Custom Annotations:
Handling Meta-Information in Modelica

Dirk Zimmer[1], Martin Otter[1], Hilding Elmqvist[2], Gerd Kurzbach[3]
[1]German Aerospace Center (DLR), Institute of System Dynamics and Control,
D-82234 Wessling, Germany,
[2]Dassault Systèmes AB, Ideon Science Park, SE-223 70 Lund, Sweden
[3]ITI GmbH, 01067 Dresden, Germany
dirk.zimmer@dlr.de, martin.otter@dlr.de, hilding.elmqvist@3ds.com, kurzbach@itisim.com

Abstract

Annotations and attributes form an important part of the Modelica language. They are used to include various meta-information such as documentation, external C-code, compilation hints, etc. Given the increasingly wide field of potential applications the set of useful annotations becomes too large to be included in the language specification. Hence we present a proposal how a Modelica modeler may define his own annotations and how such custom annotations can be organized within Modelica libraries. In the long term, the goal is to move the definition of standardized annotation, as well as of attributes, from the Modelica specification to a standard library.

Keywords: meta-information; custom annotations; optimization setup; Monte Carlo simulation setup; Kalman filter setup; uncertainty setup.

1 Introduction

The main purpose of Modelica is to enable the equation-based modeling of physical systems. In addition to this primary objective, the modeler has to care about the usability of his/her components. This includes a variety of tasks: documentation needs to be written, icons need to be drawn, a 3D visualization has to be provided, and compilers might need hints for generating more efficient code.

All this is meta-information to the actual physical model but as Figure 1 shows, it can account for a major share of the code: For the FixedTranslation component (a rigid rod in 3D Mechanics), the physical modelling contributes only to 14% of all the code. Of course such a comparison is skewed since it is doubtful to compare manually typed equations with auto-generated code for graphical objects but nevertheless the handling of meta-information deserves to be a major concern for the future design of the Modelica language.

An improved solution for meta-information in Modelica becomes necessary since there is a desire to include more and more information into the models. Especially, a model might be used not only in simulation, but in other analysis and synthesis methods, and then additional model-specific data is needed. For example, sensitivity analysis needs uncertainty data for model variables, Monte Carlo simulation needs stochastic distribution data on states and/or parameters, an optimization setup needs the information which parameters and/or input signals shall be optimized, and in which range the optimization shall take place.

To meet these demands, we propose an enhancement to Modelica: custom annotations. But before we address the new proposal, let us look at the current handling of meta-information in Modelica and its weak spots and then formulate the requirements for a new design.

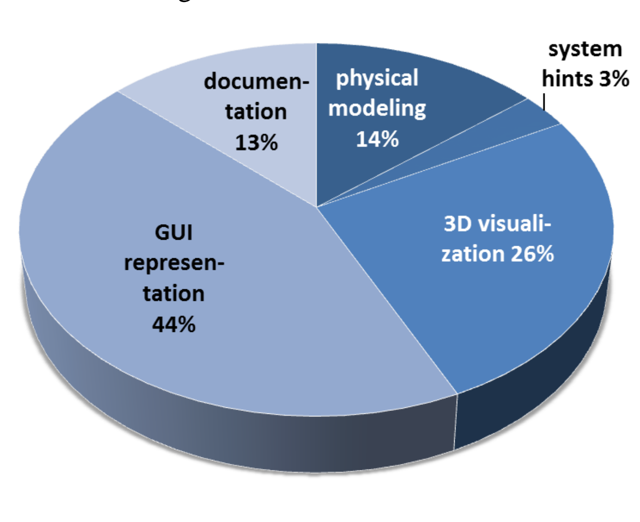


Figure 1: Percentage of characters devoted to certain tasks in Modelica.Mechanics.MultiBody.Parts. FixedTranslation. Source: (*Zimmer 2008*)

2 Handling of Meta-Information

2.1 Meta-Information in Current Modelica

The Modelica language (*Modelica Association 2012*) offers currently two devices that are used for meta-information: annotations and attributes.

Standard annotations are defined for a multitude of issues: graphical information for icons and diagrams, GUI-design, documentation, version handling, etc. Here is a typical annotation in a Modelica code. It describes the representation of a parameter in the GUI of the modeling environment and advises the compiler to evaluate this parameter before generating code:

```
parameter RotationSequence sequence
  "Sequence of rotations "
  annotation(Evaluate=true,
             Dialog(tab="Advanced",
             enable=not useQuaternions)
   );
```

Since the information is mostly of no interest for the human reader and an inadvertent manipulation shall be prevented, most modeling environments for Modelica hide annotations from the user by default.

Attributes are also used for meta-information although this information is mostly linked closer to the physical variables (or parameters) of the model: physical units, minimum and maximum boundaries or potential start values for iterative solvers are described by this language construct. The following example contains attributes for the start value, whether they are used for as initial equations, and whether the variable shall be used as state-variable depending on another parameter.

```
SI.AngularVelocity w_a[3] (
 start=Frames.resolve2(R_start,w_0_start),
 fixed=fill(w_0_fixed, 3),
 each stateSelect=
   if enforceStates then
     (if useQuaternions then
        StateSelect.always
      else StateSelect.never)
   else StateSelect.avoid)
```

The two listings above give a quick glance on how meta-information is stored within Modelica. The current solution served fine for more than a decade but it has come to its limitations. We are confronted with two major weaknesses: rising complexity and ambiguity.

The first weakness is simply the sheer amount of definitions that are needed. The current version of the specification devotes already 20 pages for more as 70 annotations and roughly 7 pages for about 10 attributes. The specification is already a long document and further inflation must be prevented. Also we have to keep in mind that the specification is primarily targeted for tool vendors and not for end-users. Most end-users should not have to consult the specification but rather refer to other material.

The second weak point is that the definition in the specification is often not complete. For example, it is usually not defined on which elements an annotation can be placed and only from context one might deduce that annotation "`Evaluate`" makes sense only for primitive data types, whereas annotation "`documentation`" might make sense at many places, but is actually in use only on classes and not on components.

The third weak spot is the ambiguity between the two different concepts. Whether some information belongs to an attribute or to an annotation is not always clear and has often be a discussion point in the design process of the language. For example, `stateSelect` is an attribute used to tell the compiler which variables shall form the state-space of the model. `Evaluation` is an annotation and used to tell the compiler which parameters to evaluate beforehand.

Such discussions are often influenced by the differences in which way attributes and annotations can be accessed. Attributes can be set in (even nested) modifiers, annotations cannot. Vendors can specify their own annotations but they are not allowed to do this for attributes.

2.2 Meta-Information in other Languages

In (*Zimmer 2008*) the handling of meta-information (here denoted as multi-aspect modelling) is discussed for various other modeling languages such as VHDL-AMS or SPICE3. Then another approach is proposed based on the experimental language Sol (*Zimmer 2009*). Here the modeler is given the opportunity to define his/her own annotations by means of environment packages and then can use them by instantiating the components of this package within pre-specified sections of his model. This is conveniently possible because in Sol components have first-class status (*Burstall and Strachey, 2000*) unlike in current Modelica.

Another (although similar) proposal is discussed in (*Zimmer, 2012*). It is based on another experimental language called Hornblower. Also here annotations can be defined within packages and then used within the models. This concept treats annotations like "loosely attached parameters". These are parameters that can be set but do not have to be set. This is possible since such parameters were ensured to always have a reference to a default object.

DOI 10.3384/ECP14096173

Meta information can also be defined in certain programming languages, especially in Java (*Coward 2004*): From http://en.wikipedia.org/wiki/Java_annotation: “*Classes, methods, variables, parameters and packages may be annotated. Unlike Javadoc tags, Java annotations can be reflective in that they can be embedded in class files generated by the compiler and may be retained by the Java VM to be made retrievable at run-time. It is possible to create meta-annotations out of the existing ones in Java*”.

General programming languages often cope much better with meta-information than declarative modeling languages because they own suitable data-structures and often contain already sufficient means for introspection. In Python, there are doc-strings for documentation but they are just predefined class members. Also class or function decorators are used to express a meta-construct on an item, but also these constructs are regular language constructs. In Python there is no need to have language constructs solely devoted to meta-information; instead the regular constructs prove to be sufficient. This shows to us that it is a good idea to reuse regular language constructs for meta-information in Modelica as much as reasonable feasible.

3 Design Goals

In concrete terms, the following goals shall be reached:

- The modeler must be able to define annotations by him- or herself.
- Existing annotations or attributes shall be defined in the same way and removed from the specification (at least as many as possible). This will require to introduce more powerful data structures in Modelica.
- The annotations shall be organized in packages so that an end-user can browse through them and do not need to address the specification anymore.
- The modeler must be able to apply custom annotations in (nested) modifiers.
- Annotation must never be required to be provided by the user. Annotations and its parts are always meta-information that can be given optionally.
- The proposed design must be backwards compatible so that current code is not broken.
- The proposed design can make some language constructs obsolete that can then be removed from the language in the future.
- It must be specified how the meta-information contained in custom annotations is handled for model-export (for instance FMI).

Based on these goals, we have developed a suitable design for a future version of the Modelica language: *Custom annotations*.

4 Design Proposal

The basic idea of our proposal is to use basic Modelica “records” to define custom annotations and then make them better applicable by enabling the use of annotations in modifiers. In this way, new features can be introduced without having to define many new language elements. Note, records are also the basis of nearly all built-in Modelica annotations. A similar concept in (Zimmer, 2012) served as additional starting point. However the transition from an experimental language to a heavily applied language like Modelica demanded several adaptations.

4.1 Use of Records within Annotations

Let us look at an example: Here the Modelica package `OptimSetup` shall be used to define an optimization setup for a model and contains the definition of three record classes that each can be applied as custom annotations.

```
package OptimSetup
  record Tuner "Parameter to be optimized"
    parameter Boolean active = true
    "= true, if parameter is optimized";
    parameter Real    min
    "Optional minimum value";
    parameter Real    max
    "Optional maximum value";
  end Tuner;

  record Minimize
  "Signal that should be minimized"
    parameter Boolean active = true
    "= true, if used as criterion";
    parameter Real    demand = 1.0
    "Value/demand is minimized";
  end Minimize;

  record OptimOptions "Default options for
     the optimization setup"
    parameter String method
    "Optional optimizer method";
    parameter Real    tolerance = 0.001
    "Tolerance of optimization";
  end OptimizationOptions;

  record SimOptions
    "Default options for simulation setup"
    parameter String method;
    parameter Real    stopTime;
  end SimulationOptions;

end OptimSetup;
```

The record and type definitions of package OptimSetup can now be used to describe the setup of an optimization. For example, a parameter can be marked to be a "Tuner" that shall be optimized. The following statement

```
parameter Real p1
  annotation(OptimSetup.Tuner(
               active=true, min=-2));
```

indicates that for a default optimization setup for this model, parameter p1 shall be used as tuner and shall have a constraint p1 >= -2. A custom annotation is identified by the full path name of the Tuner record class. This defines a new instance of the record, together with a modifier on this record. So, conceptually, this custom annotation is equivalent to the following declaration:

```
OptimSetup.Tuner name(active=true,
                       min=-2);
```

and the name of the instance is not defined, because not needed (the identification of the data is via the class name). Exactly in the same way as in a standard declaration, an element of a record needs not to have a default value or a binding equation in its class, and also not in a modifier.

The lookup of a class name inside an annotation is performed on global scope, so always full path names must be given. This simplifies and speeds up the lookup, especially once built-in annotations are defined as custom annotations in a second phase[1].
A custom annotation can be also defined on a class. Furthermore, custom and built-in annotations can be within the same annotation declaration, by using a comma-separated list as usual. For example:

```
model ControlledDrive
  ...
  annotation (Documentation(info="..."),
    OptimSetup(
      OptimOptions(tolerance=1e-3),
      SimOptions(stopTime=4.0, tol=1e-6)
   ));
end ControlledDrive;
```

4.2 Enabling Annotations within Hierarchical Modifiers

So far the only extension to the current Modelica language has been that regular Modelica records can be used within annotations. Taken for itself, this is already a progress but it does not suffice to provide the desired level of functionality. For many applications it is important that annotations can be applied within hierarchical modifiers.

Hence we propose to enable the use of annotation statements within hierarchical modifiers. Here, custom annotations can be either newly constructed or an already defined custom annotation can be modified. Let us look at two corresponding examples:

```
MyCar car(p1 annotation(
           OptimSetup.Tuner(active=false)),
          p2 annotation(
           OptimSetup(Tuner(max=4))),
          p3(min=-3) = 5 annotation(
           OptimSetup.Tuner(min=-2, max=3))
         );
```

In this example, the already defined Tuner.active value of p1 is modified to false. Parameters p2 and p3 are assumed to be defined in MyCar without any annotation. The declarations above introduce new instances of custom annotation OptimSetup.Tuner, and modify these instances with the given values.

Whereas in principle built-in annotations could be applied within hierarchical modifiers too, we propose to restrict this in a first phase because built-in annotations operate on data structures that are unavailable as the standard language elements and then a standard modifier cannot be applied. This is for instance the case for the annotations describing icons that use case records as data elements. For the future, one may solve this problem by enriching Modelica with suitable data structures. This is a topic where discussion is ongoing in parallel.

As with built-in annotations, it is not possible to read and/or use the value of a custom annotation in a Modelica class (only the translator can use the information contained within annotations by either performing appropriate actions or by passing them to its backend).

4.3 About the Use of Hierarchical Records

Since regular Modelica records can be used within annotations, this holds also true for hierarchical records. This is per se not problematic but a few details require a discussion.

Let us suppose we add a hierarchical record Options to our previously present OptimSetup package:

```
package OptimSetup

  record Tuner […]

  record Minimize […]

  record OptimOptions […]
```

[1] When built-in annotations are defined with custom annotations, it is proposed that they are placed, e.g., in ModelicaServices.Annotations, and that this package is inspected first and then the global scope.

DOI 10.3384/ECP14096173

```
  record SimOptions […]

  record Options
    OptimOptions optOpts =
      OptimOptions(method="sqp");
    SimOptions  simOpts;
  end Options;

end OptimSetup;
```

The following example model shows a correct way of using the hierarchical record and a wrong way to do it:

```
model ControlledDrive
  ...
  // correct Modelica code
  annotation(OptimSetup.Options(
      optOpts = OptimSetup.OptimOptions(
                        method="pattern"),
      simOpts(stopTime = 2)
  );

  // wrong Modelica code
  annotation(OptimSetup.Options(
    optOpts(method="pattern"),
    simOpts = OptimSetup.SimOptions(
                          stopTime=2)
  ));
end ControlledDrive;
```

The record `Options` has two element record instances:

The first element `optOpts` is defined with a record constructor. This means there is a binding equation to `optOpts`. Binding equations cannot be modified with a modifier. They can only be replaced by another binding equation. Therefore in the "correct" code, a record constructor is used to define modified elements. Note, a record constructor must return a complete record, and therefore all elements of the record must have a value (either defined in the constructor or the default values from the class).

The second element `simOpts` is defined without a default value or a binding equation. When instantiating it in the "correct" code, a modifier to its elements is given. In this case, not all elements must be modified. In the "wrong" code, a record constructor is used to define modified elements. This would be fine, but element `method` has no default value in the `SimOptions` class, and the record constructor has no input argument for this element, and this is then an error.

Please note that all this is already standard Modelica semantics and holds for standard record declarations, and therefore it shall hold for record declarations in a custom annotation as well. We have just repeated these points for the sake of clarity.

4.4 On Inherited Elements

Since models can be inherited, both the superclass and the subclass may define the same custom annotation. Two cases need to be distinguished:

Case 1: the additional custom annotation is defined in the **extends clause**, such as:

```
model MyCar
  extends Car(
    p1 annotation(OptimSetup.Tuner(
                            active=false )),
    annotation(OptimSetup.OptimOptions(
                            tolerance=1e-3))
  );
```

It seems natural to handle this case as modifier, if the custom annotation was already defined in one of the superclasses (otherwise, a new custom annotation is introduced). Therefore, the model can contain the same custom annotation on one element at most once, and the semantics is well-defined (the semantics of a Modelica modifier).

Case 2: the additional custom annotation is defined as **class annotation**, such as:

```
model MyCar
  extends Car;
  annotation(OptimSetup.OptimOptions(
                          tolerance=1e-4));
end MyCar;
```

In this example it is assumed that in model `Car` the element tolerance is defined as 1e-3 and in model `MyCar` as 1e-4. This creates an ambiguity that needs to be resolved. The Modelica built-in annotations use two different semantics: For the "documentation" annotation, all superclass definitions are ignored. For the "Icon" and "Diagram" annotations, the subclass and superclass definitions are applied after each other (so all of them have an effect).

It is hence proposed to support all these forms by allowing that the same class custom annotation can be used in super- and subclasses. The consequence of this is that an annotation may occur several times in a class. A vector of records can be used to represent the multiple occurences of one annotation in one class. The order of the vector elements thereby corresponds to the order of inheritance. In case of multiple inheritance, the order of the corresponding extends statements is used.

When generating the standardized "output" format of custom annotations in case 2, such definitions can be represented as a vector of records. The target tool that is using this custom annotations has then to decide how to comprise the information contained in all vector elements. In general, useful strategies are:

- Only utilize the latest definition (last element of the vector), and ignoring definitions in superclasses.
- Only utilize the latest definition (last element of the vector), and triggering an error, if equivalent definitions are provided in superclasses (prior elements of the vector).
- Utilizing all definitions by merging them together (following the order of the vector).

4.5 Remaining Issues

4.5.1 Handling of undefined annotations using parameters

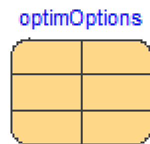
One problem arises from the point that it is often not possible to define a meaningful default value for a custom annotation element. For example, "`method`" of the optimization options defines the optimization method to be used and this depends on the optimization environment where the model and the custom annotations will be imported. It is therefore not possible to define a meaningful default value. Furthermore, the modeler may decide to not define a value for the method in the custom annotation of the Modelica model. For this reason, not all elements of a custom annotation record needs to have a value when it is actually used in an annotation. If an element, such as "`method`" is not explicitly set, then it is not present in the annotation and this implicitly means that the target environment has to cope with this undefined value in a target specific way.

This is not an issue as long as custom annotations are specified directly in the textual layer but providing custom annotations in this way is often not convenient for a user. Instead special blocks or models might be defined where all the relevant information of the custom annotation can be defined in a menu. For example, the `OptimOptions` above might be defined as a `partial model` that is used via inheritance:

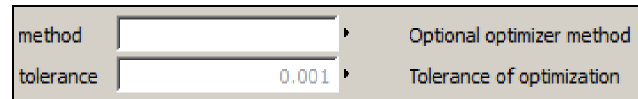
```
partial model OptimOptions
   OptimSetup.OptimOptions optimOptions
         annotation(Placement(..));
   annotation(OptimSetup.OptimOptions(
     method    =optimOptions.method,
     tolerance =optimOptions.tolerance),
     Diagram(...);
end OptimOptions;

model DrumBoiler // shall be optimized
  extends OptimOptions;
    ...
end DrumBoiler;
```

In the diagram layer of `DrumBoiler`, the `optimOptions` are present with a record icon:



and clicking on this icon opens a convenient parameter menu for the definition of the options:



This approach is welcomed by the user, but currently involves one severe drawback: the user has to provide a value also for optional elements, such as `method`, because this element is propagated to the annotation (above: `OptimSetup.OptimOptions (method = optimOptions.method)`). If no value would be provided, then the translator needs to trigger an error. Therefore, the user can no longer express to not define such an optional custom annotation.

For this reason, it is considered to introduce a limited form of "undefined" handling of definitions and modifiers that is handled during the translation process. The goal is

- to remove annotations if they are defined with "undefined" elements.
- to remove modifications performed in a lower hierarchical level.

So far several approaches to this problem have been suggested but a sufficient level of maturity has yet to be reached.

4.5.2 Restricting the application of annotations to certain types

In the current proposal, any annotation might be applied to anything. Also meaningless applications are allowed. For instance the annotation for an optimization tuner cannot only be applied to real numbered parameters but also to Boolean parameters or strings. Even an application to a model class is allowed. Also the annotation for the simulation setup cannot only be applied to model classes as originally intended but also to individual components or variables where it becomes meaningless.

Taken for themselves, such meaningless or ill-applied annotations are not harmful since the metadata cannot corrupt the main code but yet it might be better to restrict the applicability to annotation to give a better guidance where an annotation is supposed to be used.

How such a restriction is best imposed or if it shall be imposed at all, is still open for discussion. One way of doing it, could be to express the restriction by

DOI 10.3384/ECP14096173

annotations themselves in the corresponding annotation record. A new built-in annotation `AnnotationTarget` could serve this purpose. It would contain an enumeration listing for the various possibilities where an annotation can be applied:

```
record AnnotationTarget
   type Target = Enumeration(
     Any,
     ClassDefinition,
     ModelDefinition,
     BlockDefinition,
     ConnectorDefinition,
     RecordDefinition,
     FunctionDefinition,
     TypeDefinition,
     ComponentDeclaration,
     ModelDeclaration,
     BlockDeclaration,
     ConnectorDeclaration,
     RecordDeclaration,
     FunctionDeclaration,
     TypeDeclaration,
     SimpleComponentDeclaration,
     RealDeclaration,
     IntegerDeclaration,
     BooleanDeclaration,
     StringDeclaration,
     InitialEquationAndAlgorithm,
     InitialEquation,
     InitialAlgorithm,
     EquationAndAlgorithm,
     Equation,
     Algorithm,
     ConnectorEquation
   );

   type Prefix = Enumeration(
      Any, None, Constant, Parameter,
      Discrete, Input, Output, Inner,
      Outer, Flow, Stream);

   Target target[:] = {Target.Any};
   Prefix prefix[:] = {Prefix.Any};
end AnnotationTarget;
```

One can now use such an annotation to restrict the applicability of the Tuner annotation:

```
record Tuner "Parameter to be optimized"
  import A = AnnotationTarget;
  parameter Boolean active = true;
  parameter Real    min;
  parameter Real    max;
  annotation(AnnotationTarget(
    target = {A.Target.RealDeclaration},
    prefix = {A.Prefix.Constant,
              A.Prefix.Parameter}
  );
end Tuner;
```

This definition states, that the `Tuner` record is only to be used in an annotation on a `Real` declaration that has a `constant` or `parameter` prefix.

4.6 Summary

In this proposal we reuse and generalize existing concepts from the Modelica language. By doing so, we enable the handling of custom annotations. Let us recapitulate our proposed extensions to the Modelica language:

- Regular Modelica records can be used within annotations.
- Custom annotations can be applied in hierarchical modifiers.
- Hierarchical records are automatically supported in annotations.

For the moment, custom annotations are regarded as an additional feature but for the longer-term future an even extended concept shall be used to define also the standardized annotations. This would unify the language and reduce the complexity of the specification.

5 Using Meta-Information

In the previous section it was proposed how to store meta-information in a Modelica model. Due to its definition, it is not allowed to use this information in the model itself. The question is how a user or a tool can inquire the stored meta-information. Of course, if meta-information is related closely to a simulation model, most likely the respective Modelica translator has to extract and use the information in a tool specific way (for example meta-information related to the graphical representation of the model in the tool).

In this section, we analyze how to extract and utilize user-defined custom annotations for two possible applications: storing meta-information in FMI format and using meta-information in scripting. Of course many further applications are possible.

5.1 Storing Meta-Information in FMI Format

The Functional Mockup Interface (FMI) (*Blochwitz et.al., 2011 and 2012*) is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML-files and compiled C-code. For details see, https://www.fmi-standard.org/. Most Modelica tools support the export and import of models in FMI format. The FMI standard stores all static model information in a file called `modelDescription.xml` in XML format. In particular, the information of all exposed variables are stored here, such as name, data type, unit, description text, variability, causality, etc. The FMI format 1.0 has been published in 2010 and

is supported by more as 40 tools. The release candidate of FMI 2.0 has been published in October 2013.

Since custom annotation variables are basically standard Modelica variables with all the attributes of Modelica variables, it is proposed to just store them as standard FMI variables and mark the "custom annotation" property in the name. In particular, the name of a custom annotation variable shall be:

```
<ComponentName>.annotation.<Custom
   AnnotationFullClassName>.<elementName>
```

Note, "`annotation`" is a reserved keyword in Modelica and therefore a name with "`.annotation.`" cannot be used as component name, so that a name clash between standard Modelica variable names and custom annotation variable names cannot occur.

As previously mentioned, via inheritance the same custom annotation can be used several times in a class annotation. This is handled by always defining a class annotation with a vector name where the index defines the inheritance order (the post-processing tool has then to define how to handle such vectors, e.g., to only use the first one, or utilize all definitions):

```
<ComponentName>.annotation[<i>].<Custom
  AnnotationFullClassName>.<elementName>
```

Example:
The custom annotation proposal has been partially implemented in a Dymola prototype for evaluation. The Modelica model

```
model Vehicle
  parameter Real p1=2 annotation(
                OptimSetup.Tuner(min=-2));
  ...
end Vehicle;

model ControlledDrive
  Vehicle car;
  ...
  annotation(OptimSetup.OptimOptions(
                         tolerance=1e-3));
end ControlledDrive;
```

is stored in the following way in `modelDescription.xml` file with the Dymola prototype:

```
<?XML version="1.0" encoding="UTF-8"?>
<fmiModelDescription
  fmiVersion="2.0"
  modelName="ControlledDrive"
  ...
  >

  ...

  <ModelVariables>
    <ScalarVariable
      name="car.p1"
      valueReference=" 16777216"
      description    ="..."
      causality      ="parameter"
      variability    ="fixed">
      <Real start    ="2"/>
    </ScalarVariable>

    <ScalarVariable
      name="car.p1.annotation.
            OptimSetup.Tuner.min"
      valueReference="0"
      variability    = "constant">
      <Boolean start= "true" />
    </ScalarVariable>

    <ScalarVariable
      name="annotation[1].OptimSetup.
            OptimOptions.tolerance"
      valueReference="0"
      variability    = "constant">
      <Real start="1.0e-3" />
    </ScalarVariable
    ...
  </ModelVariables>
</fmiModelDescription>
```

The "`annotation`" in a name uniquely identifies the component to which this annotation is associated. For example "`car.p1.annotation....`" means that this variable is a custom annotation to variable "`car.p1`". Usually, custom annotation variables are constants or parameters that are **evaluated** during translation and therefore these variables are stored with `variability="constant"` and with a literal value in the xml file.

However, the above scheme is not restricted to this case: A custom annotation may contain time varying variables. In such a case the XML file alone is not sufficient to store the information, but a full FMU (Functional Mockup Unit) is needed, because the code to compute a time-varying variable at a particular time instant needs to be evaluated by the compiled C-code of the FMU.

If a tool already supports the export of a Modelica model in FMI format, then custom annotation variables have just to be included and stored in the standard variable tree.

5.2 Using Meta-Information in Scripting Environments

Typically, user-defined custom annotations are used to setup special analysis or synthesis environments, like optimization, nonlinear model predictive control, Monte Carlo simulation or uncertainty analysis. For this, the underlying model is needed, as well as the analysis-specific custom annotations defined in the model. If the custom annotations are stored in FMI format as proposed in the previous sub-section, the further processing is, in principal, simple: The information is stored in an XML-file and there are many scripting environments available, such as Java,

DOI 10.3384/ECP14096173

JavaScript, Matlab, Python, Ruby, to very easily read XML files and deduce the desired information from it. Therefore, meta-information about non time-varying custom annotations can be deduced in a straightforward way from all these scripting environments. If also time varying variables shall be supported, a scripting environment with FMI support is needed. Typically, the scripting environment will be used, in which the analysis or synthesis task can be directly formulated.

When using Dymola (*Dassault Systèmes, 2014*), there are scripts available to perform offline and online optimization, Monte Carlo simulation, calibration and others. It is natural to simplify the setup of these tasks by defining the model specific parts already in the respective model using custom annotations. Dymola uses the algorithmic part of Modelica as scripting language. Unfortunately, there is no API available to read XML files. This might also be not possible in a generic way, because the data structures supported by Modelica are not powerful enough for such applications.

For this reason, a special new Dymola API function was designed and implemented to read the variable information of an FMI 2.0 XML file (so the information about all exposed signals). The approach is demonstrated in the following code fragments:

```
function generateXXXsetup
  input String fileName;
protected
  ScalarVariable scalarVariable[:] =
        importScalarVariables(fileName);
algorithm
  ...
end generateXXXsetup
```

Function `generateXXXsetup` is a user-defined Modelica function to read an FMI XML file and generate the setup for the respective analysis task. The core is the new Dymola API function `importScalarVariables` that reads the `<ScalarVariable>` part of an FMI XML file and from this information all custom annotations can be deduced. The function returns a vector of records that has a complicated structure: Since Modelica does not have variant records, the different parts of the variable description are just appended. Some parts of the record definition are given below:

```
record ScalarVariable
  import Records.InternalRecords.*;
  import Records.Enumerations.*;
  String            name;
  Integer           valueReference;
  Causality         causality;
  Variability       variability;
  Initial           initialDefinition;
  OptionalInteger   previous;
  Type              variableType;
  RealAttributes    realAttributes;
  IntegerAttributes integerAttributes;
  BooleanAttributes booleanAttributes;
  StringAttributes  stringAttributes;
  IntegerAttributes
               enumerattionAttributes;
end ScalarVariable;

record RealAttributes
  OptionalString  declaredType;
  OptionalString  quantity;
  OptionalString  unit;
  OptionalString  displayUnit;
  OptionalReal    min;
  OptionalReal    max;
  Real            nominal;
  Boolean         unbounded;
  OptionalReal    start;
  OptionalInteger derivative;
  OptionalBoolean reinit;
end RealAttributes;

record OptionalReal
  Boolean present;
  Real    Value;
end OptionalReal;
  ...
```

Many attributes of `<ScalarVariable>` are optional. There is no special data type in Modelica to support optional values. For this reason, records "`OptionalXXX`" are used: Boolean element `present` defines whether the `Value` is defined, or was not given.

By inspecting all "`scalarVariable[i].name`" strings that have `.annotation.` in their name, the desired custom annotations can be deduced and can be utilized to generate the desired default setup of the respective environment.

6 Conclusions

There is a strong need to deal with meta-information in equation-based languages. We presented here a first design in order to enable a better handling of meta-information in Modelica: *custom annotations*.

These can be defined by the user and can be organized within packages. For the long term future, we hope to extend this concept to such a degree that a very high percentage of existing annotations can be covered by one unified concept and the specification can be simplified accordingly. For the near future, we are confident that the proposal will be the base for an enhancement of the Modelica language specification.

It is important to note that the presented design is a design proposal and by no means a definitive design. Also the Modelica Association offers a new process called Modelica Change Proposal (MCP) to

work a proposal into the language specification. The desired design for custom annotations will undergo this process and thereby being reviewed and eventually improved. The aim of this publication is hence to inform the public about the ongoing efforts for handling meta-information in Modelica and not to announce a definitive design decision.

Acknowledgements

The custom annotation proposal in this article was developed within the ITEA2 project MODRIO. Partial financial support of the German BMBF and the Swedish VINNOVA for this development are highly appreciated.

This development was initiated by the MORIO project leader Daniel Bouskela, because associating meta-information with a Modelica model is a core feature needed in the MODRIO project.

Several variants of this proposal have been discussed in Modelica Design Meetings and via the Modelica trac system. Comments and improvement suggestions, especially from (alphabetical list): Volker Beuter, Peter Fritzson, Hans Olsson, Jesper Mattsson, Martin Sjölund, Michael Tiller, are highly appreciated.

A Dymola prototype to export custom annotations in an FMU was implemented by Hans Olsson and Karl Wernersson. Based on this prototype, Hans Olsson proposed to simplify the name lookup as it is described now in this paper.

The Dymola API function to read the variable description from an XML-file into Modelica was implemented in a prototype by Karl Wernersson. Based on this prototype, he proposed to refine the design, especially with respect to undefined attributes.

Custom annotations are already utilized in project MODRIO to develop optimization setups of various optimization problems. This work, carried out by Bernhard Bachmann, Martin Otter, Andreas Pfeiffer and Vitalij Ruge, gave valuable hints for the design of this custom annotation proposal.

References

Blochwitz T., Otter M., Arnold M., Bausch C., Clauß C., Elmqvist H., Junghanns A., Mauss J., Monteiro M., Neidhold T., Neumerkel D., Olsson H., Peetz J.-V., Wolf S. (2011): **The Functional Mockup Interface for Tool independent Exchange of Simulation Models**. Proceedings of the 8th International Modelica Conference, Dresden, March 20-22, pp. 105-114. Download: http://www.ep.liu.se/ecp/063/013/ecp11063013.pdf

Blochwitz T., Otter M., Akesson J., Arnold M., Clauß C., Elmqvist H., Friedrich M., Junghanns A., Mauss J., Neumerkel D., Olsson H., Viel A. (2012): **Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models**. Proceedings of the 9th International Modelica Conference, September 3-5, Munich, pp. 173-184. Download: http://www.ep.liu.se/ecp/076/017/ecp12076017.pdf

Burstall R., Strachey C. (2000): **Understanding Programming Languages**. Higher-Order and Symbolic Computation 13 :52.

Coward D (2004). **JSR 175: A Metadata Facility for the JavaTM Programming Language**. Java Community Process. https://www.jcp.org/en/jsr/detail?id=175#2 (Retrieved 2013-12-09).

Dassault Systèmes (2014): **Dymola 2015 Alpha**. http://www.Dymola.com

Modelica Association (2013): **The Modelica Language Specification, Version 3.3**. Download: https://www.modelica.org/documents/ModelicaSpec33.pdf.

Zimmer D. (2008): **Multi-Aspect Modeling in Equation-Based Languages**. Simulation News Europe, Volume 18, No. 2, pp. 54-61

Zimmer D. (2009): **An Application of Sol on Variable-Structure Systems with Higher Index**. 7th International Modelica Conference, Como, Italy.

Zimmer D. (2012): **A Reference-Based Parameterization Scheme for Equation-Based Object-Oriented Modeling Languages**. 7th Vienna International Conference on Mathematical Modelling, Vienna, Austria.