Towards Improved Class Parameterization and Class Generation in Modelica

October 3, 2010

Dr. Dirk Zimmer

German Aerospace Center (DLR),
Institute of Robotics and Mechatrnics

EOOLT 2010
3rd International Workshop on Equation-Based
Object-Oriented Modeling Languages and Tools

Motivation

Institute of Robotics and Mechatronics

This presentation presents a proposal for a change in the Modelica language.

- The corresponding paper represents one of the first versions of this proposal.
- Here, the focus was to elaborate new ideas and not let ourselves hamper by political or technical restrictions (e.g. backward-compatibility).

Motivation

Later on, we (Martin Otter and me) have changed our proposal w.r.t Modelica 3 in preparation of the Modelica-Design meeting (held in Atlanta this September).

- We improved backward compatibility.
- We reduced the amounts of changes in the syntax.
- We updated many examples, etc.
- After all, the proposal changed quite a lot.

- But I still decided to present the original version of the proposal here, because I think that it points out the original ideas way better than the newer versions that includes many concessions and compromises.

Motivation

Institute of Robotics and Mechatronics

This turned out to be a good idea…

- …since one of the first things that was decided at the Modelica design meeting was to start a new development branch for Modelica 4.
- Modelica 4 should be a complete new language, essentially designed from scratch.
- So all the concessions we made, turned out to be irrelevant.
- Also the design-decisions w.r.t syntax are less relevant. Of importance are the main ideas behind the concept.

The Origin of Evil

Institute of Robotics and Mechatronics

Many users of Modelica are annoyed by the tedious handling of class parameterization in Modelica.

- The main problem originates from the fact that a single tool is used for two entirely distinct concepts.
- These are **class parameterization** and **class generation**.
- Let us look at the current applications of these two concepts within Modelica.

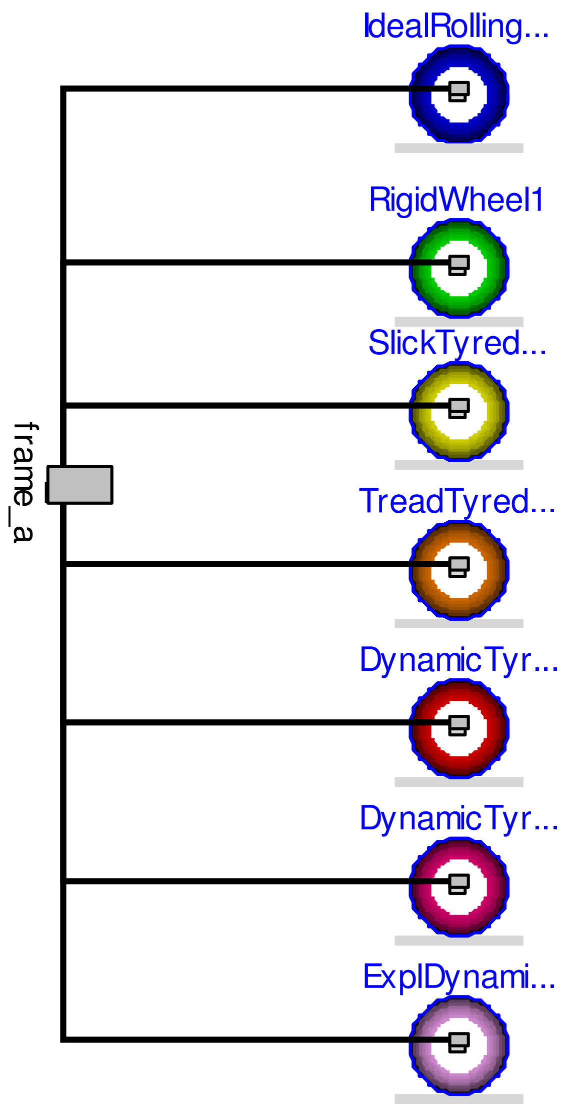
Class Parameterization

Institute of Robotics and Mechatronics

Class parameterization means that a class itself or a component is a parameter (of the model)

Example 1 for C$_{lass}$P$_{arameterization}$

DLR
Institute of Robotics and Mechatronics



- This picture 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.

Example 1 for C_{lass} P_{arameterization}

Institute of Robotics and Mechatronics

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

equation
  connect(wheel1.frame_a, frame_a);
  connect(wheel2.frame_a, frame_a);
  connect(wheel3.frame_a, frame_a);
  …


end MultiLevelWheel;
```

- 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.

Example 2 for $C_{lass}P_{arameterization}$

Institute of Robotics and Mechatronics

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




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

- Given the construct of replaceable / redeclare, this design pattern has actually become redundant.
- The standard method of model parameterization is performed by means of a replaceable model.
- An electric circuit may contain a replaceable resistor component.
- A potential user of this circuit model may now exchange the resistor.

Example 3 for C_{lass} P_{arameterization}

Institute of Robotics and Mechatronics

```
model TemperatureSensor
  replaceable package Medium =
    Interfaces.PartialMedium;

  Interfaces.FluidPort_in port
  (
    redeclare package Medium =
      Medium
  )

  Medium.BaseProperties medium;

  RealOutput T(unit="K");

equation
  …
  port.p = medium.p;
  port.h = medium.h;
  port.Xi = medium.Xi;
  T = medium.T;

end Temperature;
```

- Having parameters for class definitions enables more advanced modeling techniques.
- The models of the Modelica Fluid 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.

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.

Example 1 for C_{lass}G_{eneration}


```
package PartialPureSubstance
 replaceable model BaseProperties
   …

end PartialPureSubstance;

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;
  …
end SingleGasNasa;
```

- Another example for class generation can be found in the Modelica MediaLib.
- Here, an individual package is created for each medium.
- the package contains a model BaseProperties that describes the medium-specific equations
- 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

Example 2 for C_{lass}G_{eneration}


```
package Mech3D

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


  model FixedTranslation
    replaceable Frame_a frame_a;
    replaceable Frame_b frame_b;
    …
  end FixedTranslation;


end Mech3D;
```

- The MultiBondLib features various mechanical libraries.
- In addition to the continuous 3D-mechanical , there is the 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.

Example 2 for C$_{lass}$G$_{eneration}$


```
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;


model FixedTranslation
  extends Mech3D. FixedTranslation(
    redeclare IFrame_a frame_a,
    redeclare 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);
  …
end FixedTranslation;
```

- The connector of the impulse library was then extended from its continuous version.
- Finally, each component of the impulse-library was inherited from its continuous counterparts.
- Their had their connectors replaced and the required impulse equations added.
- The drawback is that all connectors of the continuous version appear in the parameter window.

The two concepts

Class Parameterization

vs.

Class Generation

Foresight vs. Hindsight

- 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.

Foresight vs. Hindsight

Institute of Robotics and Mechatronics

- It is performed by the model designer and requested to from an existing library.

  ➔ In contrast, class generation is done **in hindsight.**

- Since it is done in hindsight and mostly performed by experts, rules for class generation should not be prohibitive.

- A use of replaceable is not meaningful and represents an unwanted parameterization.

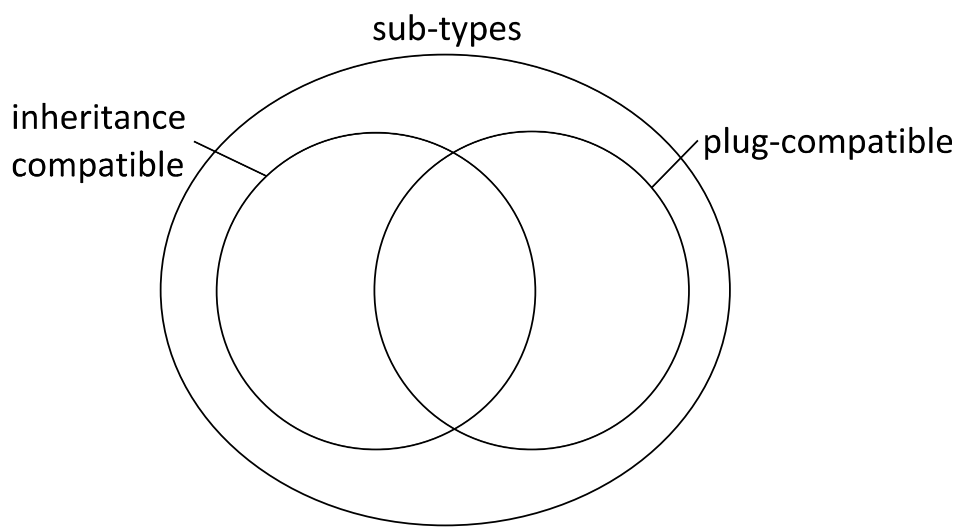
Type System

Institute of Robotics and Mechatronics

- A proper class parameterization requires that the new type **A** is compatible to the original type **B**.

- Obviously **A** must be a subtype of **B**.

- An even more strict requirement is that it needs to be plug-compatible since it is not possible to introduce new connections into a parameter-ized model.



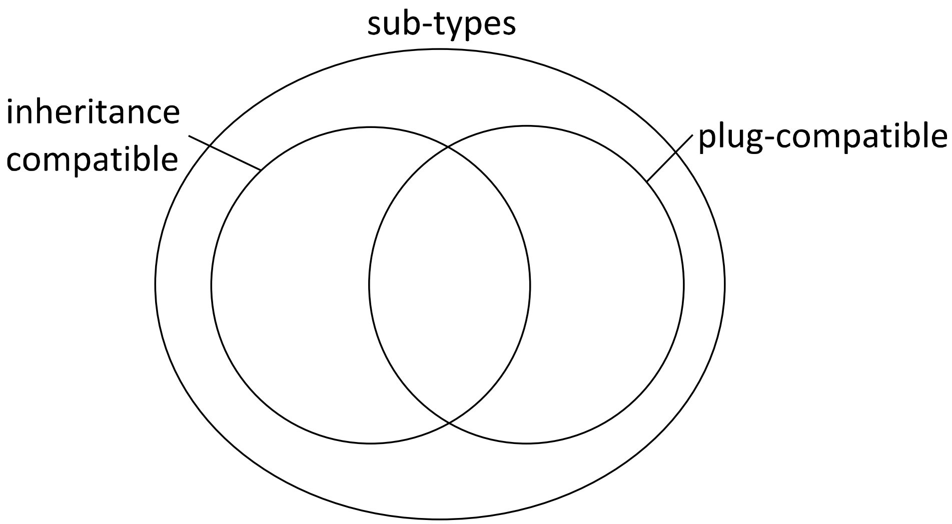
Type System

Institute of Robotics and Mechatronics

- 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.

- 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.



Current Deficiencies

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

- Non-uniform parameterization
- Inappropriate sub-elements
- Prohibitive class generation
- Unwanted parameterization
- Unnecessary restrictions
- Overelaborated syntax

Design Decisions

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

- Separate class parameterization and class generation
- Give classes first class status on the parameter level
- Enable non-prohibitive class generation
- Unify and simplify the language

Class Parameterization

New class parameterization

Example 3 for C_{lass}P_{arameterization}

Institute of Robotics and Mechatronics

```
model TemperatureSensor
  replaceable package Medium =
    Interfaces.PartialMedium;

  Interfaces.FluidPort_in port
  (
    redeclare package Medium =
      Medium
  )

  Medium.BaseProperties medium;

  RealOutput T(unit="K");

equation
  …
  port.p = medium.p;
  port.h = medium.h;
  port.Xi = medium.Xi;
  T = medium.T;

end Temperature;
```

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

  Interfaces.FluidPort_in port
  (Medium = Medium)

  Medium.BaseProperties medium;

  RealOutput T(unit="K");

equation
  …
  port.p = medium.p;
  port.h = medium.h;
  port.Xi = medium.Xi;
  T = medium.T;

end Temperature;
```

Example 3 for C$_{\text{lass}}$P$_{\text{arameterization}}$

Institute of Robotics and Mechatronics

- Class parameters are treated as that what they are:

  As parameters!

- In this way it is also possible to propagate the parameter

- Also the package is not a part of the model anymore. It is just a parameter.

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

  Interfaces.FluidPort_in port
  (Medium = Medium)




  Medium.BaseProperties medium;

  RealOutput T(unit="K");

equation
  …
  port.p = medium.p;
  port.h = medium.h;
  port.Xi = medium.Xi;
  T = medium.T;

end Temperature;
```

Example 2 for C$_{lass}$P$_{arameterization}$

Institute of Robotics and Mechatronics

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


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

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


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

Example 2 for C_{lass}P_{arameterization}

Institute of Robotics and Mechatronics

- The same can be done for components.
- Classes can be part of a normal expression (first-class status).
- Curly braces are introduced in order to distinguish a class-expression from a function call.

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

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

Example 1 for C_{lass}P_{arameterization}

Institute of Robotics and Mechatronics

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

equation
   connect(wheel1.frame_a, frame_a);
   connect(wheel2.frame_a, frame_a);
   connect(wheel3.frame_a, frame_a);
   …

end MultiLevelWheel;
```

```
model MultiLevelWheel
//enumeration
   parameter TModLevels level
   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;
```

Example 1 for C_{lass}P_{arameterization}

Institute of Robotics and Mechatronics

- Class-Expressions are very powerful.
- Now we can transform an enumeration parameter into a class parameter by means of a simple array.

```
model MultiLevelWheel
public
  //enumeration
   parameter TModLevels level
   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;
```

Class Parameterization

Institute of Robotics and Mechatronics

New class generation

Example 1 for C_{lass}G_{eneration}

Institute of Robotics and Mechatronics

```
package PartialPureSubstance
 replaceable model BaseProperties
   …

end PartialPureSubstance;

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;
  …
end SingleGasNasa;
```

```
package PartialPureSubstance
  model BaseProperties
   …

end PartialPureSubstance;

package SingleGasNasa
  extends PartialPureSubstance(…)

  redefined model 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;
  …
end SingleGasNasa;
```

Example 1 for C_{lass}G_{eneration}

Institute of Robotics and Mechatronics

- Since any model can be redefined, it is no longer necessary to mark a model definition as replaceable

- The newly redefined model, must no be inheritance-compatible to the former one.

```
package PartialPureSubstance
  model BaseProperties
   …

end PartialPureSubstance;

package SingleGasNasa
  extends PartialPureSubstance(…)

  redefined model 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;
  …
end SingleGasNasa;
```

Example 2 for ClassGeneration

Institute of Robotics and Mechatronics

```
package Mech3D

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

  model FixedTranslation
    replaceable Frame_a frame_a;
    replaceable Frame_b frame_b;
    …
  end FixedTranslation;

end Mech3D;
```

```
package Mech3D

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

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

end Mech3D;
```

Example 2 for C$_{lass}$G$_{eneration}$

Institute of Robotics and Mechatronics

- The unwanted parameterization can now be removed from the FixedTranslation model

- The keyword replaceable has become redundant.

```
package Mech3D


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


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


end Mech3D;
```

Example 2 for ClassGeneration


```
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;


model FixedTranslation
  extends Mech3D. FixedTranslation(
    redeclare IFrame_a frame_a,
    redeclare 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);
  …
end FixedTranslation;
```

```
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;


model FixedTranslation
  extends Mech3D. FixedTranslation;
  redeclare IFrame_a frame_a,
  redeclare 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);
  …
end FixedTranslation;
```

Example 2 for C_{lass}G_{eneration}

- The redeclaration can be performed on any element.
- But it is removed from the modifier and is now part of the new model.
- In this way, the existing models are protected from being corrupted.

```
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;


model FixedTranslation
  extends Mech3D. FixedTranslation;
  redeclare IFrame_a frame_a,
  redeclare 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);
  …
end FixedTranslation;
```

Conclusions

Lessons learned for Modelica 4

- Class parameterization and class generation should be treated separately.
- They have a very different underlying motivation.
- Class parameterization is requested in foresight and performed by an average user.
- Class generation is applied in hindsight and performed by a potential expert.

Lessons learned for Modelica 4

- Unlike in programming languages, it is difficult in Modelica to impose a strict hierarchy on the type system. The terms sub- and super-type are only of limited value.

- Unlike in programming languages, it is hardly sufficient in Modelica to focus on the assignment operation.

- The criteria for compatibility vary from application to application
  - It is a difference if ones compares components (fully parameterized) or classes (not parameterized).
  - It is a difference if one replaces a component/class by a given parameter or by a redeclaration in a new model.
  - There are more potential differences…

Lessons learned for Modelica 4

- It is meaningful, to have a common syntax for all parameters. Let that be parameters for values, for components, or classes.

- This simplifies the language while making it more powerful.

- It might, however, be meaningful to separate structural parameters from normal parameters. Nevertheless, they should still have the same syntax.

Lessons learned for Modelica 4

- Classes and Components shall represent simple expressions.
- The power of a well-designed language results from the possible combinations of its basic components.
- To this end, each component must form a meaningful entity by itself. Hence, the language must be designed from bottom-up.
- This is not so easy, because practitioners evaluate the language according to relevant examples. And this enforces a top-down view.

The End