# Modelica Change Proposal MCP-0021
# Component Iterators
# Status: In Development
# 2015-12-08, version v2, #1848

## Summary

It is proposed to generalize iterator expressions so that a class name can be used as loop type. The loop iterates then over all instances of this class available in the simulation model. Applications are (a) to compute total properties (e.g. total center of mass), and (b) automatically check requirements on all instances of a particular class.

## Revisions

| *Date* | *Short description of revision* |
|---|---|
| | |
| v2 Dec. 8, 2015 | Hilding Elmqvist, Hans Olsson, Martin Otter:<br>For-loop only over instances on the same or a lower level.<br>Added changes to the Modelica specification. |
| v1 Dec. 2, 2015 | Hilding Elmqvist, Hans Olsson, Martin Otter: Initial draft.<br>The proposal also includes a slight changed syntax (“in class” instead of “in”) based on a discussion between Gerd Kurzbach and Hans Olsson. |

## Contributor License Agreement

Not yet applicable (since no CLA is available).

## Table of Contents

# 1. Rational

This proposal uses text fragments from (*Elmqvist et al., 2015*) without further explicit citing.

In section 10.4.1 of the Modelica Specification 3.3 array constructors with iterators are defined. For example,

```
Real v[:] = {i*i for i in 1:10};
```

generates a vector v with 10 elements and every element is the square of its index. Section 11.2.2.2 "*Types* as Iteration Ranges" states "The iteration range can be specified as `Boolean` or as an enumeration *type*". It is proposed to generalize this scheme, so that a *class name* can be used as iterator expression and in every iteration the loop-variable is *one instance of this class*. The loop iterates over *all instances* of this class available in the simulation model. Example:

```
Real u[:] = {c.v for c in class Class};
```

To signal clearly that the iteration is over all instances of a class, the "in class" keyword is used.

The original version of this proposal used the syntax `{c.v for c in Class}`. Gerd Kurzbach pointed out that it should be clearer defined that this is not a "standard" iterator by adding an additional keyword.

This construct can be used for example in the following way:

```
record Observation "Data that shall be observed"
  constant String name;
  parameter Real m;
  parameter Real v2[3];
  Real           r2;
end Observation;

model ObservationModel "Model that produces the data"
  parameter Real p=2;
  parameter Real v[3] = {-1,2.5,6};
  Real r;
  Real w[3];
  Boolean b;
  Integer i;
  …
end ObservationModel;

model Submodel
  ObservationModel c1(p=3, v={1,-4,8});
  ObservationModel c2;
end Submodel;

model MyModel
  Submodel s1;
  Submodel s2;
  Integer i2[:] = {c.i+3 for c in class ObservationModel};
  Observation obs[:] = {Observation(m=c.p, v2=c.v, r2=c.r, name=c.getInstanceName())
                        for c in class ObservationModel};
   Integer i3[:] = {c.i for c in class ClassNotUsedInMyModel};
end MyModel;
```

In every iteration of the for loop, the iterator variable c adopts the name of an instance of class ObservationModel present in MyModel (the model and its submodels are inspected where the iterator expression is present). The built-in operator c.getInstanceName() is expanded as the instance name of c. If no instance of a class is present in a model, such as for ClassNotUsedInMyModel, then an array with zero dimensions is generated[1]. It is an error, if the class name in the iterator is not known.

[1] In order that it is possible to write generic code without knowing which classes are present in the simulation model, no error must be generated when a class is not present that is used as iterator. If this information is needed, it can be easily determined by inquirying the array size (which is zero, if the class is not used in the model).

Therefore, the above model is equivalent to the following expanded form (showing that the extension can be formally defined by a rewriting rule):

```
model MyModelExpanded
  Submodel s1;
  Submodel s2;

  Integer i2[:] = {s1.c1.i+3, s1.c2.i+3,  s2.c1.i+3, s2.c2.i+3};
  Observation obs[:] = {Observation(m=s1.c1.p,  v2=s1.c1.v,  r2=s1.c1.r,  name="MyModelExpanded.s1.c1"),
                        Observation(m=s1.c2.p,  v2=s1.c2.v,  r2=s1.c2.r,  name="MyModelExpanded.s1.c2"),
                        Observation(m=s2.c1.p,  v2=s2.c1.v,  r2=s2.c1.r,  name="MyModelExpanded.s2.c1"),
                        Observation(m=s2.c2.p,  v2=s2.c2.v,  r2=s2.c2.r,  name="MyModelExpanded.s2.c2")};
  Integer i3[0] ;
end ModelExpanded;
```

It is proposed to have the following restrictions of this new concept of component iterators:

- As class in the iterator only the specialized classes are possible that allow to construct *component* instances: model, block, connector, record, operator record, type (but not package, function, operator function).
- The iteration is only performed over instances of the given class, **not** over instances of subclasses or classes that inherit from the class.
- Component iterators can only be used in the specialized classes model and block.
- An instance iterator inside an "outer component" is ignored.
- [Implementation: If "{… for c in class XYZ}" is found during flattening, the evaluation of the for-expression is delayed, until all elements not containing "in class" are flattened. It is an error, if one of the delayed objects contains "XYZ"]
- Conditionally disabled objects are ignored.
- Examples:
  ```
  MyRecord rec[5]   // every element is an instance in the loop iteration
  MyRecord rec1(a=4);
  MyRecord rec2(a=6);
  MyRecord rec2 = if expr then rec1 else rec2;     //
  ```

Discussions:

- Henrik: This proposal should be finalized when the flattening process is better defined (in MCP 0019) and then the flattening process should be referenced.
- Peter: This is a reflective language construct. In Java there is a powerful reflective API. The question is whether this proposal could be treated as a special case of a more general reflective API.
- Hans: From user point of view the above proposal is fairly easy to use. With a general reflective API it is most likely much more complicated for the user.

This new language feature can be used, for example, in the following application areas:

- Compute total properties of a model (e.g. compute total center of mass from all parts of a satellite).
- Check that the requirements on a class are fulfilled by all instances of this class present in the model at hand.

Such types of examples are discussed in the paper (*Elmqvist et al., 2015*). These examples are also available in the attached test library "TestModels/MetaProperties.mo":

- MetaProperties.Section_2_1_ComponentIterators.Model
- MetaProperties.Section_3_1_TotalMass.TotalMassOfRobot
- MetaProperties.Section_3_1_TotalMass.TotalMassOfRobotWithLog
- MetaProperties.Section_3_2_TotalCenterOfMass.TotalCenterOfMassOfRobot
- MetaProperties.Section_4_3_ClassBinding.CheckPumpsOfBatchPlantWithForLoop
- MetaProperties.Section_4_3_ClassBinding.CheckPumpsOfBatchPlantWithForLoop2
- MetaProperties.Section_4_3_ClassBinding.CheckPumpsOfBatchPlantWithForLoop3
- MetaProperties.Section_4_4_ClassBindingWithTwoClasses.CheckCoolingSystem
- MetaProperties.Section_4_4_ClassBindingWithTwoClassesAndID.CheckCoolingSystemWithID

The example “MetaProperties.Section_3_1_TotalMass.TotalMassOfRobot” is sketched below to compute the total mass of a mechanical system.

When using the Modelica.Mechanics.MultiBody library, there are only two model classes where the mass of a body are defined:

- MultiBody.Parts.Body
- MultiBody.Parts.PointMass

All other specialized parts, like Parts.BodyShape, use an instance of Parts.Body and need therefore not to be handled specially. Model TotalMass computes the total mass of all bodies in a system.

```
model TotalMass "Compute total mass of a system"
  import Modelica.Mechanics.MultiBody.Parts;
  import SI = Modelica.SIunits;
  final parameter SI.Mass m_total = sum({b.m for b in class Parts.Body})  +
                                    sum({b.m for b in class Parts.PointMass})

  // or alternative syntax without class
  final parameter SI.Mass m_total = sum({b.m for b in Parts.Body})  +
                                    sum({b.m for b in Parts.PointMass})
end TotalMass;
```

The assumption made here is that models Parts.Body and Parts.PointMass have a parameter with name m. The sum of the m elements of all instances of Parts.Body and Parts.PointMass is assigned to parameter m_total. This model can be, for example, used to compute the total mass of the r3 robot from the Modelica Standard Library:

```
model TotalMassOfRobot "Compute total mass of r3 robot"
  extends TotalMass;
  extends Modelica.Mechanics.MultiBody.Examples.Systems.RobotR3.fullRobot;
end TotalMassOfRobot;
```

In principal, the total mass can also be computed with Modelica 3.2 language elements via inner/outer: However, this requires to prepare the model beforehand for this action, by adding an outer-declaration in Parts.Body and Parts.PointMass that reports the mass to an inner declaration. With the solution above, the model classes from which information shall be deduced are not modified.

Another example is to check the requirements of all pumps in a circuit (MetaProperties.Section_4_3_ClassBinding.CheckPumpsOfBatchPlantWithForLoop3):

```
record PumpObservation "Observation signals needed for one pump"
  constant String name  "Name of pump";
  Boolean inOperation  "= true, if in operation";
  Boolean cavitate     "= true, if pump cavitates";
end PumpObservation;

PumpRequirements req(obs={PumpObservation(name=  c.getInstanceName(),
                                          inOperation=  c.N_in > 0.1,
                                          cavitate=  c.port_a.p <= 1e4 or c.port_b.p <= 1e4)
                          for c in class Modelica.Fluid.Machines.PrescribedPump})
```

Here, iterator component “c” represents all instances of class Modelica.Fluid.Machines.PrescribedPump in this model. Via the PumpObservation record the needed variables are extracted and passed to the requirements model. Therefore, it is checked whether all instances of PrescribedPump fulfills the defined requirements. Again, this operation is carried out without modifying the underlying PrescribedPump model.

**Open issues**

In order to build modular systems it is necessary to restrict the iterator to only part of the overall model.

Therefore the iteration is here restricted to instances in the model and submodels. This implies that TotalMass must be a base-class (a similar style exists in some other object-oriented languages). Another alternative would be to first go up one “level” – allowing TotalMass to be a normal component, which may seem as better structuring. However, it would also allow multiple TotalMass-components at the same level which is not the intention.

# 2. Proposed Changes in Specification

The precise text of the proposed changes with respect to Modelica Specification 3.3 are in the accompanying document MCP-0021_ComponentIterators_SpecChanges.pdf.

# 3. Backwards Compatibility

This proposal is backwards compatible.

# 4. Tool Implementation

## 4.1 Experience with Prototype

There is a prototype in **Dymola**. The prototype does not implement exactly the above proposal but deviate in the following aspects (which is uncritical; the prototype will be adapted once agreement is reached):

- The actually used syntax is {c.v for c in Class} and not {c.v for c in class Class} as proposed above.
- The for-loop iterates over all intances in the complete simulation model, and not only over the instances on the same or a lower level.

There are the following issues:

- This proposal complicates flattening of a model because all the component iterations have to be performed before flattening of the model parts can be done in which the component iterator is present.

## 4.2 Required Patents

According to our knowledge, no patents are needed to implement this proposal.

# 5. References

Elmqvist H., Olsson H., Otter M. (2015): **Constructs for Meta Properties Modeling in Modelica**. 11th International Modelica Conference, Versailles, Sept. 21-23. http://www.ep.liu.se/ecp/118/026/ecp15118245.pdf