Constructs for Meta Properties Modeling in Modelica

Hilding Elmqvist[1], Hans Olsson[1], Martin Otter[2]

[1]Dassault Systemes, Sweden, {Hilding.Elmqvist, Hans.Olsson}@3ds.com
[2]Institute of System Dynamics and Control, DLR, Germany, Martin.Otter@dlr.de

Abstract

This article proposes two new language constructs for meta-properties modeling in Modelica: (1) Accessing all instances of a given class and (2) extracting in a convenient way the desired information from such instances by allowing to pass type compatible model instances as arguments to functions. In several applications the usefulness of the proposed features are shown. In particular global properties of a model can be computed, such as total power, total mass, total center of mass, or kinetic and potential energy of a multi-body system. An important application is to bind behavioral models and requirement models in a convenient way, for example checking requirements for all instances of a class in a behavioral model, *without* changing the behavioral model.

Keywords: Array comprehension, array constructors, component iterators, binding, instance binding, class binding, total mass, total center of mass, total power.

1 Introduction

This article proposes two new Modelica language constructs to (a) access all instances of a given class and (b) to extract in a convenient way the desired information from such instances. The primary goal for these developments have been the enhancement of requirements modeling in Modelica, as proposed for example by (*Jardin et al., 2011; Bouskela et al., 2015*) and using it concretely in combination with the Modelica_Requirements library (*Otter et. al., 2015*). The difficulty here is to extract observations from a behavioral model (a) in a convenient way, (b) *without changing* the behavioral model, and (c) binding these observations to requirement models to assess the behavioral model.

Due to their generality, these new language constructs allow also other applications which cannot be expressed in a practical way with current Modelica. Most important, global properties, such as total center of mass of a mechanical system, or total power or energy of a system, can be calculated.

The language elements proposed in section 2 are supported in a Dymola prototype (*Dassault Systèmes, 2015*) and all the examples in this paper have been tested with it.

2 Proposals for new Language Elements

2.1 Component iterators

In section 10.4.1 of the Modelica Specification 3.3 (*Modelica Association, 2014*), 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, for example:

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

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

```
record Observation
  constant String name;
  parameter Real m;
  parameter Real v2[3];
  Real r2;
end Observation;

model Class
  parameter Real p=2;
  parameter Real v[3] = {-1,2.5,6};
  Real r;
  Real w[3];
  Boolean b;
  Integer i;
  …
end Class;

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

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

In every iteration of the for loops, the iterator variable c adopts the name of an instance of class Class present in Model (the complete model is inspected, independently 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 ClassNotPresent, then an array with zero dimensions is generated.[1]

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

```
model ModelExpanded
  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="ModelExpanded.s1.c1"),
           Observation(m=s1.c2.p, v2=s1.c2.v, r2=s1.c2.r,
                       name="ModelExpanded.s1.c2"),
           Observation(m=s2.c1.p, v2=s2.c1.v, r2=s2.c1.r,
                       name="ModelExpanded.s2.c1"),
           Observation(m=s2.c2.p, v2=s2.c2.v, r2=s2.c2.r,
                       name="ModelExpanded.s2.c2")};
  Integer i3[0] ;
end ModelExpanded;
```

It is also proposed to extend the array constructor with guards to be able to restrict the set of instances using a built-in operator instanceIn(..):

```
model ModelWithGuard
  Submodel s1;
  Submodel s2;
  Integer i2[:] = {c.i+3 for c in Class
                         if c.instanceIn(s1)};
end ModelWithGuard;
```

Here c takes the values “s1.c1” and “s1.c2”.

Naturally, there are restrictions of this new concept of component iterators, in particular:

- As class in the iterator only the specialized classes are possible that allow to construct *component* instances: model, block, connector, record, operator record (but not package, function, operator function).
- Component iterators can only be used in the specialized classes model and block.
- Component instances in functions are ignored (not returned) by component iterators.

[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.

2.2 Model instances as arguments to functions

It is proposed to generalize the calling mechanism of Modelica functions so that model, block, connector, record and operator record instances can be passed as arguments to functions, provided the instance is a subtype of the corresponding record function argument. Example:

```
model Submodel
  Real r1;
  Real r2;
  Integer i2
  Pin p1, p2;
protected
  Integer i1;
  …
end Submodel;

record Record
  Real r1;
  Integer i2;
end Record;

function get
  input Record rec;
  output Real result;
algorithm
  result :=rec.r1 + rec.i2;
end get;

model Model
  Submodel s1;
  Real r=get(s1);
end Model;
```

Note that input argument rec of function get expects an instance of record Record when calling the function. However, an instance of model Submodel is passed when calling this function. The semantics is that the function extracts the values of all elements of s1 that are also present in record rec. The function call in the example is therefore equivalent to the Modelica 3.3 function call:

```
model ModelExpanded
  Submodel s1;
  Real r = get(Record(r1=s1.r1, i2=s1.i2));
end ModelExpanded;
```

Again, this language extension can be formally specified as rewriting rule. Since the rewriting is done locally, it seems like a minor convenience improvement. As the requirements binding applications in section 4 will show, this is not the case: The essential advantage is to define the elements that are extracted from a model only *once* (in the above example in the definition of function get) and the user of the function does not need to know which elements are extracted. If this function is used for many models, manually applying the rewriting would be no longer practical and would be error prone.

To summarize, the proposed language element is a short hand notation that is especially very convenient, if the record input argument to a function has many elements and the function is called many times for many model instances.

3 Application: Total Properties

In this section several applications are sketched how the language constructs from section 2 can be used in applications where total properties of a system model shall be computed.

DOI 10.3384/ecp15118245

3.1 Total mass

In a 3-dimensional mechanical system it is sometimes required to compute the total mass of a system. For example, to determine the complete mass of a vehicle, satellite, or robot from a behavioral model and compare it with the measured weight of the built system and/or with a CAD model. This allows to detect modeling errors, but it might also be needed to check a requirement (for example, the total aircraft weight must be at most xx kg).

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 system"
  import Modelica.Mechanics.MultiBody.Parts;
  import SI = Modelica.SIunits;
  final parameter SI.Mass m_total =
                        sum({b.m for b in Parts.Body})
end TotalMass;
```

The assumption made here is that model Parts.Body has a parameter with name m. The sum of the m elements of all instances of Parts.Body 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 (see also Figure 1):

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

Figure 1. Animation of robot r3.

Simulating and inspecting the result file gives
m_total = 134.3 kg

If the result is not as expected, it might be difficult to figure out the error in a larger system. It is then helpful to print out all the found masses, as performed in the next model:

```
model TotalMassWithLog Total mass  with log"
  import Modelica.Utilities.Streams.print;
  import Modelica.Mechanics.MultiBody.Parts;
  import SI = Modelica.SIunits;

  final parameter SI.Mass m_total = sum(mObs[:].m);
protected
  record MassObservation
    String  name "Name of body";
    SI.Mass m "Mass of body";
  end MassObservation;
  parameter MassObservation mObs[:] =
  {MassObservation(name=b.getInstanceName(),
                   m=b.m)
                  for b in Parts.Body};
equation
  when initial() then
     // print body names (mObs[:].name) and values
  end when;
end TotalMassWithLog;
```

Since the name of the body and its mass shall be extracted, a local record MassObservation is introduced and filled with the array comprehension language element. The built-in operator getInstanceName(), returns the names of the found body instances. When using the model for the r3 robot, the following message is printed during initialization:

```
... Body masses:
mechanics.b0.body: 1 kg
mechanics.b1.body: 1 kg
mechanics.b2.body: 56.5 kg
mechanics.b3.body: 26.4 kg
mechanics.b4.body: 28.7 kg
mechanics.b5.body: 5.2 kg
mechanics.b6.body: 0.5 kg
mechanics.load.body: 15 kg
Total mass: 134.3 kg
```

3.2 Position vector to total center of mass

In space applications there is sometimes the need to determine the position of the total center of mass of a satellite, rocket, or space station. One reason is, for example, that a path planning software computed the desired trajectory (of the total center of mass) and the detailed mechanical model of the system shall start at a point on this trajectory. Another reason is when a robot is mounted on a free flying satellite system (as for example planned for repair operations). Then, movements of the robot do not change the position of the total center of mass, and a control system for grasping has to take this effect into account (and needs to know the total center of mass). With the definitions of Figure 2:

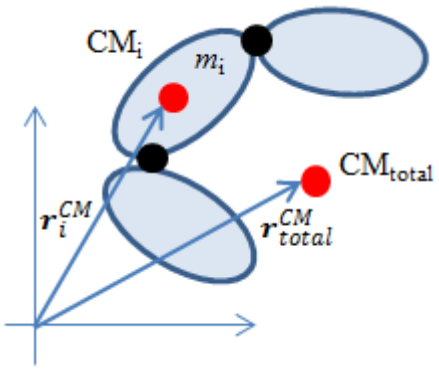

Figure 2. Computation of the total center of mass.

the well-known equation to compute the position of the total center of mass is (under the assumption that all absolute position vectors are resolved in the inertial frame):

$$\boldsymbol{r}_{total}^{CM} = \frac{\sum m_i \cdot \boldsymbol{r}_i^{CM}}{\sum m_i} \qquad (1)$$

Model `TotalCenterOfMass` computes the absolute position of the total center of mass of all bodies in a system (rCM_total):

```
model TotalCenterOfMass
  import Modelica.Mechanics.MultiBody.Frames;
  import Modelica.Mechanics.MultiBody.Parts;
  import SI = Modelica.SIunits;

  SI.Mass m_total = sum(obs[:].m) "Total mass";
  SI.Position rCM_total[3] =
                {sum(m_rCM[:,j])/m_total for j in 1:3}
                              "Total center of Mass";
protected
  record FrameObservation
    SI.Position r_0[3];
    Frames.Orientation R "Orientation matrix";
  end FrameObservation;

  record BodyObservation
    SI.Mass m "Mass of body";
    SI.Position r_CM[3] "Vector frame_a to CM";
    FrameObservation frame_a;
  end BodyObservation;

  function getObservations
    input BodyObservation obs;
    output BodyObservation result=obs;
  algorithm  annotation(Inline=true);
  end getObservations;

  BodyObservation obs[:] =
          {getObservations(b) for b in Parts.Body};
  Real m_rCM[size(obs,1),3](unit="kg.m");
equation
  for i in 1:size(obs,1) loop
    m_rCM[i,:] =obs[i].m*(obs[i].frame_a.r_0 +
       Frames.resolve1(obs[i].frame_a.R, obs[i].r_CM));
  end for;
end TotalCenterOfMass;
```

The computation is performed in the following way:

1. The variables that shall be extracted from every body are defined in the protected section. These are the mass m, the local position vector r_CM from frame_a to the center of mass of the body, the absolute position vector frame_a.r_0 from the inertial frame to frame_a and the orientation matrix frame_a.R transforming the inertial frame into frame_a. All these variables are defined in a record that has the same structure and uses the same names as used in model Parts.Body.
2. The central declaration of obs extracts the desired information from all instances of model Parts.Body:
   obs[:] = {getObservations(b) for b in Parts.Body};
   There are two possibilities, either a record constructor is used to extract the body variables (as in TotalMassWithLog), or a function is used as above (getObservations(b)) and one Parts.Body instance is passed to the function. With the new semantics of section 2.2, the tool extracts all variables from the instances and copies them into instances of the BodyObservation record.
3. Once the variables from all instances of Parts.Body are extracted, it is rather straightforward to compute the desired position vector to the total center of mass. This requires to transform all body-fixed position vectors obs[i].r_CM into the inertial frame, add the absolute position vectors at the body frames obs[i].frame_a.r_0, and use equation (1).

This model can be, for example, used to compute the position vector to the total center of mass of the r3 robot, see Figure 1:

```
model TotalCenterOfMassOfRobot
  "Compute total center of mass of r3 robot"
  import Modelica.Mechanics.MultiBody.Examples;
  extends TotalCenterOfMass;
  extends Examples.Systems.RobotR3.fullRobot;
end TotalCenterOfMassOfRobot;
```

A simulation produces the result in Figure 3.

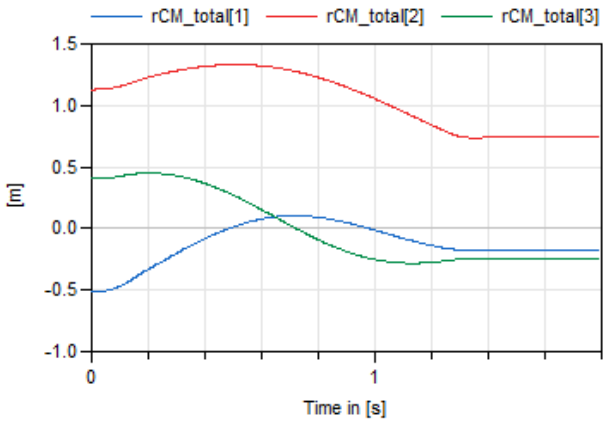

Figure 3. Simulation results to compute the position vector to the total center of mass of the r3 robot.

DOI 10.3384/ecp15118245

4 Application: Requirements Binding

In this section a class of applications is discussed how to bind requirement models in a convenient way to behavioral models using the language constructs from section 2.

4.1 Overview

In (*Jardin et al., 2011*) a concept was developed to model properties and requirements in Modelica. This was significantly enhanced in (*Bouskela et al., 2015*) and a sophisticated Modelica library for this approach was developed in (*Otter et al., 2015*).

In industry, requirements are usually defined in natural language, such as[2]

- *When in operation, pumps shall not cavitate*
  (= the pressure in a pump must be larger than a minimum pressure)
- *In flight, with only one engine running, the air distribution circuit shall provide nominal performance.*
- *After three failures of starting an engine, the APU (Auxiliary Power Unit) must be started.*

The basic idea is to provide a suitable Modelica library to model such requirements in a formal way with Modelica, see (*Otter et al., 2015*) for details. There are the following key requirements from industry (*Bouskela et al., 2015*):

1. The requirement models are developed independently from the behavioral models that shall be checked. The reason is (a) that requirements shall be formally specified before designing the system (and therefore a behavioral model is not yet available), and (b) that requirements are defined from system architects which are not the simulation specialists building up the behavioral models. As a consequence, the variables used in requirement models need not be the same (not even the data type) as the (corresponding) variables in the behavioral model.
2. When associating requirement models to behavioral models (in order to check the behavioral models), it is usually not possible or not allowed to change or adapt the code of the behavioral models.

Based on these restrictions there is a fundamental issue how to extract variables from a behavioral model (these variables are called "observation" variables below) and assign them as inputs to the requirement models. This process is called "Binding" in the sequel. In (*Jardin et al., 2011*) Modelica buses have been used for the "Binding". This violates the restrictions above since the behavioral model must be modified and the variable names in the behavioral and requirement models must be identical. Furthermore, in larger use cases of EDF and Dassault Aviation it turned out that this is not a practical approach because it is also much too inconvenient to use.

There have been also other proposals how to define the "Binding", such as (*Schamai, 2013*). Still, until now, no satisfactory approach is known to be used conveniently in Modelica. In the rest of this section it is shown that the proposed new language elements of section 2 provide a convenient and powerful way to define the "Binding".

4.2 Instance binding

The goal is to check the following requirement for all pumps present in a system:

When in operation, a pump shall not cavitate.

This requirement can be checked with the following model[3]:

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

model PumpRequirements
  import Modelica.Utilities.Streams.print;
  input PumpObservation obs[:];
equation
  for i in 1:size(obs,1) loop
    when obs[i].inOperation and obs[i].cavitate then
      print("... warning: pump " + obs[i].name +
            " is cavitating during operation");
    end when;
  end for;
end PumpRequirements;
```

The requirement definition above is independently of the construction of the pump and how the values of the Boolean variables inOperation and cavitate are determined from the behavioral model. For a concrete pump, here from:

Modelica.Fluid.Machines.PrescribedPump

a function is used to map observation variables of an instance of PrescribedPump to the variables needed by the requirement model:

```
function fromPrescribedPump
  input PrescribedPumpObservation obs;
  input String name;
  input Modelica.SIunits.Pressure p_cavitate=0.99e5;
  output PumpObservation result(
              name        = name,
              inOperation = obs.N_in > 0.1,
              cavitate    = obs.port_a.p <= p_cavitate or
                            obs.port_b.p <= p_cavitate);
```

[2] These and further examples from this section are from (*Bouskela et al., 2015*) or (*Otter et al., 2015*)

[3] In case of violation, only a warning message is printed. In (*Otter et al., 2015*) a more involved handling is performed.

```
  protected
    record PortObservation
      Modelica.SIunits.Pressure p;
    end PortObservation;
    record PrescribedPumpObservation
      Real N_in(unit="1/min");
      PortObservation port_a;
      PortObservation port_b;
    end PrescribedPumpObservation;
  algorithm
    annotation(GenerateEvents=true);
  end fromPrescribedPump;
```

Here PrescribedPumpObservation is a record declared internally in the function that defines which variables shall be extracted from an instance of the PrescribedPump model. In this case, these are N_in, the speed of the pump shaft, as well as port_a.p and port_b.p, the pressures at the pump ports. This record is used as input argument together with the name of the pump. As output argument, an instance of the PumpObservation record is used and via a record constructor the variables from the PrescribedPumpObservation are mapped to the PumpObservation output argument.

When using the record constructor, relations are present, such as obs.N_in > 0.1. With a normal function, this would lead to an error during translation, because (a) relations in functions do not generate events, (b) this function is called in the continuous-time part of Modelica and (c) in Modelica it is not allowed that Boolean variables can change during continuous-time integration. This problem is resolved with the annotation GenerateEvents=true. This is a standard Modelica annotation and defines that relations in this function generate events. The effect is that the Boolean variables can only change at event points.

The PumpRequirements model and the mapping function from a PrescribedPump to this model is now evaluated with example BatchPlant_StandardWater form the Modelica Standard Library. A screen shot of this model is shown in Figure 4. This model has two instances of model PrescribedPump, named P1 and P2 (at the bottom of the diagram). This system is checked with the following model:

```
  model CheckPumpsOfBatchPlant
    import Modelica.Fluid.Examples.AST_BatchPlant;
    extends AST_BatchPlant.BatchPlant_StandardWater;

    PumpRequirements req(obs=
                         {fromPrescribedPump(P1,"P1"),
                          fromPrescribedPump(P2,"P2")});
  end CheckPumpsOfBatchPlant;
```

As can be seen, an instance of the PumpRequirements model is defined. The pump instances P1 and P2 from the BatchPlant_StandardWater model are passed as arguments to function fromPrescribedPump. With the proposed language feature of 2.2, observation variables

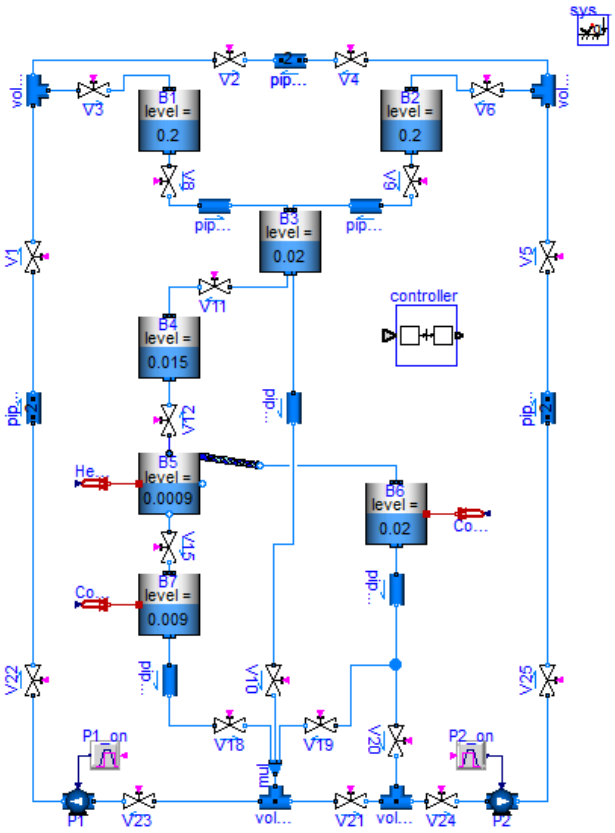

Figure 4. Example model BatchPlant_StandardWater form the Modelica Standard Library.

are extracted from the pumps and are transformed as needed from the requirement models.

Note, as required the behavioral model (= BatchPlant_StandardWater) is not modified and the observation variables used in the behavioral model and the requirement model might be different. Simulation results are shown in Figure 5.

As can be seen, one of the pumps is cavitating once. As a result, the log window contains a warning message:

... warning: pump P1 is cavitating during operation

There is always the need to specify for one or more individual instances specific requirements (for example, "*at least one pump present in room A must always be in operation*"[4]), and then the approach above, also called instance binding, has to be applied.

However, there are also requirements that hold for *many* instances, and the instance binding may then become inconvenient. In the next section this case is handled by "class binding".

[4] In this case a vector of pumps must be passed to the requirement model consisting of the pump instances present in room A.

DOI 10.3384/ecp15118245

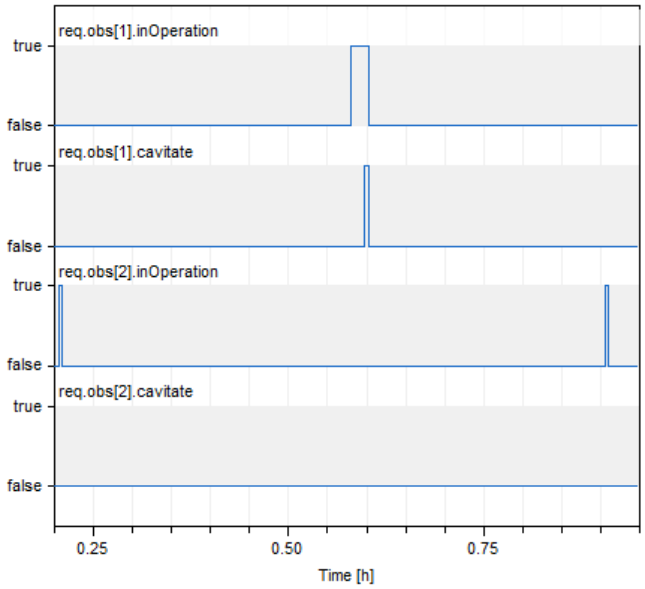

Figure 5. Simulation results for the requirements model of BatchPlant_StandardWater of Figure 4.

4.3 Class binding

In case a requirement holds for all instances of a class, the array of observations need not be defined manually but can be generated with the array comprehension for classes of section 2.1. The previous example can then be defined as:

```
model CheckPumpsOfBatchPlantWithForLoop
  import Modelica.Fluid.Examples.AST_BatchPlant;
  extends AST_BatchPlant.BatchPlant_StandardWater;

  PumpRequirements req(obs=
    {fromPrescribedPump(p, p.getInstanceName())
      for p in Modelica.Fluid.Machines.PrescribedPump});
end CheckPumpsOfBatchPlantWithForLoop;
```

Note, that this model generates requirement checks for **any number of pumps** in the circuit. With the planned guard on for-loops, it would also be possible to limit the for-loop to instances of the desired class in a specific sub-model.

4.4 Advanced class binding

Class-binding becomes more involved if instances for two or more classes have to be treated simultaneously. Here is a sketch of two different approaches based on the scenario defined in (*Bouskela et al, 2015*):

A pump might be built up from several components, for example with a centrifugal pump and with an electric motor that drives the centrifugal pump. However, the requirements from section 4.2, PumpRequirements, are always the same, independently of the underlying technology of the pump.

Assume that a cooling circuit is defined by two subsystems that contain each three pumps built up by centrifugal pump and electric motor components:

```
model Subsystem
  CentrifugalPump P1;
  ElectricMotor   M1;

  CentrifugalPump P2;
  ElectricMotor   M2;

  CentrifugalPump P3;
  ElectricMotor   M3;
  ....
end Subsystem;

model CoolingSystem
  Subsystem subsystem1;
  Subsystem subsystem2;
end CoolingSystem;
```

The goal is to check the pumps. In order to do this one has to collect observation variables, say, from P1 and M1 and pass them to PumpRequirements. This is straightforward for instance binding, but more complicated if code for any number of instances shall be implemented.

The essential difficulty is that information is missing: It is not known from the Subsystem definition whether P1 and M1 or P1 and M2 or P1 and M3 form the pump. It might be possible to deduce this information from the connection of the components but it seems quite complicated to provide language elements to the user such that he/she can implement code to deduce the connection structure. Furthermore, even then there might be not a unique solution because the motor M1 might not be directly connected to P1 (but via another auxiliary component), or two motors might be connected to P1, but only one of them is relevant for the requirement model.

The solution proposed here is to add more information. If it is not allowed or not possible to modify the behavioral model, the only way is to list the instances that belong together. This is performed in the following model:

```
model CheckCoolingSystem
  extends CoolingSystem;
  constant String pumpMotorAssociations[:,3]=
    ["subsystem1", "P1", "M1";
     "subsystem1", "P2", "M2";
     "subsystem1", "P3", "M3";
     "subsystem2", "P1", "M1";
     "subsystem2", "P2", "M2";
     "subsystem2", "P3", "M3"];

  constant Integer motorIndices[:]=
    associateCPumpsAndEMotorsByNames(
      {p.getInstanceName() for p in CentrifugalPump},
      {m.getInstanceName() for m in ElectricMotor},
      pumpMotorAssociations, getInstanceName());

  PumpRequirements req(obs=
      fromCPumpAndEMotor(
          {fromCPump(p) for p in CentrifugalPump},
          {fromEMotor(m) for m in ElectricMotor},
          motorIndices));
end CheckCoolingSystem;
```

Array pumpMotorAssociations has three columns: The first column contains the path name of the subsystem in which the pump is present, such as "subsystem2". The second and third columns contain the names of the centrifugal pump and the electric motor that form the pump, such as "P3", "M3". This array has to be manually constructed for the circuit at hand.

With function associateCPumpsAndEMotorsByNames the association of centrifugal and electric motor instances is determined once during translation of the model and the result is assigned to the constant Integer array motorIndices, such that if centrifugal pump i is associated with electric motor j, then motorIndices[i]=j.

In order to map the observations from the behavioral model to the PumpRequirements model several new mapping functions are needed. For example, function fromCPumpAndEMotor can be implemented as:

```
function fromCPumpAndEMotor
  input PumpObservation_cavitate      pObs[:];
  input PumpObservation_inOperation mObs[:];
  input Integer motorIndices[size(pObs,1)];
  output PumpObservation obs[size(pObs,1)];
algorithm
  for i in 1:size(pObs,1) loop
    obs[i].cavitate     := pObs[i].cavitate;
    obs[i].inOperation :=
                        mObs[motorIndices[i]].inOperation;
  end for;
end fromCPumpAndEMotor;
```

As can be seen, the motorIndices vector is used to extract observation variables from the electric motor observations mObs[motorIndices[i]] that are associated with the corresponding centrifugal pump observations pObs[i].

In case it is possible to modify the behavioral model to be checked (here: CoolingSystem), another approach might be more convenient and less error prone: Every component gets an additional unique Integer identification number, called “id”. A centrifugal pump and an electric motor belong together and form one pump, if both have the same “id”. It is not allowed that any other pump in the circuit has the same “id”. The circuit can then be modelled in the following way:

```
model SubsystemWithID
  CentrifugalPumpWithID P1 (id=1);
  ElectricMotorWithID     M1(id=1);
  CentrifugalPumpWithID P2 (id=2);
  ElectricMotorWithID     M2(id=2);
  CentrifugalPumpWithID P3 (id=3);
  ElectricMotorWithID     M3(id=3);
end SubsystemWithID;

model CoolingSystemWithID
  SubsystemWithID subsystem1;
  SubsystemWithID subsystem2(P1(id=4),M1(id=4),
                             P2(id=5),M2(id=5),
                             P3(id=6),M3(id=6));
end CoolingSystemWithID;
```

The checking of the requirements can be performed as:

```
model CheckCoolingSystemWithID
  extends CoolingSystemWithID;

  constant Integer motorIndices[:]=
      associateCPumpsAndEMotorsByID(
          {p.id for p in CentrifugalPumpWithID},
          {m.id for m in ElectricMotorWithID});

  PumpRequirements req(obs=
      fromCPumpAndEMotor(
          {fromCPump(p) for p in CentrifugalPumpWithID},
          {fromEMotor(m) for m in ElectricMotorWithID},
          motorIndices));
end CheckCoolingSystemWithID;
```

Since the information about the association of centrifugal pump and electric motor is within the behavioral model, the code for the requirement check in CheckCoolingSystemWithID is generic. Function associateCPumpsAndEMotorsByID determines the same index vector motorIndices as before. The implementation of this function is however simpler:

```
function associateCPumpsAndEMotorsByID
  input Integer pumpIds[:];
  input Integer motorIds[:];
  output Integer motorIndices[size(pumpIds,1)];
algorithm
  for i in 1:size(pumpIds,1) loop
    for j in 1:size(motorIds,1) loop
      if motorIds[j] == pumpIds[i] then
        motorIndices[i] :=j; break;
      elseif j == size(motorIds,1) then
        assert(false, "id's are wrong");
      end if;
    end for;
  end for;
end associateCPumpsAndEMotorsByID;
```

In order to provide better diagnostics in case of an error, it is useful to pass the instance names of the centrifugal pumps and of the electric motors also to this function. For simplicity this was not done above. Furthermore, it should also be checked, that the id’s are unique.

5 Summary

This paper proposes two new Modelica language elements to extract information from a model in a convenient way. This opens up new applications of Modelica that could not be practically handled before. The language elements and the sketched applications have been evaluated and tested with a Dymola prototype.

Acknowledgements

This paper is based on research performed within the ITEA2 project MODRIO. Partial financial support of the Swedish VINNOVA and the German BMBF is highly appreciated.

Helpful discussions with Daniel Bouskela, Nguyen Thuy, Audrey Jardin (EDF), Eric Thomas, Maxim Payelleville (Dassault Aviation), Wladimir Schamai (Airbus Defence and Space), Peter Fritzson, Lena Buffoni (PELAB), Alfredo Garro and Andrea Tundis (UNICAL) on the "Requirements Binding" application of section 4 are appreciated.

References

Bouskela D., Thuy N., Jardin A. (2015): **D2.1.1 – Modelica extensions for properties modelling, Part II: Modeling Architecture for the Design Verification against System Requirements**. Internal report, ITEA2 MODRIO project, March 2015.

Dassault Systèmes (2015): **Dymola 2016**. http://www.Dymola.com

Jardin A., Bouskela D., Thuy N., Ruel N., Thomas E., Chastanet L., Schoenig R., Loembé S. (2011): **Modelling of System Properties in a Modelica Framework**. Proceedings 8th Modelica Conference, Dresden, Germany, March 20-22., pp. 579-592. Download: http://www.ep.liu.se/ecp/063/065/ecp11063065.pdf

Modelica Association (2014): **Modelica, A Unified Object-Oriented Language for Systems Modeling. Language Specification, Version 3.3, Revision 1**, June 11, 2014. Download: https://www.modelica.org/documents/ModelicaSpec33Revision1.pdf

Otter M., Thuy N., Bouskela D., Buffoni L., Elmqvist H., Fritzson P., Garro A., Jardin A., Olsson H., Payelleville M., Schamai W., Thomas E., Tundis A. (2015): **Formal Modeling and Automatic Verification of Requirements**. Proceedings 11th Modelica Conference, Versailles, France, Sept. 21-23.

Schamai, W. (2013): **Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool**. Ph.D. Thesis, No. 1547, University of Linköping. Download: http://liu.diva-portal.org/smash/record.jsf?pid=diva2:654890