D5.1.3 – Development of an XML and Modelica-based standardized interface for symbolic exchange of optimal control/NMPC problems based on DAE models

WP 5.1 Nonlinear Model Predictive Control
WP 5 Optimized system operation

MODRIO (11004)

Version 1.0

Date 02/07/2015

Authors

Toivo Henningsson — Modelon AB

Executive summary

This document describes the effort to develop a Modelica XML format for exchange of Modelica models in symbolic form. The format is intended as a complement to the FMI standard, which allows the exchange of models in black-box form. As the scope of Modelica use is expanding, more and more applications are gaining interest where models need to be supplied in symbolic form. So far, to make such applications practical usually necessitates that one have access to and proficiency with the innards of a Modelica compiler. The aim of the format is to decouple compilation from further processing so that applications can have a well-defined interface to Modelica models in symbolic form. The resulting format can be seen as an abstract syntax for (a superset of) Modelica. Modelica compilers usually carry out a number of different kinds of transformations. The format has been designed to be able to represent models at various stages of these transformations to accommodate for the needs of different applications.

The report begins with an outline of the background and motivation for the effort. The guiding design principles are then discussed, followed by an overview of the schema itself. The XML schema is then described in a bottom-up fashion, considering in turn the available constructions in the schema.

The format is also suited to representing models with the aim of Nonlinear Model Predictive Control (NMPC) or for state and parameter estimation. The schema provides for the ability to encode Modelica annotations; the annotations to be used to represent these specific aspects are standardized in other deliverables.

Contents

1. Introduction

As the utilization of Modelica models is expanding from modeling and simulation towards systems

design, it is becoming desirable to apply an ever greater variety of analyses and algorithms to them. This shift in usage is placing increasing importance in tool interoperability, since a single tool can no longer be expected to encompass all desired workflows.

The objective of the Modelica XML effort is to establish a standard XML based exchange format for Modelica models at various stages of processing, complementary to the existing FMI standard. FMI provides for exchange of models in a black box form. In contrast, the Modelica XML format targets exchange of models in symbolic form, which is a prerequisite to being able to apply many algorithms of interest. The format is described in the form of XML Schemas [1].

2. Modelica XML schemas

The schemas aim to create a single unified format to represent Modelica models that have been processed to varying degrees, including at least:

I. **Modelica source code.** This format is an abstract representation of Modelica code and should provide a one to one mapping between Modelica source code and a corresponding XML representation.

II. **Instantiated hierarchical models.** In this model representation, Modelica modifiers have taken effect, and the hierarchical structure is preserved.

III. **Flattened Modelica models.** This model representation is flat in the sense that it does not contain hierarchical model elements. It may, however, contain array declarations.

IV. **Flattened transformed models.** In this representation, array variables and equations have been scalarized, alias variables have been eliminated and DAE index has been reduced, if needed.

The format should be uniform and minimalistic, and the resulting XML should not be overly verbose. It should eventually provide full coverage of the Modelica 3.x language, though the focus of current version is to cover flattened representations.

Draft XML schemas that cover a majority of Modelica constructs were first presented at the 79th Modelica Design Meeting in Coventry, June 3-5, 2013, and have since been developed further, in conjunction with a prototype implementation of export and import in JModelica.org. The schemas have taken inspiration from earlier unified XML schemas created by Hilding Elmqvist, and minutes from the Modelica Association discussing various aspects of how to represent (flattened) Modelica constructions in an XML format, but have been developed from the ground up within MODRIO. The have been made available online at https://github.com/modelica-association/ModelicaXML.

The schemas are modeled quite closely on the syntax of Modelica, and it is with the background of the semantics of the Modelica language that the format should be understood. However, when Modelica syntax allows several representations that are semantically equivalent, an effort has been made to have a single representation in the XML schemas. To leave some room for extensions to the Modelica language and for unanticipated use cases, as well as to make the format simpler and more uniform, the schemas aim to model a superset of valid Modelica by reusing the same construction wherever meaningful.

To reduce verbosity, the schemas make extensive use of the `<xs:group>` construction in XML Schema. E.g., the group named `Expression` contains a choice between different types of expression nodes. This allows other nodes to contain a sequence of expressions without wrapping each in an intermediate node.

As an example, the expression `x+8.5`, where `x` is a local variable, is described by the XML code

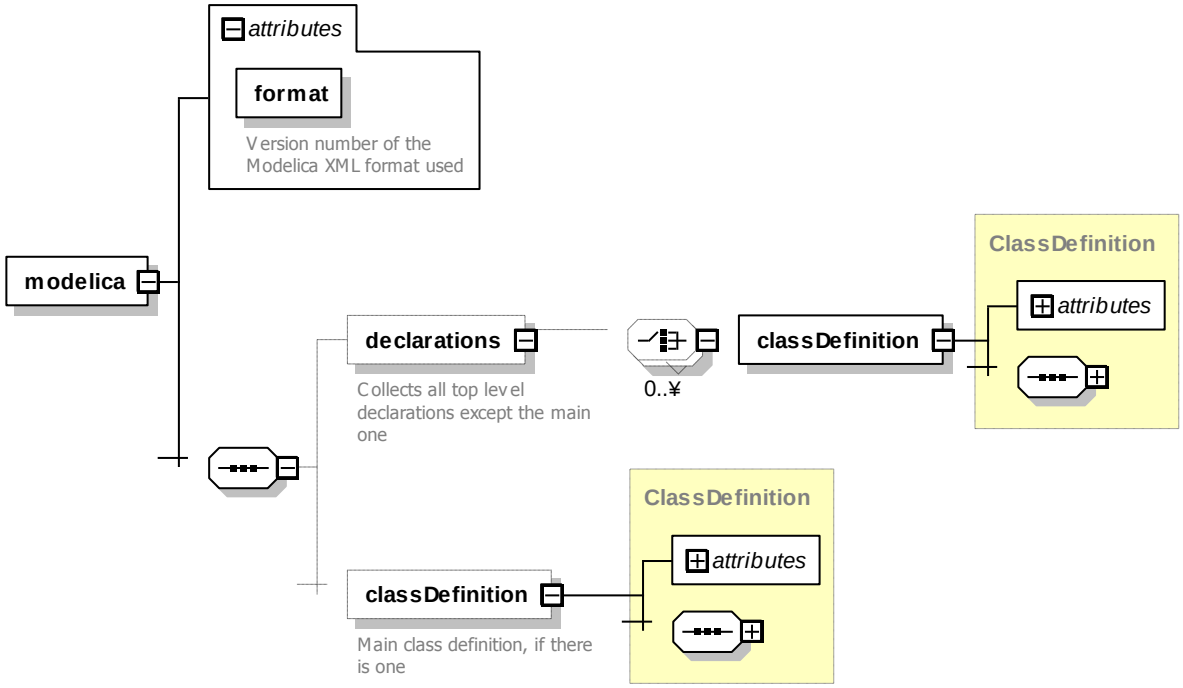
```
<apply builtin="+">
  <local name="x"/>
  <real value="8.5"/>
</apply>
```

where level + expression wraps subexpressions for `x` and `8.5`.

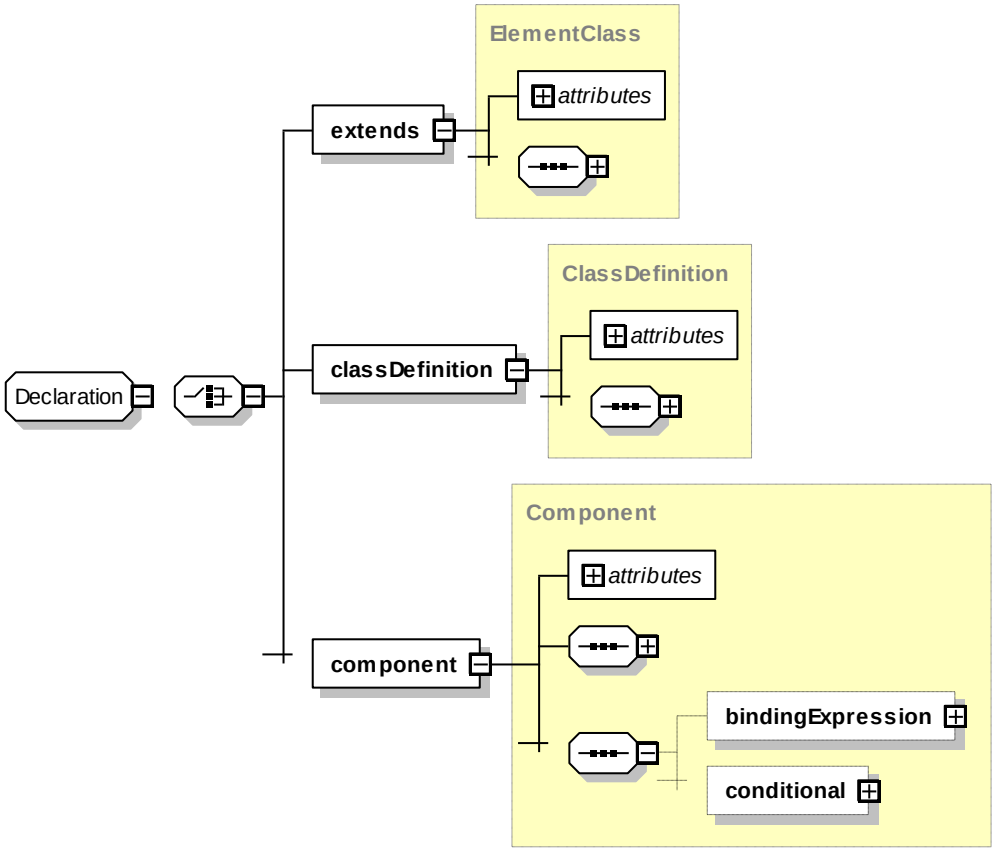
2.1. Overview

The XML schemas are divided into four parts: expressions, equations and statements, types (notably classes), and top level, each building on the ones before it. Though there is considerable overlap in form between equations and statements, the differences in semantics as well as form were considered too great to reuse the same definitions. Notably, classes can be represented separately from naming them, allowing models at the level of instance trees and flattened models to represent instances directly.

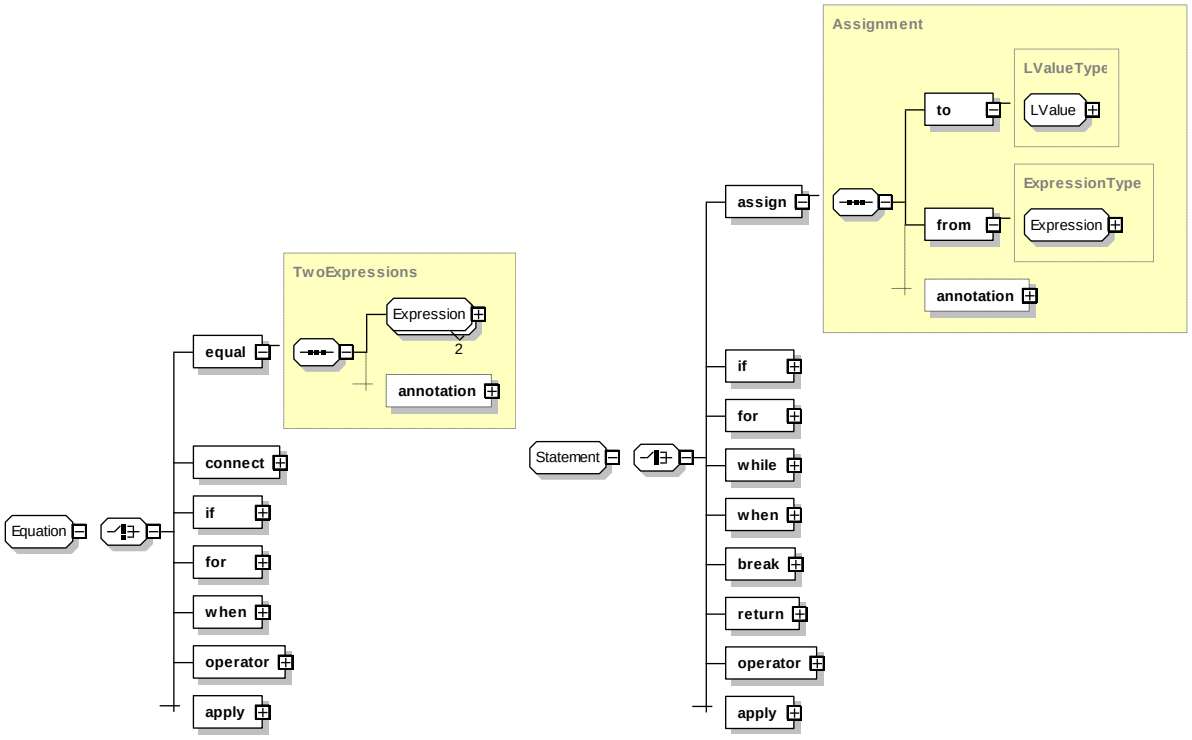
The root element in an XML file that adheres to the schema is given by the `<modelica>` element, which may contain a main class definition and a collection of supporting class definitions, depending on the application.



The contents of a class are described by the complexType (the type of an XML tag, see [1]) `ClassContents`, which includes declarations, equations and algorithms:



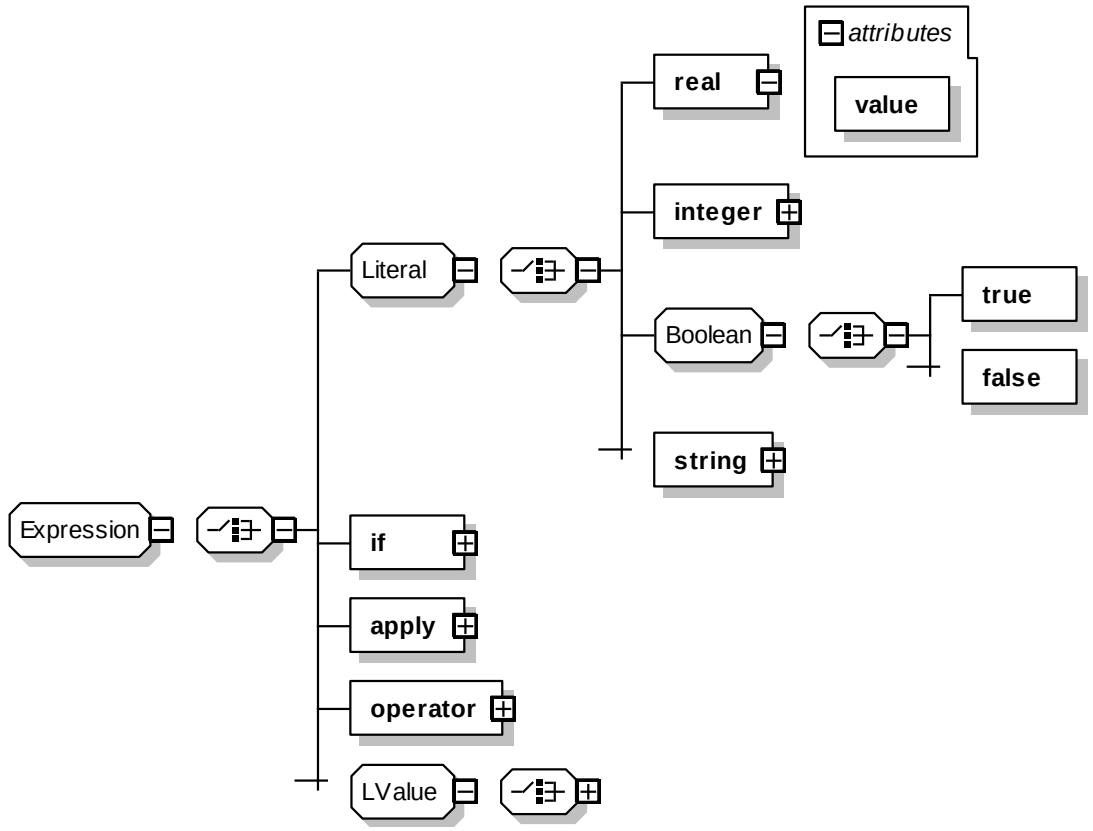
Equations and statements are described by the `Equation` and `Statement` groups, respectively. They differ mainly in that the primitive equations and statements are different (equalities vs. assignments), and that statements allow more constructions.



Expressions are described by the `Expression` group, with which we begin our detailed description of

the schema below.

3. Expressions

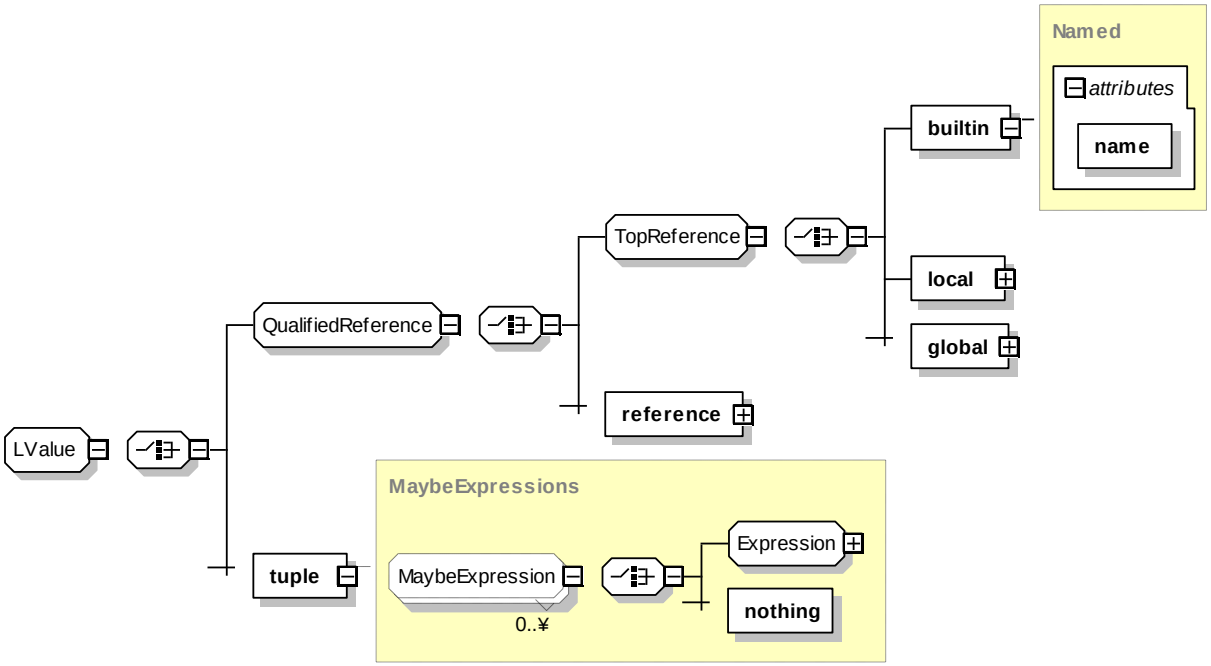


All expressions are represented using the `Expression` choice group (as seen above), including sub-expressions. All expressions may not be semantically valid in all places.

3.1. Literals

Literals are expressed simply as e.g. `<real value="5.5">`, `<integer value="11">`, `<string value="abc">` etc. for compactness, boolean literals are represented explicitly as `<true/>` or `<false/>`.

3.2. L-values

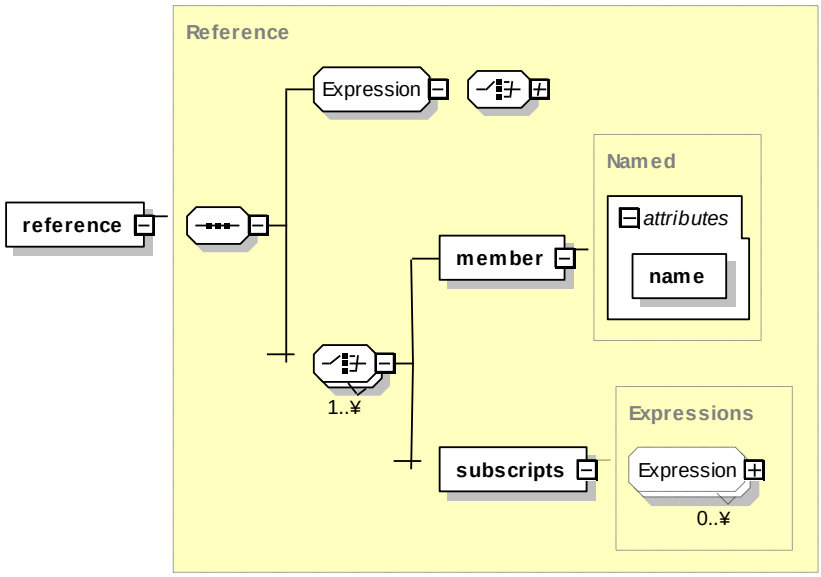


L-values are the only kinds of expressions that may appear at the left hand side of an assignment.

3.2.1. Top references

The simplest L-values are top references using the `<builtin>`, `<local>`, and `<global>` elements. These references denote lookup of the provided name in the built-in, local, or global namespace respectively, e.g. `<builtin name="Real">` for the built-in Modelica type `Real`, or `<local name="x">` for a local variable `x`.

3.2.2. Member and subscript references



The `<reference>` element is used to represent references based on existing expression: member access such as `m.x` using the `<member>` element, and subscripts such as `v[5]` using the `<subscripts>` element. Nested references can be listed within the same `<reference>` element in

the order that they are applied, e.g. `obj.x[k, 3]` is represented as

```
<reference>
  <local name="obj"/>
  <member name="x"/>
  <subscripts>
    <local name="k"/>
    <integer value="3"/>
  </subscripts>
</reference>
```

The base expression to which member lookup and/or subscripts are applied can be any kind of expression, unlike Modelica, where the base expression of a component reference may only be a top reference.

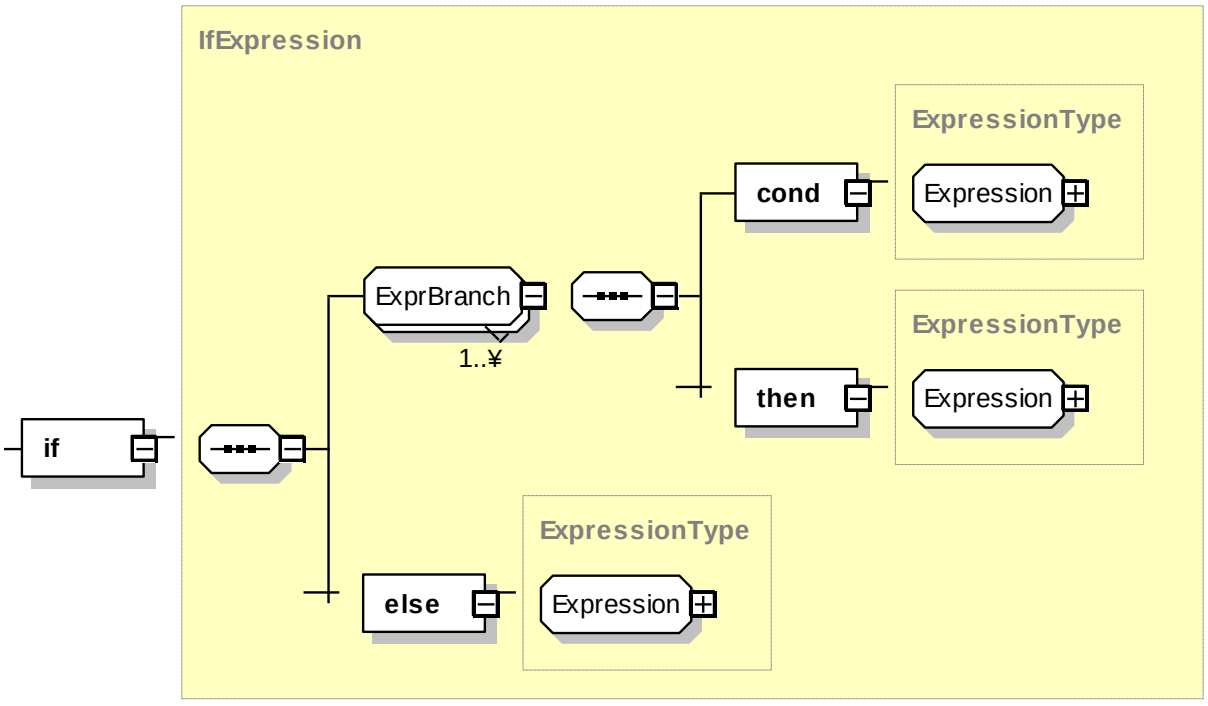
3.2.3. Tuples

The `<tuple>` element is the left hand side of function call equations such as `(x,,y) = f(z)` and function call statements such as `(x,,y) := f(z);` both to invoke functions with multiple return values. The tuple `(x,,y)` is represented as

```
<tuple>
    <local name="x"/>
    <nothing/>
    <local name="y"/>
</tuple>
```

where the `<nothing/>` element signifies that the corresponding return value is ignored.

3.3. `if` expressions

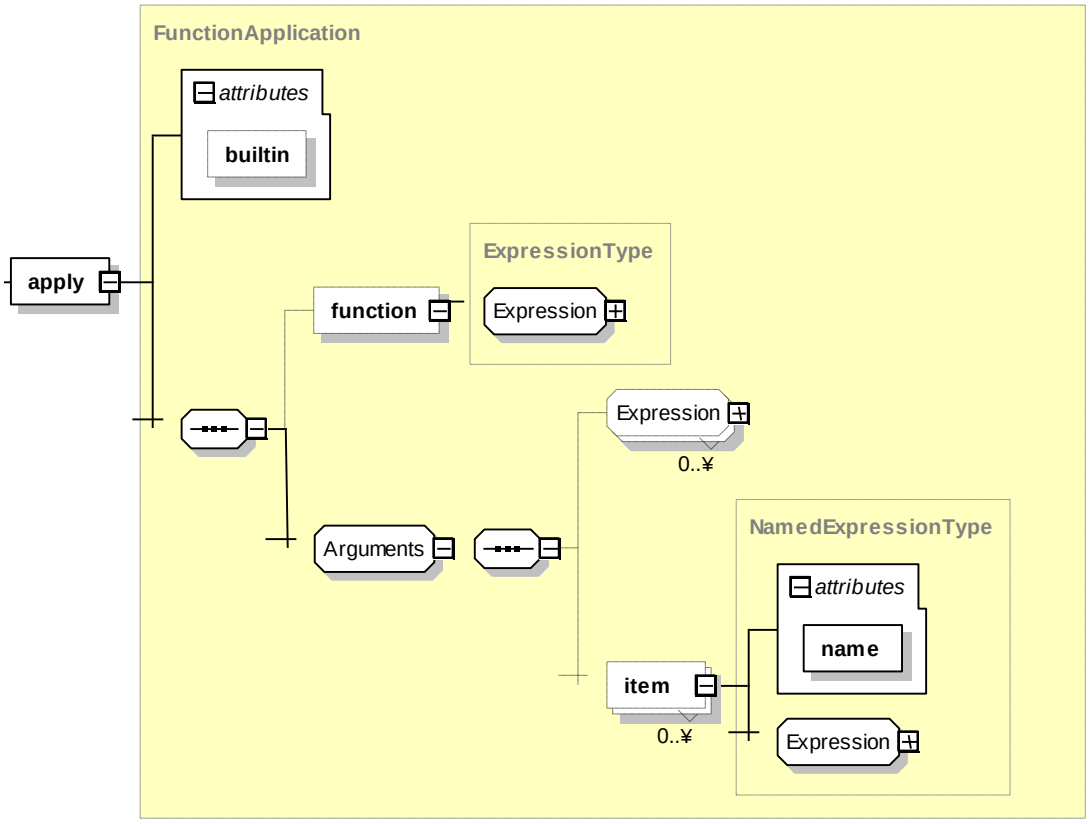


`If` expressions are represented by an alternation of `<cond>` and `<then>` elements, followed by an `<else>` element; each containing a single expression. The value of the `if` expression is the value of

the first <then> expression whose corresponding <cond> expression evaluates to true, or the value of the <else> expression if none of them do. Example:

```
<if>
  <cond>
    <local name="p"/>
  </cond>
  <then>
    <local name="x"/>
  </then>
  <cond>
    <local name="q"/>
  </cond>
  <then>
    <local name="y"/>
  </then>
  <else>
    <real value="0"/>
  </else>
</if>
```

3.4. Function invocation expressions



Function invocations are represented using the `<apply>` element, and may be used to represent anything that is a mathematical function, i.e., the return value must depend only on the values of the arguments. Positional arguments are listed in sequence, followed by named arguments wrapped in `<item name="argname">` elements.

The function to be invoked can be specified in one of two ways:

- using the `<function>` element, which takes an expression that denotes the function.
  Example: the function invocation `f(x, positive=true)` is represented as

```
<apply>
  <function>
    <local name="f"/>
  </function>
  <local name="x"/>
  <item name="positive">
    <true/>
  </item>
</apply>
```

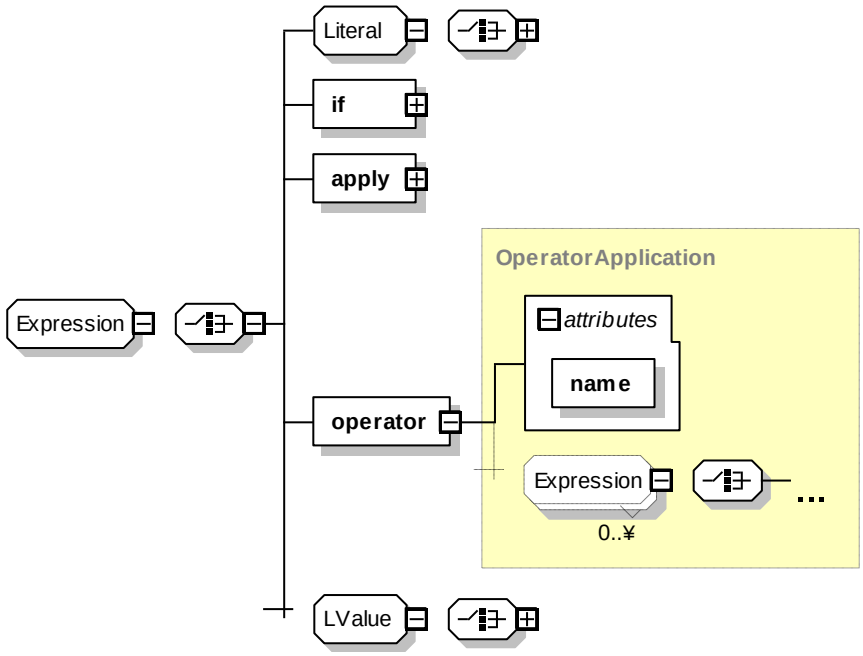
- using the `builtin` attribute. This is a short form to invoke built-in functions, including arithmetic operations. Example: `x+8.5` can be represented as

```
<apply builtin="+">
  <local name="x"/>
  <real value="8.5"/>
</apply>
```

  Note that it is also valid to represent the same expression using the equivalent long form

```
<apply>
  <function>
    <builtin name="+"/>
  </function>
  <local name="x"/>
  <real value="8.5"/>
</apply>
```

3.5. Operator expressions



Operator expressions are used to represent operators that are not mathematical functions of their arguments' values, such as `der`, `delay`, and `noevent`. The format is a subset of that allowed for function invocations with `<apply>`. The allowable values for the `name` attribute have not been restricted in the schema, in order to allow for future extensions. Example: `der(x)` is represented as

```
<operator name="der">
  <local name="x"/>
</operator>
```

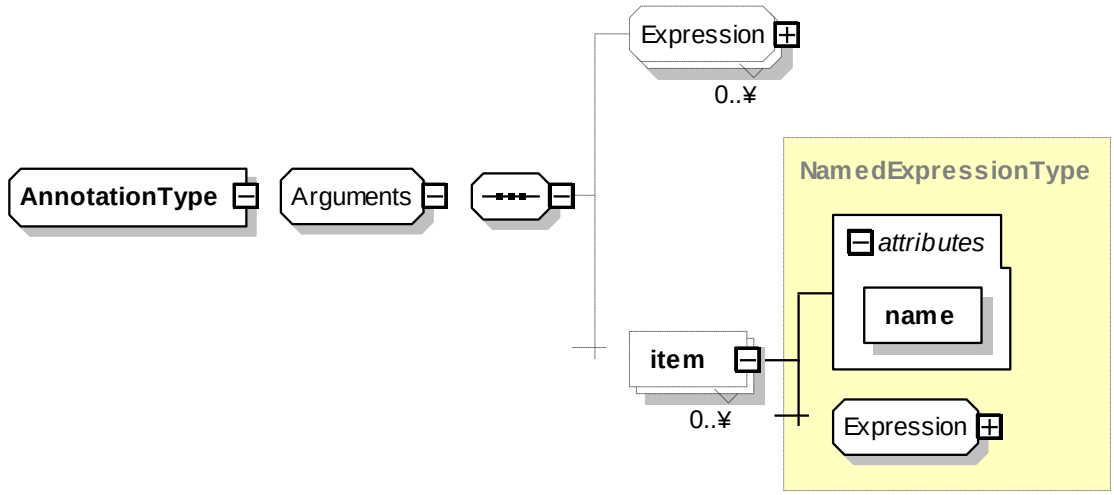
4. Equations and statements

Equations and statements are in many ways similar, but different enough that different constructions are used to represent them. The main differences motivating this split are that

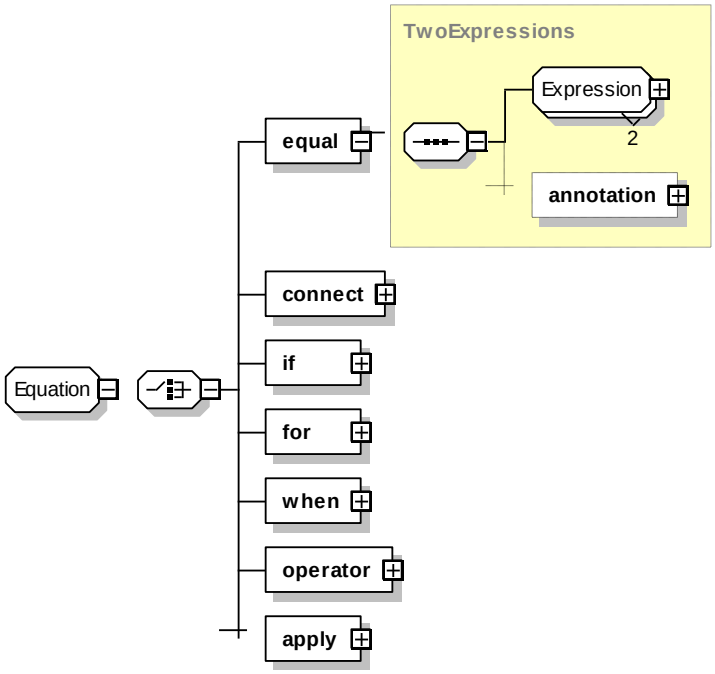
- The semantics of equations and statements are different since equations are simultaneous and statements are sequential.
- The form of primitive equations and statements are different: a primitive assignment statement `lhs := rhs` requires the left hand side to be assignable, while a primitive equation `lhs = rhs` places no such restrictions. For this reason, only L-value elements are allowed at the left hand side of an assignment.

4.1. Annotations

Equations and statements, as well as classes, may have annotations on them. Annotation contents are represented in the same way as function arguments, and may include positional and named arguments.



4.2. Equations



Equations make up the contents of equation sections, and are also contained inside compound equations (`if`, `for`, and `when` equations). Expression elements `<operator>` and `<apply>` are also allowed directly as equations.

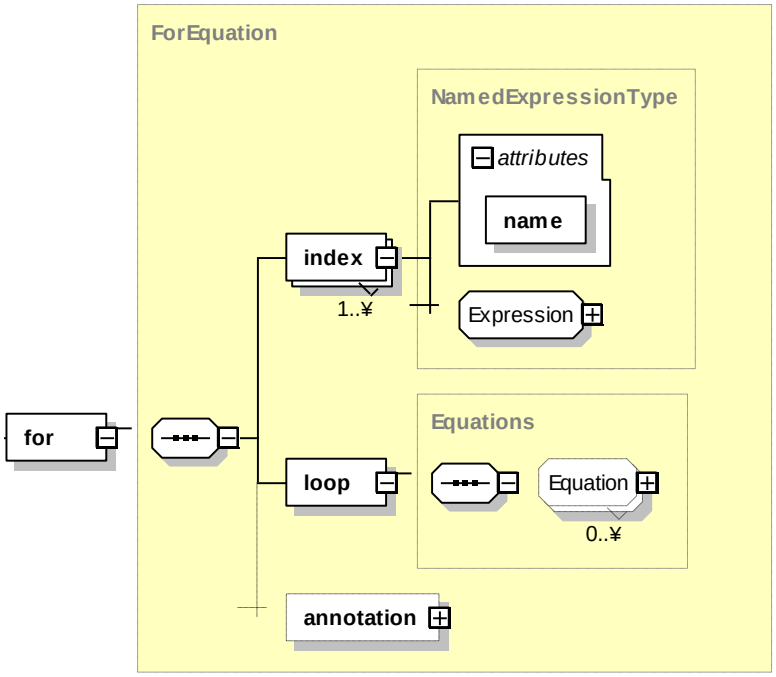
`<equal>` equations represent primitive equality equations `lhs = rhs`. Function call equations with multiple return values are represented with a `<tuple>` as the left hand side and `<apply>` as right hand side. Example: the equation `x = 1` is represented as

```
<equal>
  <local name="x"/>
  <real value="1"/>
</equal>
```

Connections are represented in a very similar way using `<connect>` equations.

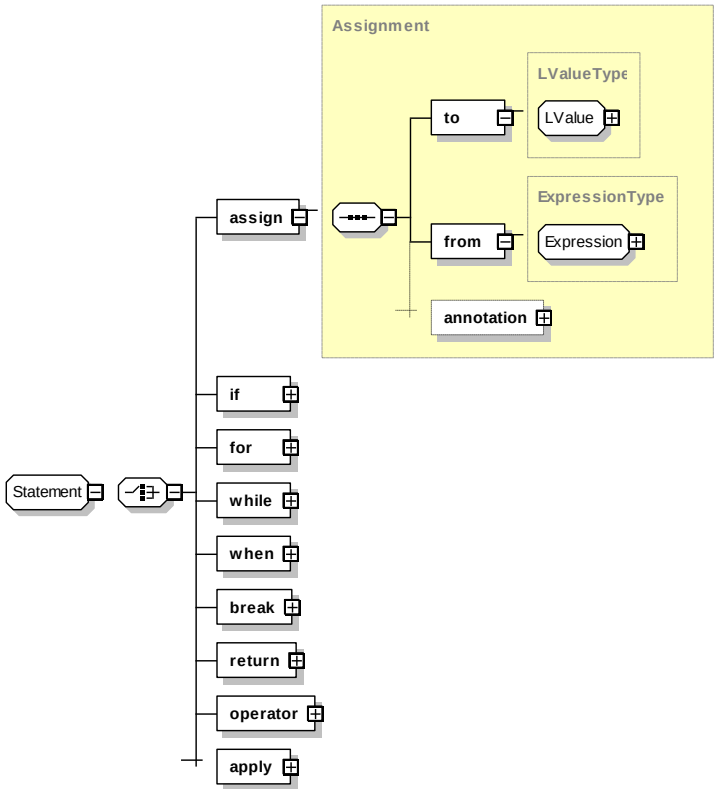
`<if>` equations have the same format as `<if>` expressions, except for that the `<then>` and `<else>` branches are equations instead of expressions.

<when> equations have the same format as <if> equations except that there is no <else> branch (the semantics are different of course).



<for> equations introduce a number of index variables and corresponding ranges, followed by a body of equations.

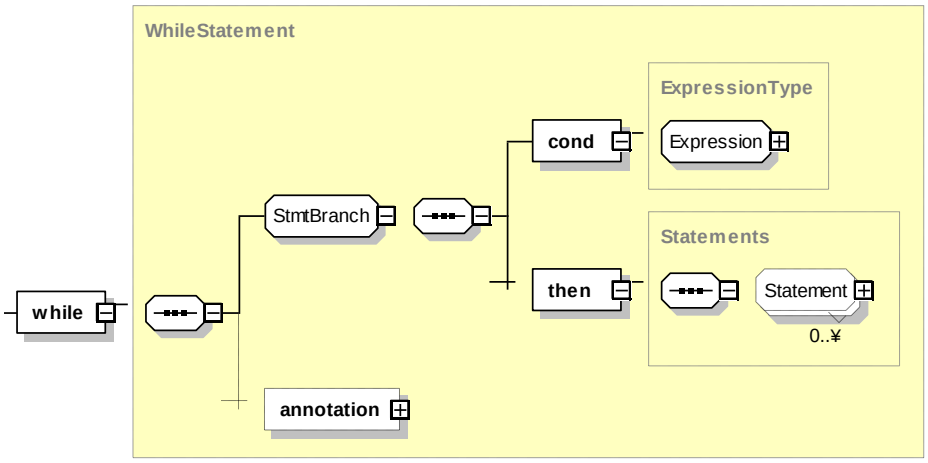
4.3. Statements



In analogy with equations, statements make up the contents of algorithm sections, and are also contained inside compound statements (`if`, `for`, `while`, and `when` statements). Expression elements `<operator>` and `<apply>` are also allowed directly as statements. All kinds of equations except primitive equalities and connections have direct counterparts as statements, but `while`, `break`, and `return` exist only as statements.

`if`, `for`, `when`, `operator`, and `apply` statements have a completely analogous format to the corresponding equations, except for that the first three contain nested statements instead of nested equations.

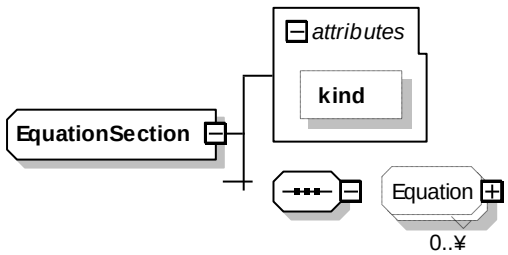
`break` and `return` statements are represented by empty elements `<break>` and `<return>`, respectively.

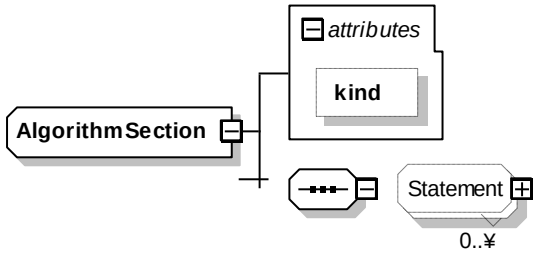


`while` loops are represented using the `<while>` element, which contains a single loop condition in the `<cond>` element and a body in the `<then>` element. Example:

```
<while>
  <cond>
    <local name="c"/>
  </cond>
  <then>
    <assign>
      ...
    </assign>
  </then>
</while>
```

4.4. Equation and algorithm sections





Equation and algorithm sections are represented by the complex types `EquationSection` and `AlgorithmSection`, containing zero or more equations and statements, respectively. The attribute `kind` can be `default`, `initial`, or `parameter`, where `default` corresponds to a regular equation or algorithm section and `initial` corresponds to an algorithm or equation section.

4.4.1. Parameter equations

An equation section with the `kind` attribute set to `parameter` is a *parameter equation section*. A parameter equation section is very similar to a list of binding equations and should be sorted in evaluation order. An example of when a parameter equation can be used but not a binding equation is with a function `f` that returns a `Real[2]` and a Modelica model

```
model m
  parameter Real p1 = 2;
  parameter Real[2] p2 = f(p1);
end m;
```

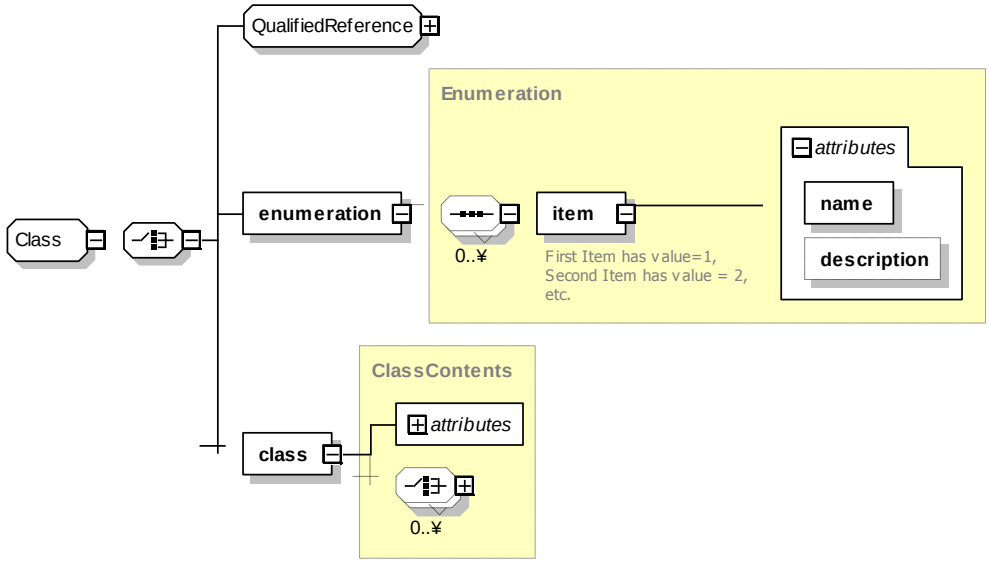
When scalarizing the parameter `p2`, its binding equation can be transformed into a parameter equation

```
{p2[1], p2[2]} = f(p1);
```

It is yet unclear whether there is a need for a separate parameter algorithm section.

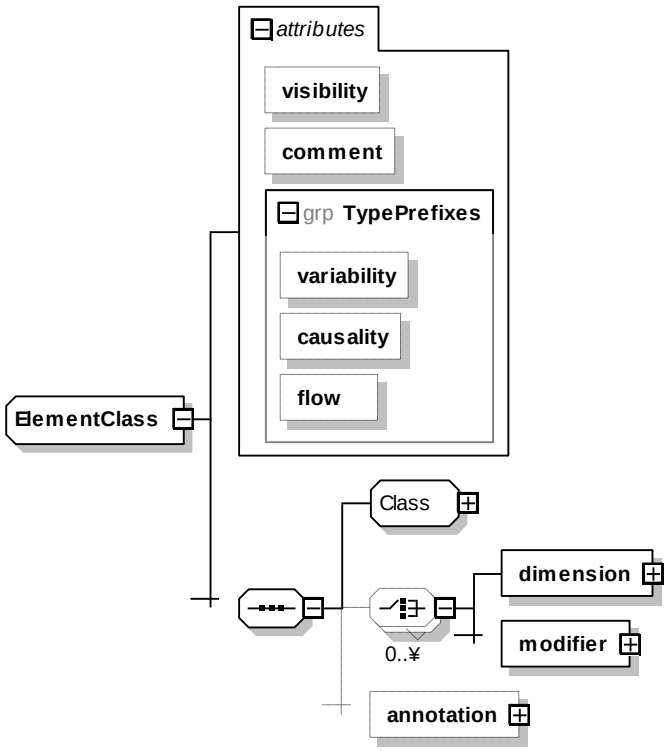
5. Types

Types include both primitive types, array types, and all kinds of classes that can be represented in Modelica, including models and functions. They are used to represent whole models as well as type definitions and types of components inside classes, and the types that they extend from. Consequently, complex types are built from simpler types, just as in Modelica itself.



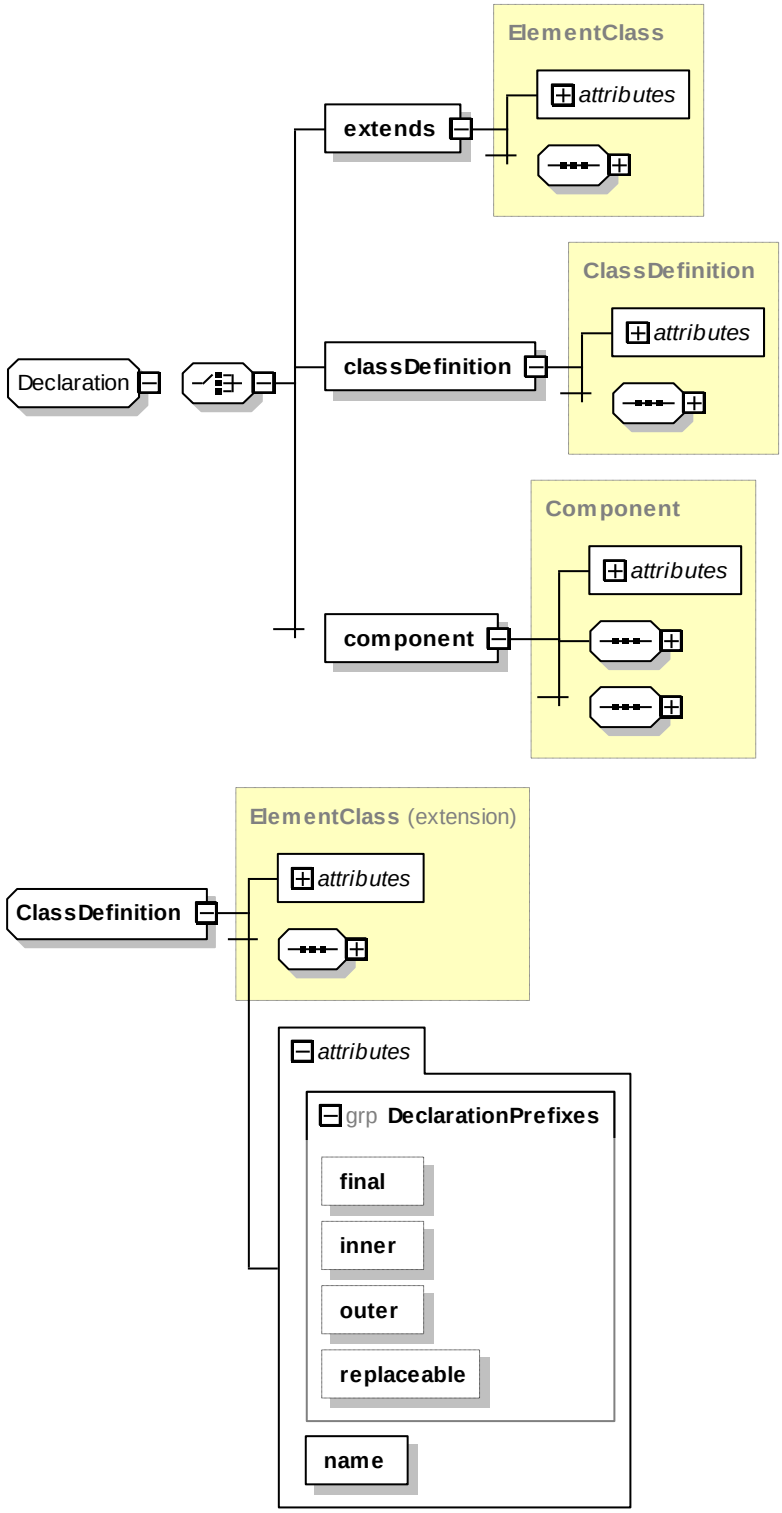
The choice group `Class` represents a basic type, which can either be a reference to a built-in or previously defined type (using the `QualifiedReference` choice group from the expression

schema), a newly defined enumeration, or a class type. Class types are described by the `ClassContents` complex type, which will be defined at the end of this section.

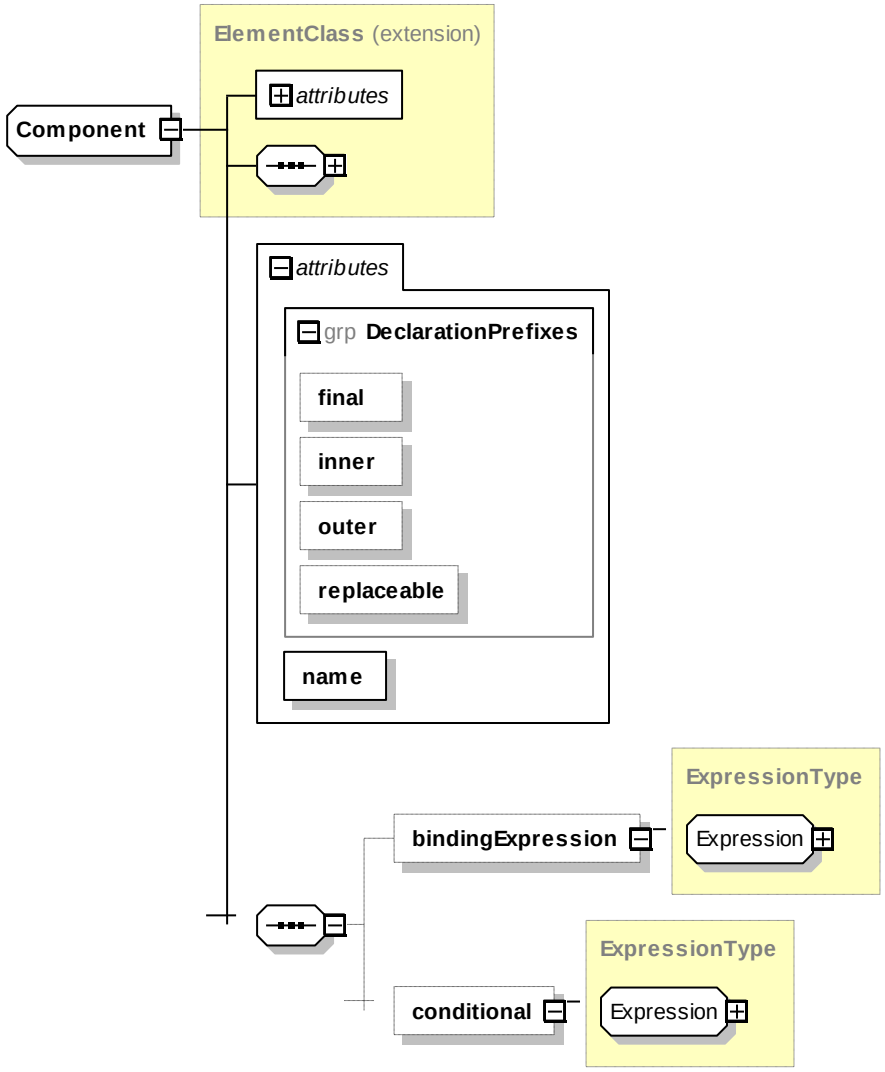


The complex type `ElementClass` represents the type of an element (extends, component or class definition), and adds a number of attributes as well as optionally wrapping array dimensions and modifiers around the core `Class` representation.

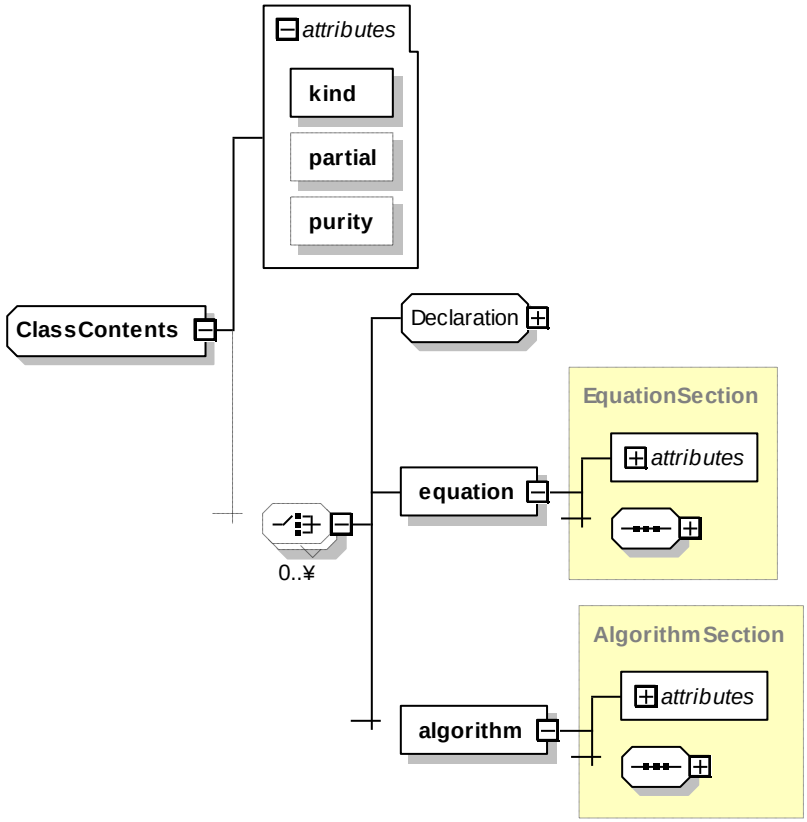
The `Declaration` choice group is used to represent a single element in a Modelica class (an extends-, component or class definition). An `<extends>` element only needs to specify a type, and so uses `ElementClass` directly. Class definitions and component declarations need specify additional information, and so use complex types `ClassDefinition` and `Component`, based on `ElementClass`.



`ClassDefinition` adds declaration prefixes and a name.



`Component` also adds an optional binding expression and condition. The expression supplied to `<conditional>` should be a boolean expression; the component is only taken to be part of the class when the expression evaluates to true.



Finally, we can describe the complex type `ClassContents`, which is used to describe the contents of a class, and used in the `<class>` option of the `Class` choice group described in the beginning of this section.

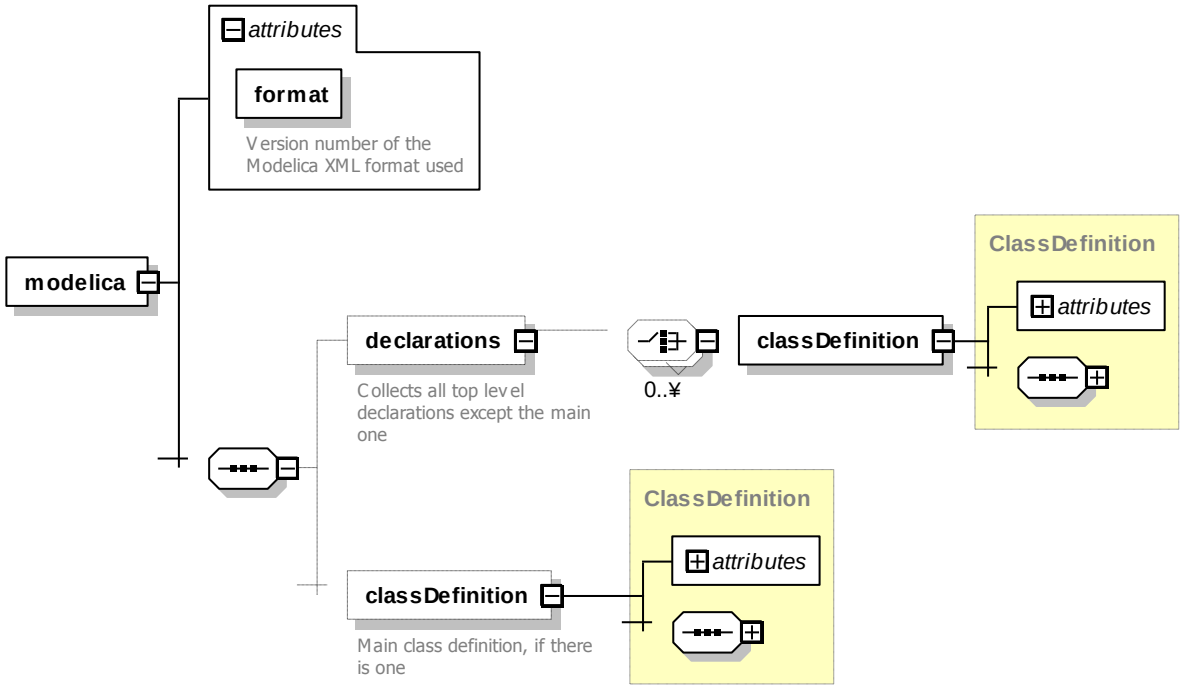
The class contents are made up of a number of element declarations, equation sections, and algorithm sections, described by the choice group `Declaration`, and the complex types `EquationSection` and `AlgorithmSection` respectively.

The class types also have attributes:

- `kind` can be one of `class`, `model`, `record`, `operator record`, `block`, `connector`, `expandable connector`, `package`, `function`, `operator function`, and `function`.
- `partial` can be `true` or `false`.
- `purity` can be `default`, `pure`, or `impure`.

6. Top level: the <modelica> element

The schema defines the top level `<modelica>` element, which is the only top level (root) element allowed by the schema.



The `<modelica>` element has a required `format` attribute to specify the version of the XML schema used to encode the file. The format version for the schema described in this document is `"1.0"`.

The element may contain a number of class definitions as given through the `ClassDefinition` complex type; this includes a name for each class definition.

- For some applications, there will be a main class; its definition should then be given last. Supporting class definitions can be given inside the `<declarations>` element before it.
- In other applications, there may be a number of class definitions where none is more central than the others. In such cases, all class definitions should be provided within the `<declarations>` element.

7. Conclusion

The XML schema presented above is capable of representing most Modelica constructs in a symbolic format, including flattened and scalarized Modelica models. Further work for future versions is to add support for a few remaining constructions.

8. References

[1] XML Schema standard, http://www.w3.org/standards/xml/schema