an object.
- to evaluate each component and parameter value exactly once (in contrast to components or parameters, variables are not evaluated at all).

The output of the instantiation stage is Hornblower code containing all instantiated objects with their equations and evaluated parameter values.

For instance, the instantiation of the second example in section 3.3 leads to the following code:

```
define object F
  define object B
    […]
  end;

  define object E
    […]
  end;

  component sameBody;

implementation
  sameBody := B;
  B.I := 2.0;
  E.tau + B.tau = 0;
  E.phi = B.phi;
end;
```

An object is a general container for the declaration of components, parameters, variables as well as for the statement of assignments, connections, and equations. Each parameter of an object is determined by exactly one assignment. The equations are contained in their respective objects and their expressions refer to original instances not to aliases anymore. Aliases are still declared as components.

In order to generate this code, objects need to be instantiated. In order to instantiate an object, its parameters must be evaluated. Since the parameters can represent objects themselves, their evaluation is not straight forward.

All objects, components and parameters can be represented by vertices in a directed graph. Each assignment may introduce several dependences between these vertices. These are represented by directed edges (arrows). The assignments must be stated in such a way that the resulting graph is acyclic. If there are cyclic (or recursive) dependences between the parameters, an error is yield.

The resulting directed acyclic graph (DAG) gives rise to a partial order that is required for the instantiation process. In order to determine the correct order of evaluation a topological sorting of the graph is required. However, the standard algorithms for topological sorting cannot be applied since the DAG is not fully available in its complete form at the beginning. The evaluation of parameters may involve the creation of new objects that declare new parameters in turn.

Hence a different, simpler, and more robust algorithm is used for parameter evaluation. We denote it as crawling algorithm:

1) The flag *instantiation Progress* is set to *false*.
2) The crawling algorithm traverses all non-evaluated assignments.
3) An evaluation of each member is attempted during the traversal. Before the actual evaluation takes place, it is checked if all required components or parameters have been evaluated.
4) If the evaluation or instantiation of a single member has been successful, the flag *instantiation Progress* is set to *true*.
5) If the top-level object is instantiated (and that includes all its content), the instantiation has been successful and can be terminated.
6) Else: if *instantiation Progress* is true then repeat from (1)
7) Else: The instantiation has failed due to recursive dependency.

This simple crawling algorithm includes much unnecessary iteration. Evidently, it is possible to improve this algorithm. For instance, by crawling through the classes, a partial topological order can be deduced by assigning integer numbers to each element. At the next iteration, this partial order can then be taken into account.

In practice however, there is probably not a strong need to invest much effort in more sophisticated algorithms. The crawling algorithms is bad if the underlying directed acyclic graph is deep (containing long chains of dependences), but in virtually all relevant examples the graph is broad with many elements of the same topological order. In such graphs, the crawling algorithm performs actually pretty well.

Please note, that this algorithm may not terminate. There are two reasons for this. One, a class may contain a (conditional) recursive declaration of itself. This can lead to an infinitely large model that naturally cannot be instantiated in finite time and memory. An abort will result. Two, the evaluation of expressions may call functions that may be not be proven to terminate for the given set of inputs.

5. IMPLEMENTATION

The language Hornblower is currently implemented in Python 2.7. The implementation concerns the first four stages of Figure 1.

The current implementation consists in roughly 4300 lines of code (including comments and blank lines). Given the fact, that the implementation covers already all relevant translation stages, this shows already a substantial degree of simplification. Please note also that the implementation is self-contained. No “magic” external libraries are used. In fact, even the parser is handwritten (and accounts for 700 lines of code)

In this way, Hornblower can prove the simplicity of its concepts. The coding effort is significantly smaller, although the implementation is not complete yet. The effort undertaken in Hornblower may serve as guideline for future versions of Modelica.

6. CONCLUSIONS

Modelica 3 contains many diverse language constructs: there is a separate syntax for normal parameters and component parameters. In addition, the concept of outer models is part of the language specification. The specification is further enlarged by the description of various annotations.

In Hornblower, all these different concepts can be expressed by a much smaller and consistent set of language constructs. By introducing a reference-based parameter scheme, object-oriented, equation-based languages can be significantly simplified while increasing their level of expressiveness.

The modeller can profit from the flexibility of dynamic typing for the higher-level modelling task. On the other side, the resulting modelling code is statically typed and hence leads to the generation of efficient simulation code. In order to ensure this, Hornblower establishes a strict distinction between the declaration of component and parameters on the one hand side and the declaration of time-varying variables on the other hand side.

We hope that the research undertaken in the Hornblower project may influence the future development of Modelica.

REFERENCES

Burstall, R., Strachey, C (2000). Understanding Programming Languages. *Higher-Order and Symbolic Computation* 13:52,.

Fritzson, P. (2004) *Principles of Object-oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, 897p.

Zimmer, D. (2008) Zimmer, D. (2008) Multi-Aspect Modeling in Equation-Based Languages, *Simulation News Europe*, Volume 18, No. 2, pp. 54-61.

Zimmer, D. (2010) Towards Improved Class Parameterization and Class Generation in Modelica. *Proceedings of the 3rd International Workshop on Equation-Based Object-Oriented Languages and Tools* (EOOLT), Oslo, Norway.