Modelica Change Proposal MCP-0023
Model to Record
Status: Under Evaluation
2016-09-12 version v2, #1953

Summary

It is proposed to utilize a record class name as cast-operator to extract all models with the same name (and type) from a model and return a record. The main motivation for this feature is to extract in a simple way variable values from model instances and utilize them in formally defined requirements to check properties of these instances in an automatic way by simulation.

Revisions

Date	*Short description of revision*
v2 Sept. 12, 2016	Minor improvements of the text by Martin Otter.
v1 March 7, 2016	Prototype in Dymola by Hans Olsson; this initial version of the document by Martin Otter, based on MCP-0020.
Dec. 14-16	At 88th Modelica Design Meeting, MCP-0020 was analyzed and issues detected. Martin Sjölund proposed instead to define a cast from a model to a record
Dec. 2, 2015	MCP-0020: Hilding Elmqvist, Hans Olsson, Martin Otter: Initial draft to call models as arguments to functions

Contributor License Agreement

Not yet applicable (since no CLA is available).

Table of Contents

1. Rationale

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

It is proposed to use a record class name to cast variables of a model to a record instance. Example:

```
model Submodel              model Model
  Real r1;                    Submodel s1;
  Real r2;                    MyRecord rec = MyRecord(s1);
  Integer i2                end Model;
  Pin p1, p2;
protected
   Integer i1;
  …
end Submodel;

record MyRecord
  Real r1;
  Integer i2;
end MyRecord;
```

Note that the record constructor MyRecord is overloaded. When this operator has only a single input argument and this input argument is an instance of a model, block, connector then the record constructor acts as cast operator that extracts the values of all (potentially nested) elements of s1 that are also present in record MyRecord. The function call in the example is therefore equivalent to the Modelica 3.3 function call:

```
model ModelExpanded
   Submodel s1;
   MyRecord rec = MyRecord(r1=s1.r1, i2=s1.i2);
end ModelExpanded;
```

This language extension can be formally specified as rewriting rule. There is the obvious restriction that the instance passed to the function, must have all elements (with identical types) that are present in record MyRecord.

Since the rewriting is done locally, it seems like a minor convenience improvement. 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 record MyRecord) 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.

Note, the cast operation is uniquely distinguished from a record constructor call, because an argument of the record constructor cannot be a model, block or connector instance.

The major application area for this extension is to extract information from model instances and use them to check requirements by simulation. Another application area is to extract information from models to compute total properties (like total center of mass). Several examples of this type are provided in the paper (*Elmqvist et al., 2015*). These examples could be slightly reformulated with the proposed cast operator. This is shown with test model

MetaProperties2.CastModelToRecord.CheckPumpsOfBatchPlant2

This model is sketched below:

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[1]:

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

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

```
model PumpRequirements "Requirements on a set of pumps"
  input PumpObservation obs[:] "Generic observation signals for a set of pumps";
equation
  for i in 1:size(obs,1) loop
    when obs[i].inOperation and obs[i].cavitate then
        … // Requirement violated, print warning message
    end when;
  end for;
end PumpRequirements;
```

Let's assume that for the architecture under consideration, a Modelica behavioral model is implemented using Modelica.Fluid.Machines.PrescribedPump instances. One possibility is specializing the generic pump requirement to a requirement on the PrescribedPump type by mapping variables from the PrescribedPump to the generic description:

```
record PrescribedPumpObservation
  Real N_in(unit="1/min") "Speed of pump";
  PortObservation port_a "Variables from port_a";
  PortObservation port_b "Variables from port_b";
end PrescribedPumpObservation;

record PortObservation
  Modelica.SIunits.Pressure p "Pressure in the port";
end PortObservation;

model PrescribedPumpRequirements
  constant String modelNames[:];
  PrescribedPumpObservation modelObs[size(modelNames,1)];
  parameter Modelica.SIunits.Pressure p_cavitate=0.99e5;
  extends PumpRequirements(
    final obs = {PumpObservation(name=modelNames[i],
                    inOperation = modelObs[i].N_in > 0.1,
                    cavitate = modelObs[i].port_a.p <= p_cavitate or
                               modelObs[i].port_b.p <= p_cavitate) for i in 1:size(modelNames,1)});
end PrescribedPumpRequirements;
```

Note, it is assumed that the PrescribedPump is in operation, when the pump speed exceeds a minimum value. Checking the generic requirements for a set of PrescribedPump instances can now be simply defined with:

```
model CheckPumpsOfBatchPlant
  extends Modelica.Fluid.Examples.AST_BatchPlant.BatchPlant_StandardWater
  PrescribedPumpRequirements req(modelNames={"P1", "P2"},
                                 modelObs={PrescribedPumpObservation(P1),
                                           PrescribedPumpObservation(P2)});
end CheckPumpsOfBatchPlant;
```

Instance req needs a vector of PrescribedPumpObservation records that pass the actual variable values of the pumps. This is conveniently performed by the newly proposed feature that extracts all variables from a model instance, such as "P1" (a pump instance defined in BatchPlant_StandardWater), passes them to the record constructor PrescribedPumpObservation(P1) and returns an appropriate instance of this record type. This automatic variable casting is allowed, as long as all variable names present in the record are also variable names present in the model instance passed as argument. This approach could be further simplified if the record constructor is also supported on a vector of models and is then applied on each element individually (this feature is not yet supported in the Dymola prototype):

```
model CheckPumpsOfBatchPlant2
  extends Modelica.Fluid.Examples.AST_BatchPlant.BatchPlant_StandardWater;
  PrescribedPumpRequirements req(modelNames={"P1", "P2"},
                                 modelObs = PrescribedPumpObservation({P1, P2}));
end CheckPumpsOfBatchPlant2;
```

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-0023_ModelToRecord_SpecChanges.pdf.

3. Backwards Compatibility

This proposal is backwards compatible.

4. Tool Implementation

4.1 Experience with Prototype

There is a prototype in **Dymola**. No particular difficulties had been detected with the prototype.

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