Towards Improved Class Parameterization and Class Generation in Modelica

Dr. Dirk Zimmer

German Aerospace Center, Institute of Robotics and Mechatronics, Germany,
Dirk.Zimmer@dlr.de

Abstract

Class parameterization and class generation enhance the object-oriented means of Modelica, either by making them better accessible for the user or more powerful to apply for the library designer. Nevertheless, the current solution in Modelica does not properly distinguish between these two concepts, and hence it does not represent a fully satisfying solution. This paper presents a proposal or vision for a partial redesign of the language by separating class parameterization and class generation. In this way, the language becomes simpler and yet more powerful. The derived concepts may serve as guideline for future considerations of the Modelica language design.

Keywords ***language design, class-parameterization***

1. Introduction

This paper presents the concepts of class parameterization and class generation for equation-based modeling languages as Modelica. It is highlighted why these concepts are important for a modeling language and how they could be better regarded in the future.

The paper is organized as a proposal for a future design of Modelica. It is instructive in order to be concise. The suggestions are concrete in order to be illustrative. Nevertheless, what finally matters is the abstract idea behind our concept that could as well be finally realized in a different form.

To understand the current situation in Modelica [6,8], the problems of a language designer, and the motivation behind our proposal, let us review the most important fundamentals.

1.1 Processing Scheme

The translation of Modelica models into code for simulation purposes, involves several stages. These are depicted in Figure 1.

The semantics of the language concern nearly every part of this processing scheme. For instance, the causalization of an equation is done in stage 4, whereas the realization of a model extension concerns stage 2. Even with the same syntactic elements, a modeler can formulate expressions that belong to different stages.

An if-branch that depends on a parameter value corresponds to stage 2. If its condition is, however, dependent on a variable then it belongs to stage 4 and 5. In this paper, we are concerned with class parameterization and class generation. These two aspects belong to stage 2 of the processing scheme.

Modelica contains language constructs of all these processing stages in one single layer. This makes the language very powerful and highly convenient. To some degree this style results out of the declarative character of Modelica. It enables the modeler to focus on what he wants to model rather than thinking about how to create a computational realization. In this way, a modeler can achieve his or her goals without being fully aware of the underlying processing scheme.

Nevertheless, this puts an increasingly higher burden on the designers of such a language. Whereas the modeler does not need to know about the processing scheme, a language designer must have a very detailed knowledge. He or she is required to foresee all possible combinations with their potential problems that are introduced by a new language construct. As a language drifts towards higher complexity, this becomes a very hard task.

1.2 Structural Type System

The declarative style of Modelica is supported by a structural type system [1]. This means that the type results solely out of the structure of a class. Roughly speaking, type A is a sub-type of (or compatible to) type B, if all (public) elements of B are declared (by the same identifier) in A, and these elements are themselves sub-types of their counterparts in B.

In a structural type-system, the type is therefore independent from the methods used for its generation, and hence different lines of implementation may lead to compatible types. This is a big strength of structural type systems. Compatible types can have a common ancestor (mostly a partial model), but it is not required.

With respect to class generation and class parameterization, two additional definitions of compatibility must be concerned that impose additional restrictions on the simple sub-type relationship. Plug compatibility requires that, in addition to sub-type compatibility, no further connections are introduced that must be connected from outside. Plug compatibility is required when models get exchanged by class parameterization.

3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools, October, 2010, Oslo, Norway.

EOOLT 2010 website:
http://www.eoolt.org/2010/

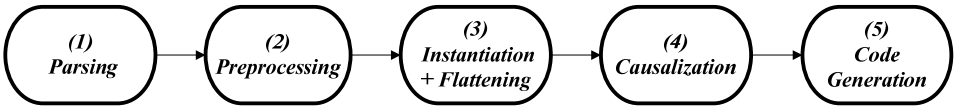

Figure 1: Processing Scheme of a Modelica Translator

Inheritance compatibility means that type A could replace type B as an ancestor for an arbitrary type C. To this end, the sub-type requirements are extended to protected elements. Inheritance compatibility is required for class generation purposes. The relation between these different sub-type relations is depicted in Figure 2.

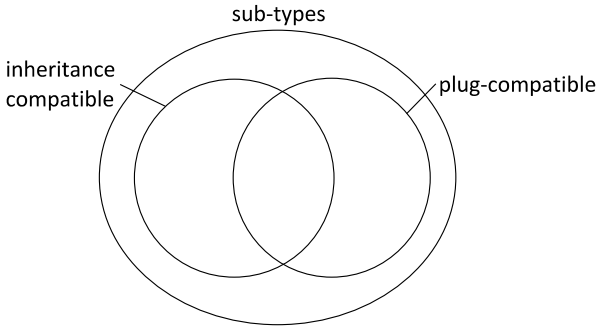

Figure 2. Set relation of different type requirements

1.3 Available Language Constructs

Let us briefly review those language constructs that are to be revised in the future. The use of all these keywords is then demonstrated by means of examples in the next section.

1.3.1 `replaceable` and `redeclare`

A modeler can declare a component B of model M as **`replaceable`**. By doing so, this component can be replaced, either in a possible extension of the model M or by a modifier that is applied to an instance M.

In order to replace the component B, the keyword **`redeclare`** (**`redeclare replaceable`** resp.) has to be applied. A new component A is then put into the place of B.

The type of the component A can be further constrained with the keyword **`constrainedby`**. It is applied at the original declaration that was marked as replaceable.

1.3.2 Parameters for classes

The keyword **`replaceable`** cannot only be applied to the declaration of components but also to the definition of **`models`**, **`packages`**, **`records`**, etc. To this end, the term replaceable model (or package, record, …) has been introduced.

Such definitions can then be extended by the use of the term **`redeclare model`** or (**`redeclare replaceable model`** resp.). Also the replacement of definitions can be constrained by the keyword **`constrainedby`**.

It is in general not possible to extend from replaceable model definitions. An exception is enabled by the term **`redeclare [replaceable] model extends`**.

1.3.3 Conditional Declarations

In addition to these tools, there are also conditional declarations available in Modelica. To this end, a short if-statement is appended to the normal declaration of a component. It is, however, not possible to combine conditional declarations with replaceable components or with components that are being redeclared.

2. The Important Difference between Class Generation and Class Parameterization

The presented language elements in Modelica may now serve two entirely distinct purposes: class parameterization and class generation. It is very important to make a proper distinction between these two concepts, since the lack of this distinction is the root of the current problems in Modelica. In order to clarify the situation, we present a representative set of examples for both concepts.

2.1 Examples for Class Parameterization

Class parameterization means that a class itself or a component is a parameter.

Class parameterization with respect to Modelica does mostly mean, model parameterization. To this end, a sub-component is made exchangeable by means of the parameter menu. Let us review three typical examples of this process.

2.1.1 Container Model (Wheels and Tires)

The container model is one of the most primitive methods to achieve class-parameterization. Essentially, it represents a set of conditionally declared components. Given a parameter value (mostly an enumeration value), one of the conditions evaluates to true, whereas all other components are disabled.

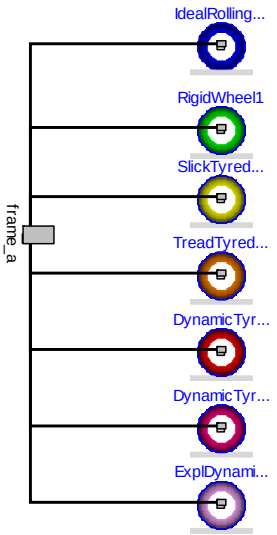

Figure 3. Container model for different wheel models

Figure 3 presents a container model that enables switching between different wheel models. The model parameterization is done indirectly by transforming a regular parameter into the conditional declaration of sub-models.

```
model MultiLevelWheel
public
  parameter ModLevels level //enumeration
  Interfaces.Frame_a frame_a;
  …
  IdealWheel wheel1(...)
    if level == ModLevels.IdealWheel;
  RigidWheel wheel2(...)
    if level == ModLevels.RigidWheel;
  SlickTyredWheel wheel3(...)
    if level == ModLevels.SlickTyredWheel;
    …
equation
  connect(wheel1.frame_a, frame_a);
  connect(wheel2.frame_a, frame_a);
  connect(wheel3.frame_a, frame_a);
  …
end MultiLevelWheel;
```

Given the construct of **replaceable**/**redeclare**, this design pattern has actually become redundant. It is, however still applied. It is better suited if the sub-models shall not be public but protected. Another application results, if the standard dropdown list (of the Dymola GUI) for replaceable components is not the preferred parameterization since another user interface is demanded.

2.2 Exchangeable Resistor Model

The standard method of model parameterization is performed by means of a replaceable model. An electric circuit may contain a resistor component. If it is declared as replaceable:

```
model Circuit1
  replaceable Resistor R1(R=100);
  …
end Circuit1;
```

A potential user of this circuit model may now exchange the resistor

```
model Test
  Circuit1 C(
    redeclare ThermoRes R1(R=100)
   );
   …
end Test;
```

If the circuit contains two resistors, each can be redeclared separately. Alternatively, the circuit can have a parameter for the model definition.

```
model Circuit2
  replaceable model R = Resistor(R=100);
  R R1;
  R R2;
  …
end Circuit2
```

A user can now redefine the model definition:

```
model Test
  Circuit2 C(
    redeclare model R = ThermoRes(R=100)
  );
   …
end Test
```

2.2.1 Media-Exchange

Having parameters for class definitions enables more advanced modeling techniques. The models of the Modelica Fluid [4,5] library serve as a good example. Here each fluid model contains a parameter for a package definition. Given this package, the model declares now those package members that it requires.

```
model TemperatureSensor
  replaceable package Medium =
    Interfaces.PartialMedium;
  Interfaces.FluidPort_in port(
    redeclare package Medium =
      Medium
    )
  Medium.BaseProperties medium;
  Modelica.Blocks.Interfaces.RealOutput
T(unit="K");

equation
  …
  port.p = medium.p;
  port.h = medium.h;
  port.Xi = medium.Xi;
  T = medium.T;
end Temperature;
```

2.3 Examples for Class Generation

Class Generation is a collective term for all those methods that are used to generate a new class. Most commonly, the new class is created out of one or more existing ones.

The most common technique of class-generation in Modelica is class extension that is represented by the keyword extends.

Mostly, replaceable and redeclare are used for class parameterization, but there are also applications for class generation. The following two examples shall demonstrate this.

2.3.1 MultiBondLib

The MultiBondLib [11] features various mechanical libraries based on the bond-graphic modeling methodology. There is the planar mechanical library and the 3D-mechanical library. In addition, there is the 3D-mechanical library that includes the modeling of force-impulses. This library was derived from its continuous-system version. To this end, the connector of the classic mechanical package was made replaceable.

```
connector Frame
  Potentials P;
  flow SI.Force f[3];
  flow SI.Torque t[3];
```

```
end Frame;

model FixedTranslation
  replaceable Interfaces.Frame_a frame_a;
  replaceable Interfaces.Frame_b frame_b;
  …
end FixedTranslation;
```

The connector of the impulse library was then extended from its continuous version.

```
connector IFrame
  extends Mech3D.Interfaces.Frame;
  Boolean contact;
  SI.Velocity Vm[3];
  SI.AngularVelocity Wm[3];
  flow SI.Impulse F[3];
  flow SI.AngularImpulse T[3];
end IFrame;
```

Finally, each component of the impulse-library was inherited from its continuous counterparts, had its connectors replaced and the required impulse equations added:

```
model FixedTranslation
  extends Mech3D.Parts.FixedTranslation(
    redeclare Interfaces.IFrame_a frame_a,
    redeclare Interfaces.IFrame_b frame_b
   );
  …
equation
  …
  frame_a.contact = frame_b.contact;
  frame_a.F + frame_b.F = zeros(3);
  frame_a.T + frame_b.T +
    cross(r,R*frame_b.F) = zeros(3);
  frame_a.Vm + ( transpose(R) *
    cross(frame_a.Wm,r) ) = frame_b.Vm;
  frame_a.Wm = frame_b.Wm;
end FixedTranslation;
```

Making the connector directly replaceable is not the preferred solution given the current means of the language. It would be better to use a model parameter C (via replaceable model) for the connectors and declare the connectors by the use of C. At its time of creation, however, this solution was not available for the MultiBondLib.

2.3.2 Medium equations in the MediaLib

Another example for class generation can be found in the Modelica MediaLib. Here, an individual package is created for each medium. Among other members the package contains a model `BaseProperties` that describes those balance equations that are specific to the medium (e.g. the ideal gas law).

A new medium may now inherit from an existing medium package and redefine its `BaseProperties` model. In this way a class is generated for each medium:

```
partial package SingleGasNasa
  extends PartialPureSubstance(…)
  redeclare model extends BaseProperties(…)
equation
  …
  MM = data.MM;
  R = data.R;
  h =h_T(data, T, h_offset);
  u = h - R*T;
  d = p/(R*T);
  state.T = T;
  state.p = p;
end BaseProperties;
```

2.4 Foresight versus Hindsight

Since both, class parameterization and class generation are performed during the preprocessing stage in the translation process, it may be tempting to use one set of tools for both purposes as is done in Modelica. However, this turns out to be problematic because of the entirely distinct motivation behind these two concepts.

Class parameterization is requested by the model designer to be performed by a user of its library. Thus, it is performed in *foresight* since the corresponding parameterization needs to be declared. Rules for class parameterization must be rather strict to prohibit abuses by the user to a meaningful extent.

In contrast, class generation is done in *hindsight*. It is performed by the model-designer and requested from a previous library. Since it is done in hindsight and mostly performed by experts, rules for class generation should not be prohibitive. It is not possible to foresee which models might be extended; so a potential keyword **`extendable`** does not make much sense. It is, however, also not foreseeable which elements might be **`redeclared`**; so the keyword **`replaceable`** is inappropriate. Prohibitive measures will tend rather to corrupt existing classes than to prevent the faulty creation of new classes.

2.5 Different Aspects of the Type System

Another vital difference between class parameterization and class generation is highlighted by the criteria of the type system that are relevant for each concept.

A proper class parameterization requires that the new type A is compatible to the original type B. Obviously A must be a sub-type of B. An even more strict requirement is that it needs to be plug-compatible since it is not possible (and certainly not convenient) to introduce new connections into a parameterized model.

Plug-compatibility is of no relevance for class generation. When a new class is generated, new connections can also be introduced in an effortless way. Instead, it is important that the new type is inheritance-compatible since potential extensions of a redefined model ought to remain valid.

Evidently, separate aspects of the type system need to be concerned for both tasks.

2.6 Current Deficiencies

The confusion of class parameterization and class generation involves a number of disadvantages:

- *Non-uniform parameterization*: The syntax that has been chosen for class parameterization purposes is different to those of normal parameters. One unfortunate consequence of this decision is that class

parameterization becomes inaccessible for normal parameter computations. For instance, it is not possible to combine if-statements with redeclarations. This means that redeclarations cannot be coupled to conditions.

- *Inappropriate sub-elements*: Since model parameters are not properly declared as parameters but more as a replaceable sub-element this leads to inappropriate structures. For instance, models that contain sub-packages. It makes sense that a model cannot contain a package, but it makes no sense that a model can contain a package just because it is replaceable.
- *Prohibitive class generation*: Since potential redeclarations and redefinitions must be marked as replaceable in advance, the options for class generation are unnecessarily limited. Often this leads to an ex post modification of the original library in order to enable the desired class generation
- *Unwanted parameterization*: Since potential redeclarations or definitions for the purpose of class generation need to be marked as replaceable an unwanted parameterization is introduced into the models. In order to avoid this, the replaceable objects are often moved to the protected section.
- *Unnecessary restriction*: To extend from replaceable model definitions is currently prohibited in Modelica (with one exception). This restriction will turn out to be unnecessary.
- *Overelaborated syntax*: The current syntax is simply more complicated than actually necessary and can be simplified.

3. Design Decisions

For the partial redesign of Modelica, we establish the following guidelines:

- Separate class parameterization and class generation

  We want to clearly separate class parameterization from class generation. In this way, the specific needs and motivation of each concept can be optimally taken into account.

- Give classes first class status on the parameter level

  We want to treat class parameters just as any normal parameter. There is no reason why parameters should be restricted to base-types or quasi base-types. This will simplify and unify the syntax. Furthermore, class parameterization can be integrated in the normal computation process for parameters.

- Enable non-prohibitive class generation

  Class generation shall be performed by a special subset of keywords. It shall be designed in a way that it is not hampered or prohibited by means that require foresight. Maximum freedom should be given to the modeler in order to create new classes. On the other hand side, existing classes shall be protected from corruption.

- Unify and simplify the language

  The complete language should be simpler and more powerful after the revision. It should also be more intuitive to understand and to learn.

4. Improved Class Parameterization

In this section, we will propose new language constructs for class parameterization. In order to show their potential applications and highlight their advantages, we will review the examples of chapter 2.1.

4.1 Unification of Expression

In a first, preparatory step, we integrate the expression of classes into normal statements. To this end, we have to slightly change the modifier syntax of an expression: the modification is now applied in curly brackets instead of round parentheses.

This change has been implemented in order to make a class-definition with its modifier distinguishable from a function-call with its argument-list. In this way you know that `foo(x=a)` represents a function call but `bar{x=a}` represents a class-definition with its modifier. The term `baz{p=b}(a)` represents then consequently a parameterized function call.

Since classes can be used in expressions, the language power is increased, e.g., by using classes in if-clauses or as arrays:

```
// The result of this if-clause is a class
if expr then foo2{x2=b} else foo2{x2=c};

// An array of 4 classes
 foo2[4]
 foo2{x2=a}[4]
```

One might hesitate, to integrate class-expressions as basic part of normal expressions, since this gives classes a first-class status [2,3] and opens up the grammar quite substantially. It might seem smarter (and easier to achieve) to form two separate kinds of expressions that are distinguished on the top level: normal expressions and class expressions. However, this is misleading for the following reasons.

Firstly, normal expressions and class expressions can both start with a name. This means that an undefined number of look-up tokens are required to distinguish these two kinds of expressions. Practically this means that an extra keyword is needed, but this leads to an ugly and unpractical syntax. Also, many syntax elements would need to be doubled and still two kinds of grammars would be required. Hence such a solution would not be fully generic.

`A1.B2.C3.Model{…}` → class expression
`A1.B2.C3.Function(…)` → normal expression

Secondly, the integration of class-expressions into normal expressions provides an important generalization for future language extensions. Whereas many syntactic formulations such as `foo{x2=3} + foo{x2=2}` are semantically still invalid for this proposal, this may change in future revisions. Let us envision a future

version of Modelica (5 or 6) that enables anonymous declarations of models or records. Then, the former statement `foo{x2=3} + foo{x2=2}` may become valid if, for instance, `foo` is a record and overloads the + operator. Hence the integration of class-expressions opens up a number of fruitful opportunities for future language revisions. It is notable that the first-class status of higher-level language constructs is absolutely common in contemporary programming languages. Even a few equation-based modeling languages (Sol [9,10], Hydra [7], Modeling Kernel Language [2]) have explored this important topic.

In this proposal, only the following uses of class-expressions shall be semantically supported. All other uses yield error messages.

- Pure class expressions: `foo{x=y}`
- Class expressions in if-statements: **`if`** a **`then`** `foo{x=y}` **`else`** `bar{x=y}`
- Array-lists of class-expressions: `{foo{x=y}, bar{x=y}, …}`

4.2 Say It As You Want It: Treat Component Parameters as Normal Parameters

If a component (let us suppose: a resistor) shall be a parameter of a model, it is the most natural thing, just to write it down as a normal parameter. Instead of the awkward formulation:

```
model Circuit1
  replaceable ThermoRes R1(k=0.5)
  constrainedby Resistor(R=100);
  …
end Circuit1
```

simply write it as a component parameter:

```
model Circuit1
  parameter component Resistor R1{R=100} =
ThermoRes{k=0.5}
  …
end Circuit1
```

A user of this circuit model may now give a new parameter value and thereby replace the prior model.

```
model Test
  Circuit1 C{R1 = ThermoRes{k=1.2}};
  …
end Test
```

The type of the parameter hereby represents the constraint type for the parameterized model. Naturally one can apply modifiers also on the constraint type, and of course the new resistor type must be plug-compatible to this constraint type.

The keyword **`component`** is necessary in order to avoid potential ambiguities. These originate from the fact that the formulation of a component parameter is a language construct that performs two tasks at the same time. One, it enables direct class-parameterization of a component. Two, it declares a component that invokes an instance.

Hence it must be clarified if the = operator assigns a value

```
parameter Real r = 1;
```

or a component (sub-model)

```
parameter component Resistor R = ThermoRes;
```

In these cases, the meaning is clear, but when records are concerned both interpretations of the assignment are meaningful:

```
//value assignment
parameter Complex c1 = Complex.j();

//component assignment
parameter component Complex c2 =
                        Quaternion;
```

In fact, the keyword does not just change the interpretation of the assignment, but also if the values of the instance shall be constant or not. In the example above, `c1` is constant-valued but `c2` may express variable values.

4.3 Say It As Want You Want It: Treat Class Parameters as Normal Parameters

The very same can be done for parameters that identify class definitions, such as model parameters or package parameters. Again, the best solution is to simply write it down as one wants it to have. So, instead of writing:

```
replaceable model R1 = Resistor{R=100}
  constrainedby Resistor;
```

you can simply turn the model into a parameter:

```
parameter model Resistor R1 =
                  Resistor{R=100};
```

Since such parameters will ultimately always be used for class parameterization, plug-compatibility shall also be required here. In this way, the temperature sensor of section 2.1 could be formulated as follows:

```
model TemperatureSensor
 parameter package
  Interfaces.PartialMedium Medium;

  Interfaces.FluidPort_in port{
    Medium = Medium}
  Medium.BaseProperties medium;
  Blocks.Interfaces.RealOutput T{unit="K"};
equation
  …
end Temperature;
```

4.4 Improved Computational Power

One obvious advantage is that the language has been unified. Now, the same notation is used for all kinds of parameters. It has also become simpler. The keywords **`replaceable`**, **`redeclare`** and **`constrainedby`** are not needed any more.

Another major advantage is that class parameters can be computed with as any other parameter. In this way,

conditional declarations become redundant in many cases. Let us review the example of the container model. Here we had to transform an enumeration into a class. This was done by number of conditional declarations. Now, we have the option to use an array of model parameters for this purpose.

```
model MultiLevelWheel
public
  parameter TModLevels level //enumeration
  Interfaces.Frame_a frame_a;
  …
protected
  final parameter model BaseWheel
    wheelModels[7]= { IdealWheel {…},
                      RigidWheel{…},
                      SlickTyredWheel{…},
                      … };
  final parameter component BaseWheel wheel
    = wheelModels[level];
equation
  connect(wheel.frame_a, frame_a);
end MultiLevelWheel;
```

Here we can organize different model parameters in an array. In the same way, this could be done in a record. It is important to notice that class parameters become accessible to all kinds of computations. Especially useful is the conditional evaluation:

```
parameter Boolean constantTemp = true;
final parameter BaseTempModel
ambientTemperature =
  if constantTemp then ConstTempModel{…}
  else TempFileHistory{…};
```

One inconvenience of the proposed notation is that it sometimes leads to redundant formulation. In some applications, the default parameter value will equal the type constraint.

```
parameter component Resistor R1{R=100} =
                    Resistor{R=100};
```

Here, `Resistor{R=100}` had to occur twice. If this turns out to be a frequent case, one may consider adding a new keyword **`itself`** in order to provide some syntactic sugar.

```
parameter component Resistor R1{R=100} =
                    itself;
```

5. Improved Class Generation

Having available powerful and well-integrated means for class parameterization, we can now provide separate means for class generation. To this end, we need to keep our eye on two different targets:

1. Enable the convenient creation of new classes out of existing classes.
2. Prevent the corruption of existing classes.

The second goal is easily forgotten, but it is equally important to the first goal. Again, we explain the new language constructs by means of examples and review for this purpose the models from section 2.2.

5.1 The New Role of Redeclared

We replace the former keyword **`redeclare`** by a new keyword **`redeclared`**. The new keyword is now solely implemented for the purpose of class generation. It can actually be applied similar to the former keyword. Let us therefore review the example of the mechanical impulse library where we wanted to exchange the continuous-system connector with an extended counterpart.

```
model FixedTranslation
  extends
Mechanics3D.Parts.FixedTranslation;
  redeclared Interfaces.IFrame_a frame_a;
  redeclared Interfaces.IFrame_b frame_b;
  …
equation
  …
  frame_a.contact = frame_b.contact;
  frame_a.F + frame_b.F = zeros(3);
  frame_a.T       +       frame_b.T       +
cross(r,R*frame_b.F) = zeros(3);
  frame_a.Vm
  + transpose(R)*cross(frame_a.Wm,r)
  = frame_b.Vm;
  frame_a.Wm = frame_b.Wm;
end FixedTranslation;
```

This solution is very similar to the existing methods in Modelica, but there are crucial and important differences. Most importantly, the elements are not **`redeclared`** in the modifier of the extension belonging to the existing class, but in the public section of the new class. In this way, we prevent the existing class from being corrupted and we prohibit an abuse of the keyword **`redeclared`** for the purpose of class parameterization. For this reason, the use of **`redeclared`** in modifiers is strictly forbidden.

Furthermore the **`redeclaration`** can be applied to all inherited components without restriction. It is not necessary (nor desirable) to mark these components as replaceable beforehand in the inherited models. Doing so would be not even superfluous but even harmful since...

1. it would require an inappropriate amount of foresight.
2. it is very tempting to add the **`replaceable`** keyword ex post and thereby to corrupt the original models that should not be touched
3. the **`replaceable`** keyword actually introduces an unwanted parameterization of the original model.

Hence the original translation model can be formulated just normally without replaceable connectors.

```
model FixedTranslation
  Interfaces.Frame_a frame_a;
  Interfaces.Frame_b frame_b;
    …
end FixedTranslation;
```

Also the class requirements that are imposed on redeclarations are different. For class parameterization the replaced models must have been plug-compatible to the constraint or the original model, respectively. This was necessary since additional connections could not be introduced anymore. In the case of class generation, it would be easy to add a new connection in the new model, and hence the only requirement is that the **redeclared** model is a sub-type of the original type. Plug-compatibility is not requested anymore.

5.2 The New Role of Redefined

Another application of class generation is the redefinition of whole models, packages, etc. To this end, the keyword redefined is provided. When class definitions are inherited (for instance by inheriting a package), any definition can be redefined. To clarify this, let us review the example of the MediaLib.

```
partial package SingleGasNasa
  extends PartialPureSubstance{…};

  redefined model BaseProperties{…}
    extends itself;
  equation
    …
  end BaseProperties;

end SingleGasNasa;
```

In principle, not much has changed on the syntax level. Nevertheless, there are again important differences to the prior solution. First of all, the original `BaseProperties` model did not need to be marked as replaceable. The reasons for this are exactly the same as for redeclared components. Correspondingly, the use of redefined is also banned from modifiers.

Second, the use of **redefined** on class definitions imposes different type restrictions than the use of **redeclared** on components. Since class definitions might get extended within the inherited package, the redefined type must be inheritance-compatible to the original type. It is still possible that there are conflicts for inheritance since the redefined class may generate a sub-type that leads to name clashes. However, these type of errors can be fairly well reported and it is a rather uncommon situation.

Since, now the proper type restrictions are applied (and not the inappropriate plug-compatibility), the redefinition of base-classes is enabled. In fact, this can be a powerful design tool. Let us consider once more the mechanical impulse library of the MultiBondLib: instead of redeclaring the connectors in each component, it would be far more elegant, to extend the whole package and redefine the connector. Then all components adapt automatically and the missing equations can be added to each component by providing an extended redefinition of itself.

```
package Mechanics3D;

  connector Frame
    Potentials P;
    flow SI.Force f[3];
    flow SI.Torque t[3];
  end Frame;

  connector Frame_a extends Frame;
      …
  end Frame_a:

  model FixedTranslation
    Interfaces.Frame_a frame_a;
    Interfaces.Frame_b frame_b;
    …
  end FixedTranslation;

  …
end Mechanics3D;

package Mechanics3DwithImpulses;
  extends Mechanics3D;

  redefined connector Frame extends itself
    Boolean contact;
    SI.Velocity Vm[3];
    SI.AngularVelocity Wm[3];
    flow SI.Impulse F[3];
    flow SI.AngularImpulse T[3];
  end Frame;

//The extended model Frame_a will automatically adapt and does not need to be redefined.

  redefined model FixedTranslation
  //Here the connectors do not need to be  redeclared
  equation
    …
    frame_a.contact = frame_b.contact;
    frame_a.F + frame_b.F = zeros(3);
    frame_a.T + frame_b.T +
      cross(r,R*frame_b.F) = zeros(3);
    frame_a.Vm + (transpose(R)*
      cross(frame_a.Wm,r))= frame_b.Vm;
    frame_a.Wm = frame_b.Wm;
  end FixedTranslation;
  …

end Mechanics3DwithImpulses;
```

6. Final Review

Let us quickly review the proposed modifications of the language.

6.1 Simplification of the Language

The grammar has become simpler and more unified (see appendix). The keywords **replaceable** and **constrainedby** have become obsolete. Non-uniform and complicated construction as: **redeclare replaceable model extends** can also be removed from the language. The keyword **redeclare** is replaced by **redeclared** or **redefined** respectively that have a different meaning. These new keywords are also removed from the modifiers, which simplifies the grammar.

6.2 Higher Degree of Expressiveness

The unification of class parameters and normal parameters not only simplifies the grammar and makes the language more intuitive to understand. It also improves the expressiveness of the language. Now we can compute with class parameters just as with normal parameters and create all kinds of models.

The separation of class generation from class parameterization helps to protect existing classes from the introduction of unwanted parameterization. Since class generation is now applicable to all components (but only in a new class), less foresight is required and more can be done in hindsight without having to modify the original models. This separation also helps to impose the correct class requirements for each operation.

6.3 Deficiencies of this proposal

Syntactically, the introduction of the keyword **`itself`** is regrettable. Here the former notation was more convenient. However, the new grammar enables to formulate a component or class parameter without a default value and, in this way, to enforce a parameterization in an evident manner. This is not possible in the current grammar. Also, the keyword **`itself`** can be reused in the extends-clause and here it leads to a more natural and better understandable expression.

Semantically, the new notation almost completely covers the expressiveness of the current Modelica. Only for the redefinition of classes, there exists no short notation. For instance, if the redefinition of a model occurs in a modifier of a replaceable class, then we have a problem.

```
replaceable package Medium =
  PureSubstance(redeclare model
    BaseProperties = myProperties
  )
```

The new language version (maybe rightfully) prohibits such ad-hoc class-generations. To this end, we have to create a new class and assign it to the model parameter.

```
partial package newMedium
  extends PureSubstance;
  redefined  model BaseProperties
   = myProperties;
  …
end newMedium;

package parameter Medium = newMedium;
```

Please keep in mind: this is only the case for this specific kind of redefinitions. Most of the current redeclarations get replaced by parameter assignments and these are totally uncritical. Hence, this is a rather uncommon case and since such a transformation is better implemented manually. To our knowledge, this application does hardly occur.

6.4 Backward Compatibility

Backward compatibility is major issue since this proposal would definitely represent a drastic change of the Modelica language. Unfortunately, it is not easy to achieve. Our proposal distinguishes class generation from class parameterization. This was not done before. Hence, one needs to separate what is currently intermixed. It is possible to do so for 90% of all occurring cases but there remain, inevitably, some cases that cannot be resolved automatically.

6.5 Final Conclusions

The main two points of this paper are:

- It is highly meaningful to distinguish class generation from class parameterization since entirely different motivations are underlying these two concepts.
- Introducing class expressions (and thereby giving classes a first-class status) can drastically simplify the grammar while making the language more powerful. Class parameterization is only one possible application of class expressions.

Appendix

These are the resulting grammar changes to the Modelica language. Please note, this represents not our exact proposal. We provide this just in order to show the simplifications and to concretize the conceptual explanations of this paper.

The following keywords are removed from the language:

`replaceable`
`constrainedby`
`redeclare`

The following keywords are introduced into the language**:**

`component`
`redeclared`
`redefined`
`itself`

The following grammar changes are listed according to the order of the language specification. New elements are underlined, removed elements are ~~scratched.~~

B 2.2 Class Definition

```
element:
       import_clause |
       extends_clause |
       ~~[ redeclare ]~~
       [ redeclared ]
       [ final ]
       [ inner ] [ outer ]
       (( class_definition |
       component_clause ) |
       ~~replaceable ( class_definition |~~
       ~~component_clause)~~
         [~~constraining_clause~~ comment])
```

B 2.3 Extends

```
extends_clause :
```

```
      extends (name | itself)
     [ class_modifications ] [annotation]

contraining_clause:
      constrainedby name
[class_modification ]
```

B 2.4 Component Clause

```
type_prefix:
      [flow | stream]
      [discrete | (parameter  [par-
specifier]) | constant ]
      [input | output]
par_specifier:
      (component | class | model | record
| block | connector | type |
       package | function | operator |
operator function | operator record)
```

B 2.5 Modification

```
modification:
      class_modification ["="
(expression|itself) ]
      | "=" (expression|itself)
      | ":=" expression

class modification :
      "{" element_modification {"," 
element_modification }  "}"

argument_list:
      argument {“,” argument}

argument :
      element_modification_or_replaceable
      element_redeclaration

element_modification_or_replaceable
      [ each ] [ final ] (element
modification | element_replaceable)

element_modification:
      [ each ] [ final ] name [
modification ] string_comment

element_redeclaration :
      redeclare [ each ] [ final ]
      ( ( class_definition |
component_clause) | element_replaceable )

element_replaceable:
      replaceable (class definition |
component_clause1)
      [constraining clause ]

component_clause1:
      type_prefix type_specifier
component_declaration1

component_declaration1 :
      declaration comment
```

B 2.7 Expressions

```
primary:
      UNSIGNED_NUMBER
      | STRING
      | false | true
      | class_expression
      | (der | initial)
(function_call_args)
      | component reference
      | "(" output_expression_list } ")"
      | "[" expression_list { ";"
expression_list } "]"
      | "{" function_arguments "}"
      | end

class_expression:
 name [class_modification]
 [(function_call_args)]
```

Acknowledgements

I’d like to thank Martin Otter for the enjoyable and fruitful discussions on this topic and for further suggestions. I also like to thank Hans Olsson for further critical remarks.

References

[1] David Broman, P. Fritzson, S. Furic. Types in the Modelica Language. *Proceedings of the Fifth International Modelica Conference*, Vienna, Austria Vol. 1, 303-315, 2006.

[2] David Broman and Peter Fritzson Higher-Order Acausal Models *Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT)*, Paphos, Cyprus, 2008.

[3] Rod Burstall, Christopher Strachey. Understanding Programming Languages. *Higher-Order and Symbolic Computation* 13:52, 2000.

[4] Hilding Elmquist, H. Tummescheit, Martin Otter. Object-Oriented Modeling of Thermo-Fluid Systems. *Proceedings of the 3rd Modelica Conference*, pp 269-286, 2003.

[5] Rüdiger Franke, F. Casella, M. Otter, M. Sielemann, S.E. Mattson, H. Olsson, H. Elmquist. An Extension of Modelica for Device-Oriented Modeling of Convective Transport Phenomena. *Proc. 7th International Modelica Conference*, Como, Italy, 2009.

[6] Peter Fritzson. *Principles of Object-oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons, 897p. 2004

[7] George Giorgidze and Henrik Nilsson. Higher-Order Non-Causal Modelling and Simulation of Structurally Dynamic Systems. In: *Proceedings of the 7th International Modelica Conference*, pp. 208 - 218, Como, Italy, 2009.

[8] Modelica® - *A Unified Object-Oriented Language for Physical Systems Modeling*. Language Specification, Version 3.2, www.modelica.org.

[9] Dirk Zimmer. *Equation-Based Modeling of Variable-Structure Systems.* PhD-Dissertation, ETH Zurich, 2010.

[10] Dirk Zimmer. Introducing Sol: A General Methodology for Equation-Based Modeling of Variable-Structure Systems In: *Proc. 6th International Modelica Conference*, Bielefeld, Germany, Vol.1, 47-56, 2008.

[11] Dirk Zimmer and F.E. Cellier, The Modelica Multi-bond Graph Library. In: *Simulation News Europe*, Volume 17, No. 3/4, pp. 5-13, 2007.