Adrian Pop, Peter Fritzson
PELAB, Linköping University:
ModelicaXML: A Modelica XML Representation with Applications
pp. 419-430

ModelicaXML: A Modelica XML Representation with Applications

Adrian Pop Peter Fritzson

PELAB, Programming Environment Laboratory,
Department of Computer and Information Science,
Linköping University, SE-58183, Linköping, Sweden
adrpo@ida.liu.se, http://www.ida.liu.se/~adrpo
petfr@ida.liu.se, http://www.ida.liu.se/~petfr

Abstract

This paper presents the Modelica XML representation with some applications. ModelicaXML provides an Extensible Markup Language (XML) alternative representation of Modelica source code. The language was designed as a standard format for storage, analysis and exchange of models. ModelicaXML represents the structure of the Modelica language as XML trees, similar to Abstract Syntax Trees (AST) generated by a compiler when parsing Modelica source code. The ModelicaXML (DTD/XML-Schema) grammar that validates ModelicaXML documents is introduced. We reflect on the software-engineering analyses one can perform over ModelicaXML documents using standard and general XML tools and techniques. Furthermore we investigate how can we use more powerful markup languages, like the Resource Description Framework (RDF) and the Web Ontology Language (OWL), to express some of the Modelica language semantics.

1 Introduction

The structure of a Modelica model can be derived from the source code representation, by using a Modelica compiler front-end (the lexical analyzer and the parser).

The compiler front-end takes the source code representation and transforms it to *abstract syntax trees* (AST), which are easier to handle by the rest of the compiler. As pointed out in [20], a clear disadvantage of this procedure is the need of embedding a compiler front-end in every tool that needs access to the structure of the program. Writing such a front-end for an evolving and advanced language like Modelica is not trivial, even with the support of automated tools like Flex/Bison or ANTLR [28].

To overcome these problems, a standard, easily used, structured representation is needed. ModelicaXML is such a representation that defines a structure similar to abstract syntax trees using the XML markup language.

This representation provides more functionality than a typical C++ class library implementing an AST representation of Modelica:

- Declarative query languages for XML can be used to query the XML representation.
- The XML representation can be accessed via standard interfaces like Document Object Model (DOM) [3] from practically any programming language.

The usages of the ModelicaXML representation for Modelica models, combined with the power of general XML tools, will ease the implementation of tasks like:

- Analysis of Modelica programs (model checkers and validators).
- Pretty printing (un-parsing).
- Translation between Modelica and other modeling languages (interchange).
- Query and transformation of Modelica models.

Although ModelicaXML captures the structured representation of Modelica source code, the semantics of the Modelica language cannot be expressed without implementing specific XML-based tools. To address this issue we have investigated the benefits of using other markup languages like the Resource Description Framework (RDF) and the Web Ontology Language (OWL). These languages, developed in the Semantic Web Community [13], are used to express semantics of data in order to be automatically processed by machines. We believe that using such technology for Modelica models would enable several applications in the future:

- Models could be automatically translated between modeling tools.
- Models could become autonomous (active documents) if they are packaged together with the operational semantics from the compiler, and therefore, they could be simulated in a normal browser.
- Software information systems (SIS) could more easily be constructed for Modelica, facilitating model understanding and information finding.
- Model consistency could be checked using Description Logic (DL) [2].
- Certain models could be translated to and from the Unified Modeling Language (UML) [15].

The paper is structured as follows: Related work is presented in Section 2. Modelica, XML and the ModelicaXML Document Type Definition (DTD) are discussed in Section 3. In Section 4 we present the software-engineering tasks one can perform on the ModelicaXML representation using XML tools and technologies. Section 5 investigates the use of RDF and OWL for representing semantics of Modelica models. Conclusions, future research directions and summary of the work are presented in Section 6.

2 Related Work

In the field of general programming languages, JavaML [20] has been developed as structured representation of Java source code. JavaML emphasizes the power of such structured representation when leveraging XML tools. When it comes to domain specific modeling languages, there are several [21, 22, 27] approaches to specifying models in XML. These approaches deal with model transformation, exchange and management (regarding adaptation to already existing simulation tools) or with code generation from the intermediate XML representation to C++. Our interest focuses more on providing flexible and general software-engineering tooling support for the Modelica programmer. For this purpose the ModelicaXML is covering the full Modelica language [8, 23], including algorithm sections and expression operators. Furthermore, we consider more powerful markup languages for defining some of the Modelica static semantics and we discuss future use of such Semantic Web technologies.

3 Modelica XML Representation

Modelica [8, 23] is an object-oriented language used for modeling of large and heterogeneous physical systems. For modeling with Modelica, commercial software products such as MathModelica [7] or Dymola [4] have been developed. However, there are also open-source projects like the OpenModelica Project [24]. Our research is part of the OpenModelica Project and aims at providing a more flexible framework with the use of XML technologies.

In sub-section 3.1 we briefly introduce the concepts of XML and DTD and give an example of a Modelica model with its ModelicaXML representation.

3.1 The eXtensible Markup Language (XML)

The Extensible Markup Language (XML) [5] is a standard recommended by the World Wide Web Consortium (W3C). XML is a simple and flexible text format derived from Standardized Generalized Markup Language (SGML) [14]. The XML language is widely used for information exchange over the Internet. The tools one can use for parsing, querying, transforming or validating XML documents have reached a mature state. Such tools exist both as open-source projects and commercial software products.

A small example of an XML document is shown below:

```
<?xml version="1.0"?>
<!DOCTYPE persons SYSTEM "persons.dtd">
<persons>
  <person job="programmer">
    <name>John Doe</name>
    <email>
      grigore@none.ro
    </email>
  </person>
    …
  <person job="manager">
    <comment>Classified</comment>
  </person>
</persons>
```

An XML document is simply a text in which the information is marked up using tags. The tags are the names enclosed in angle brackets. For easy identification we show *elements* in **bold** face and *attribute* names in *italics* throughout the XML example. The information delimited by <**persons**> and </**persons**> tags is an XML element. As we can see, it can contain other elements called <**person**> that nests additional elements within itself.

The attributes are specified after the tag as an unordered name/value list of `name="value"` items. In our example, the attribute *`job`* with the value `"programmer"`.

The first line states that this is an XML document. The second line express that an XML parser must validate the contents of the elements against the Document Type Definition (DTD) [18] file, here named `"persons.dtd"`. The DTD provides constraints for the contents much like grammars used for programming languages.

There are two criteria to be met in order for an XML document to be valid. First, all the elements have to be properly nested and must have a start/end tag. Second, all the contents of all elements must obey their DTD grammar specifications.

We will define a DTD for the above example:

```
<!-- the person.dtd file  -->
<!ENTITY % person-job-attribute
   "job(programmer|manager)
    #REQUIRED">
<!ELEMENT persons (person*)>
<!ELEMENT person
      ((name+, email*) | comment+)>
<!ATTLIST person
      project CDATA #IMPLIED
      &person-job-attribute;>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT comment (#PCDATA)>
```

The above DTD defines one entity, four elements, and one attribute list containing two attributes. The entities are underlined, **bold** is used for elements, and attributes are specified in *italics*.

The entity `(ENTITY)` declaration defines `person-job-attribute` as a text value that can be used anywhere inside the DTD and the XML document. The XML parser will replace the entity with its defined text where it is used. The principal element `(ELEMENT)` declared in DTD is **`persons`** and has zero or more elements **`person`** nested inside. The special characters inside the element definitions are "`*`" meaning: zero or more, "`|`" meaning: selection – either left side or right side, "`+`" meaning: one or more.

The attribute `(ATTLIST)` list defines two attributes for the **`person`** element: *`project`* and *`job`*.

The *`project`* attribute can contain character data `(CDATA)` and is optional `(#IMPLIED)`. The *`job`* attribute can only have one of the two values, either "`programmer`" or "`manager`".

There is another XML document structure standard, called XML-Schema [18], which is similar to DTD but is encoded in XML. This standard reconstructs all the capabilities of the DTD and extends them with: namespaces, context sensitivity, the possibility to define several root elements in the same schema, integrity constraints, regular expressions, sub-typing, etc. Tools for transforming XML-Schema representations from/to a DTD representation are available. We use the DTD variant in this example only because it is clearer than the too verbose XML-Schema.

One can consult the World Wide Web Consortium website [5, 18] for more information regarding XML, DTD and XML-Schema.

3.2 ModelicaXML example

To introduce the Modelica XML representation, we give a Modelica example and show its corresponding representation as ModelicaXML.

Elements are in **bold**, attributes are in *italic* and entities are using underline throughout this section, except from Modelica keywords.

```
class SecondOrderSystem
  parameter Real a=1;
  Real x(start=0); Real xdot(start=0);
equation
  xdot=der(x); der(xdot)+a*der(x)+x=1;
end SecondOrderSystem;
```

For ease of presentation, a ModelicaXML document is split into several parts, each representing a more nested level. The ellipses from one level are detailed in the next level:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE program SYSTEM
          "ModelicaXML.dtd">
<program within="...">
  <definition
    ident="SecondOrderSystem"
    restriction="class">
    ...
  </definition>
</program>
```

The root element is a Modelica **`program`**. The child elements of **`program`** are a sequence of **`definition`** elements and an optional *`within`* attribute (see Figure 1, sub-section 3.3 for schemata).

```
<definition
      ident="SecondOrderSystem"
      restriction="class">
  <component>...</component>
  ...
  <equation>...</equation>
  ...
</definition>
```

The **`definition`** element can have **`import`**, **`extends`**, `elements`, **`equation`**, or **`algorithm`** as sub-elements. In our case we only have **`component`** (i.e., variable) and **`equation`** sub-elements inside **`definition`** (see Figure 2, sub-section 3.3 for schemata).

```
<component
    ident="a" type="Real"
    variability="parameter"
    visibility="public">
  <modification_equals>
    <real_literal value="1"/>
  </modification_equals>
</component>
...
<component
    ident="x"
    type="Real"
    visibility="public">
  <modification_arguments>
   <element_modification>
    <component_reference ident="start"/>
      <modification_equals>
        <real_literal value="0"/>
      </modification_equals>
   </element_modification>
  </modification_arguments>
 </component>
```

The first **`component`** (i.e., variable, see Figure 3, sub-section 3.3 for schemata) has the *`variability`* attribute set to `"parameter"` as in `"parameter Real a=1;"`. The second **`component`** declaration (i.e., variable) in the example represents the "`Real x(start=0);`" line from our Modelica class. All components have the *`visibility`* attribute set to "`public`". The last **`component`** is similar to the second **`component`** and is not presented.

```
<equation>
 <equ_equal>
  <component_reference ident="xdot"/>
  <call>
     <component_reference ident="der"/>
      <function_arguments>
        <component_reference ident="x"/>
      </function_arguments>
  </call>
 </equ_equal>
</equation>
```

Equations are enclosed in the **equation** element (see Figure 4, sub-section 3.3 for schemata)

The equation section of the SecondOrderSystem model describes two equations. The first equation is quite straightforward. Equality is represented by an **equ_equal** element with two elements inside. The right-hand side is a function call (using the **call** element) to a derivative and the left hand side is a component reference represented with the element with the same name. The second equation below is more complex. It has function calls represented using the **call** element, binary operations (see Figure 6, sub-section 3.3 for schemata) such as **add**, **mul** for addition (+) and multiplication (*). The **component_reference** elements denote variable references. For the function calls, the arguments are specified using the element **function_arguments** that can contain expressions, named arguments or for indices.

```
<equation>
 <eq_equal>
  <add>
   <call>
    <component_reference ident="der"/>
    <function_arguments>
     <component_reference
             ident="xdot" />
    </function_arguments>
   </call>
   <add>
    <component_reference ident="x"/>
    <mul>
     <component_reference ident="a"/>
     <call>
      <component_reference
           ident="der"/>
      <function_arguments>
         <component_reference
             ident="x" />
      </function_arguments>
     </call>
    </mul>
   </add>
  </add>
  <integer_literal value="1"/>
 </equ_equal>
</equation>
```

ModelicaXML Schemata are explained in the next sub-section.

3.3 ModelicaXML Schema (DTD/XML-Schema)

When designing the ModelicaXML representation we started from the Modelica grammar. We simplified the common cases to compact the XML representation without loss of information or structure. The Modelica DTD/XML-Schema has a rather close correspondence to the Modelica grammar with the following exceptions: attributes are used to make the XML representation more concise and the DTD/XML-Schema jumps over some non-terminals from the Modelica grammar to make the XML representation more compact.

The OpenModelica Project [29] parser for Modelica source code, written in ANTLR [28], was changed to output the ModelicaXML representation. There are many components in the OpenModelica Project that use the ANTLR Modelica parser. Using our ModelicaXML language such tools can be decoupled from this parser. One clear advantage of this approach is that only one parser is maintained and future Modelica language extensions or modifications could be easily integrated.

For presentation purposes we translated our first DTD implementation to XML-Schema using XML Spy [19]. The purpose of this translation was to generate pictures from the XML-Schema. Also, another reason was to have schemata files in both formats for future use. Perhaps, the DTD variant will be discontinued in the future because the XML-Schema is more widely used now.

All elements from our schema have the optional attributes from the location entity (which are *sline*, *scolumn*, *eline* and *ecolumn*) and the *info* attribute, which can be used to store additional information. These location attributes are used to generate a mapping between key elements in our schema and the Modelica source code representation. In the following we present some of the important elements from the DTD/XML-Schema.

The content of our ModelicaXML root element, namely **program** is depicted in Figure 1. Inside the root element we can have none or several **definition** elements. The optional attribute *within* can be used inside a **program** element. The rounded corner boxes on the line connecting two elements can be sequence (like in Figure1) or choice (like in the bottom part of Figure 2).

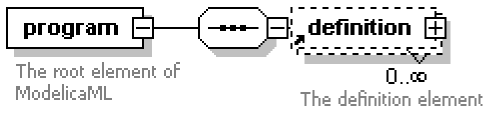

Figure 1: The **program** (root) element of the ModelicaXML Schema

The required attributes for **definition** are *ident* and *restriction* (which can have one of the "class", "model", "record", "block", "connector", "type", "package", or "function" values). Optional attributes are *final*, *partial*, *encapsulated*, *replaceable*, *innerouter*, *visibility* (one of "public", "private" values) and *string_comment*.

The **definition** element is detailed in Figure 2. Presented in the picture at the bottom are the **derived** element (that handles constructs of the type "class X = Y;") and the **enumeration** element used to declare enumeration types. The upper part of Figure 2 shows the other allowed elements that can appear inside the **definition** element. All the elements in the upper part have the *visibility* attribute, taking one of the "public" or "private" values. The *visibility* attribute values are stating the "public" or "private" part from the Modelica source code. We can see that the **definition** element is recursive, which allows the declaration of classes inside classes.

The **definition** element can contain **import**, **extends**, **external**, **equation**, **algorithm**, **annotation** and **component** elements. The latter can use **constrain** element for handling statements like `"type X=Y extends Z;"`.

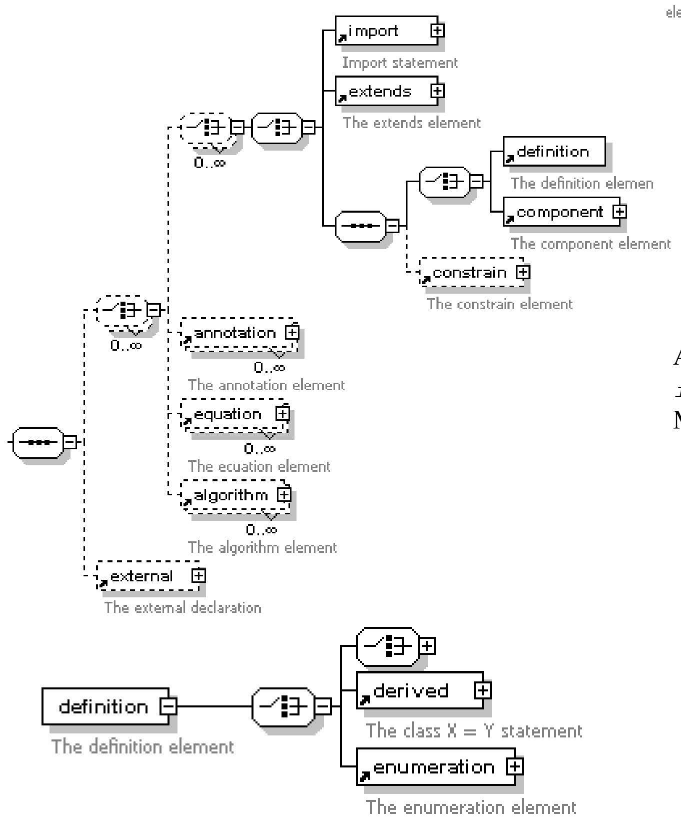

Figure 2: The **definition** element from the ModelicaXML Schema

Component elements, with schemata presented in Figure 3, have attributes representing the Modelica type prefix (*`flow, variability`* *and* *`direction`*), and type name (*`type`*).

The name of the component is stored in the *ident* attribute. These attributes are important because one can query the ModelicaXML representation for a specific component having desired *type* and *ident*. How XML query languages can be used is explained in section 4.

The **type_array_subscripts** element and the **array_subscripts** element are expressing the fact that Modelica array subscripts can be declared either at the type level or at the component level.

One can use the element **modification-_arguments** to further modify the component. Comments for a **component** can be specified with the **comment** element. The elements **modification-_equals** and **modification_assign** are used to modify the component; as sub-elements they can have Modelica expressions.

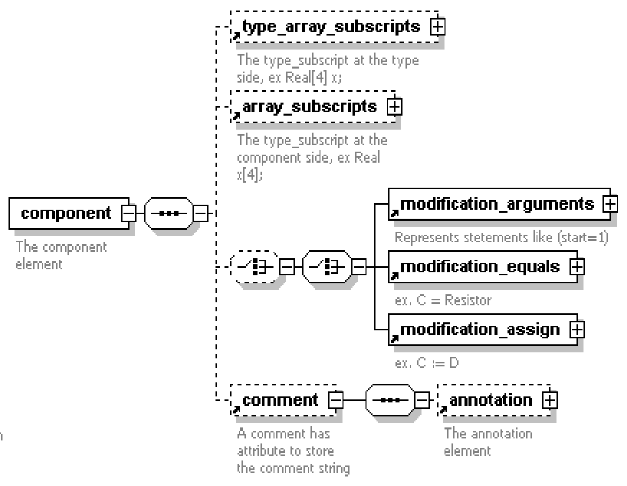
type_array_subscripts
The type_subscript at the type side, ex Real[4] x;
array_subscripts
The type_subscript at the component side, ex Real x[4];
component
The component element
modification_arguments
Represents stetements like (start=1)
modification_equals
ex. C = Resistor
modification_assign
ex. C := D
comment
A comment has attribute to store the comment string
annotation
The annotation element

Figure 3: The **component** element from the ModelicaXML Schema

An **equation** element, presented in Figure 4, can have *`initial`* as an attribute to state if it represents a Modelica initial equation.

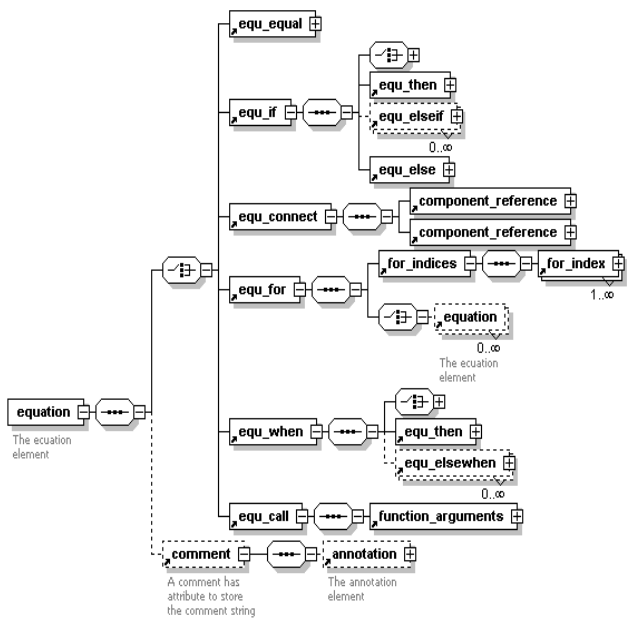

Figure 4: The **equation** element from the ModelicaXML Schema

The content and the structure of the **equation** element are closely following the definition from the Modelica Language Specification [8]. The **equ_connect** element takes component references as arguments here, instead of connect references, as in the version 2.0 of the Modelica Language Specification.

The collapsed parts from the **equ_if** and **equ_when** elements are the Modelica expressions, detailed in Figure 6. The Modelica expressions are present in the collapsed parts of the algorithm elements **alg_if** and **alg_when** and **alg_while**.

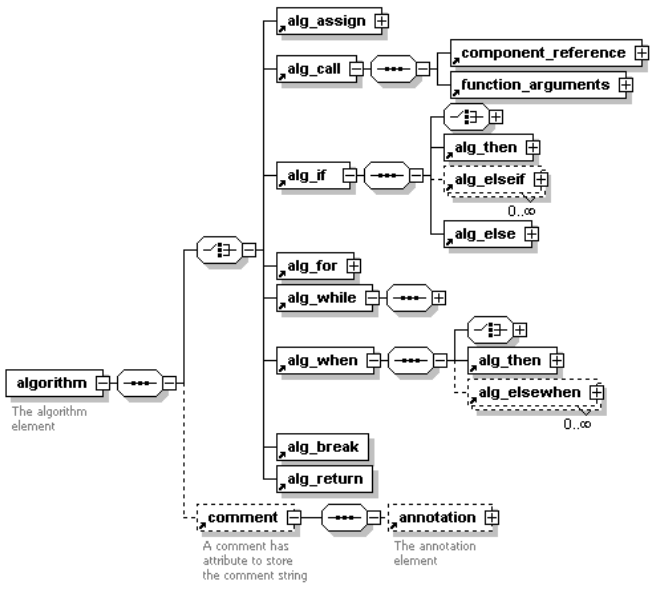

Figure 5: The **`algorithm`** element from the ModelicaXML Schema

The **`algorithm`** element is presented in Figure 5. We point out that the elements **`alg_break`** and **`alg_return`** are recently added statements of the algorithm section in the latest version (2.1) Modelica Language Specification.

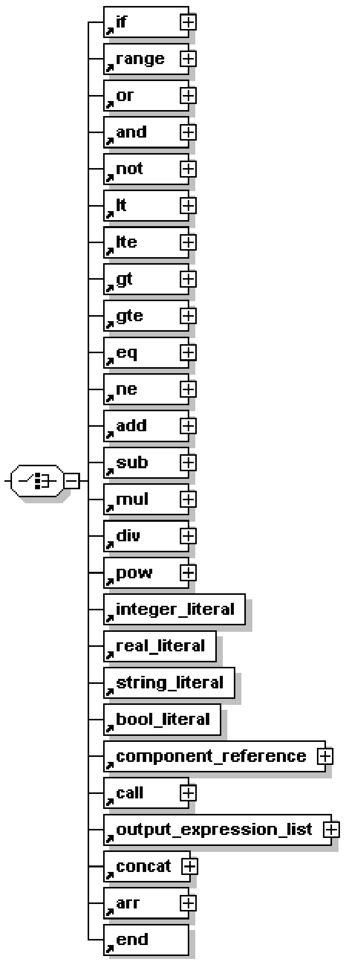

Figure 6: The expressions from ModelicaXML schema

The elements that can appear in ModelicaXML expressions can be found in Figure 6. These are binary operations, literals, component references, array constructions, array operators and logical operations.

The constructs from the ModelicaXML schemata not covered here, along with the full "`modelicaXML.xsd`" (the XML-Schema version) and "`modelicaXML.dtd`" (the DTD version), can be found at the OpenModelica Project website.

4 ModelicaXML and XML tools

This section introduces various XML tools and explains their usage in conjunction with ModelicaXML. In the following, in different sub-sections we cover: the stylesheet language for transformation (XSLT) [6], the query language for XML documents (XQuery) [17] and the Document Object Model (DOM) [3].

4.1 The Stylesheet Language for Transformation (XSLT)

XSL is a stylesheet language for XML. XSLT is the part of XSL that deals with transformation of XML documents.

Using XSLT one can implement pretty printers (un-parsers) that can transform ModelicaXML back into Modelica source code. Alternative transformations could transform ModelicaXML into other general, modeling or markup languages (HTML, XHTML, etc). Transformers that translate other modeling languages (provided that they have an XML representation) into ModelicaXML can also be implemented with XSLT. Using XSLT and ModelicaXML, implementation of HTML documentation generators, similar with what the commercial software Dymola provides, becomes trivial. We cannot provide the HTML documentation generator here because of space reasons, but it will be included in the OpenModelica Project.

We illustrate the usage of XSLT with an example that transforms Modelica code. For this example we assume that Modelica code was already translated to ModelicaXML. After the transformation, one can output the Modelica code from the changed ModelicaXML representation using our "`modelica-xml2modelica.xslt`" stylesheet from the OpenModelica Project.

Example of changing a component name, both in the declaration of the component and in the component references:

```
<xsl:stylesheet version="1.0 …>
<!-- example of component rename -->
<xsl:param name="comp_old_name"/>
<xsl:param name="comp_new_name"/>
<!-- we echo everything that is not a
component or a component reference -->
<xsl:template match="*|@*|text()">
   <xsl:copy>
```

```
        <xsl:apply-templates
              select="*|@*|text()"/>
    </xsl:copy>
  </xsl:template>
  <!-- we match the old component and we
  output the new name -->
  <xsl:template match="component
          [@ident=$comp_old_name]">
    <component ident="{$comp_new_name}">
       <xsl:apply-templates/>
    </component>
  <!-- we match the old component
  reference and we output the new
  component name -->
  </xsl:template>
  <xsl:template match="component_reference
          [@ident=$comp_old_name]">
    <component_reference
            ident="{$comp_new_name}">
         <xsl:apply-templates/>
    </component_reference>
  </xsl:template>
  </xsl:stylesheet>
```

The XSLT engine is using templates that match on the XML tree structure. The matching is performed by the XPath expression appearing as the value of the `match` attribute. By using **`xsl:apply-templates`** element we instruct the XSLT engine to apply the rest of the templates on the sub-tree that we already matched. When this stylesheet is applied on our `SecondOrderSystem` example from section 3.2 with the parameters "`xdot`" and "`xdot_new`" it will change the component name and all the component references of xdot to xdot_new.

XSLT can distinguish between components with the same name defined in different classes by the use of XPath expressions. To rename such occurrences we first match the class in which is defined and then the actual component. This applies for both declarations and component references.

A search-and-replace tool could perform this transformation, but such a tool has no knowledge about the context and it will replace even the occurrences appearing inside comments.

4.2 The Query Language for XML (XQuery)

XQuery is a query language similar with what SQL is for relational databases. Using XQuery, one can easily retrieve information from XML documents. The XQuery and XSLT are overlapping in some features, and our example could be implemented in XSLT also.

We give a short example of a query over our "`SecondOrderSystem.xml`" example from section 3.2. In words, "find all parameter components with type Real and show the initialization value":

```
  <table border="1">
  {
   for $b in
   (document("SecondOrderSystem.xml")/*/
    definition/component)
   where $b/@type = "Real" and
         $b/@variability="parameter"
   return <tr><td>
          { $b/@* }
          { $b/modification_equals }
          </td></tr>
  }
  </table>
```

We executed this query in the Qexo [9] implementation of XQuery and the result in HTML is as follows:

```
  <table border="1">
   <tr><td>
     ident="a" type="Real"
     variability="parameter"
     visibility="public"
     <modification_equals>
       <real_literal value="1" />
     </modification_equals>
    </td></tr>
  </table>
```

As expected, the attributes and the set value of the element corresponding to "`parameter Real a=1;`" from our Modelica example was returned as the answer.

Using XQuery, any types of queries can be asked about the Modelica model. This opens-up the possibility of easily debugging very large models. User interfaces can be implemented to hide the query building from the user. Static type checking can also be implemented as a series of queries on the model, but is not trivial, because the class hierarchy is not explicitly defined in XML.

XQuery uses XPath as sub-language to select the part of tree that matches the XPath expression. In our XML representation one can match an entire component having a specified *ident* attribute. The XPath language can be used to handle scooping.

4.3 Document Object Model (DOM)

The Document Object Model (DOM) [3] is a standard interface that allows programs to access/update the content, structure and style of XML documents. DOM is similar with a general tree-management library.

There are open-source implementations for DOM APIs in Java, C, C++, Perl, Python and other programming languages.

Any Modelica tool written in various programming languages can use the DOM API to directly access/modify the ModelicaXML representation.

5 Towards an Ontology for the Modelica Language

This section investigates the possibility of using the markup languages Resource Description Framework (RDF) [11], RDF Vocabulary Description Language (RDFS) [10] and OWL [16] developed in the Semantic Web Community [13] for development of a Modelica ontology.

An ontology is a description (like a formal specification of a program) of both the objects in a certain domain and the relationships between them. In the context of the Semantic Web there is a layered approach for specifying increasingly richer semantics for the upper layers as in Figure 7.

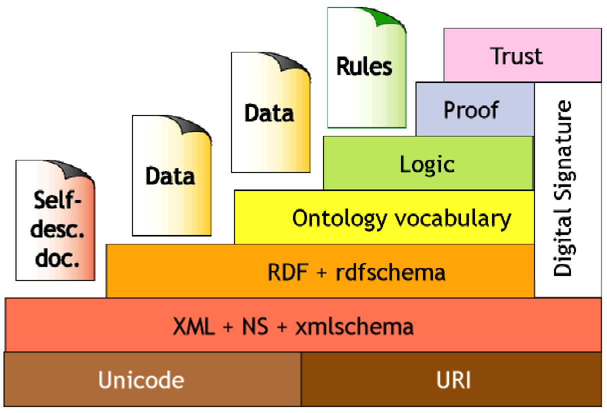

Figure 7: The Semantic Web Layers

At the bottom in top of Unicode and Uniform Resource Identifiers (URI) is XML, namespaces (NS) and XML-Schema. XML specifies a term list with no relations. On top of XML comes RDF to define a vocabulary and some relations. RDFS (RDF schema) defines a vocabulary for constructing RDF vocabularies.

The Ontology layer uses languages like OWL to define description logic relationships.

With ModelicaXML we are now at the XML level! Using RDF we can express graphs and we can model inheritance relationships and place queries over this relation. This can be achieved easily with a smart parser. Using OWL we can place restrictions over relations and concepts and we can reason with inference using Description Logics.

5.1 The Semantic Web Languages

This sub-section briefly introduces the Semantic Web Languages: Resource Description Framework (RDF/RDFS) and Web Ontology Language (OWL).

We illustrate the use of Semantic Web Languages by taking a Modelica model and its representation in OWL.

```
class Body "Generic body"
  Real mass;
  String name;
end Body;
class CelestialBody "Celestial body"
  extends Body;
  constant Real g = 6.672e-11;
  parameter Real radius;
end CelestialBody;

CelestialBody moon(name = "moon",
     mass = 7.382e22, radius = 1.738e6);

Body body_instance(name = "some body",
     mass = 7.382e22);
```

Our Modelica model has two classes (concepts) **Body** and **CelestialBody** the latter being a subclass of the former (by using "extends" statement).

The encoding in OWL is as follows:

```
<?xml version="1.0" ?>
<rdf:RDF

  <!-- namespaces declaration -->
  xmlns=".../inheritance.owl#"
  xmlns:modelica=".../inheritance.owl#"
  xml:base=".../inheritance.owl">
 <owl:Ontology rdf:about=
     ".../inheritance.owl" />

 <!-- define Body -->
 <owl:Class rdf:ID="Body">
   <rdfs:label>Generic Body</rdfs:label>
 </owl:Class>
 <!-- define mass -->
  <owl:DatatypeProperty rdf:ID="mass">
    <rdfs:domain rdf:resource="#Body"/>
    <rdfs:range
      rdf:resource="XMLSchema#float"/>
  </owl:DatatypeProperty>
  <!-- define name -->
  <owl:DatatypeProperty rdf:ID="name">
    <rdfs:domain rdf:resource="#Body"/>
    <rdfs:range
      rdf:resource="XMLSchema#string"/>
  </owl:DatatypeProperty>

  <!-- define CelestialBody -->
  <owl:Class rdf:ID="CelestialBody">
    <rdfs:label>
       Celestial Body
    </rdfs:label>
    <rdfs:subClassOf
       rdf:resource="#Body" />
    <!-- cardinality restriction on the
         g constant: one and only one in
         CelestialBody -->
    <rdfs:subClassOf>
      <owl:Restriction>
       <owl:onProperty
          rdf:resource="#g"/>
       <owl:cardinality rdf:datatype
        ="XMLSchema#nonNegativeInteger">
        1
       </owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <!-- define g -->
  <owl:DatatypeProperty rdf:ID="g">
    <rdfs:domain
      rdf:resource="#CelestialBody"/>
    <rdfs:range
      rdf:resource=" XMLSchema#float"/>
  </owl:DatatypeProperty>
  <!-- define radius -->
  <owl:DatatypeProperty
    rdf:ID="radius">
    <rdfs:domain
       rdf:resource="#CelestialBody"/>
    <rdfs:range
       rdf:resource=" XMLSchema#float"/>
  </owl:DatatypeProperty>
```

```
<!--
 instance declaration of CelestialBody
-->
<CelestialBody rdf:ID="moon">
 <name rdf:datatype="XMLSchema#string">
    moon
 </name>
 <mass rdf:datatype="XMLSchema#float">
    7.382e22
 </mass>
 <radius rdf:datatype="XMLSchema#float">
    1.738e6
 </radius>
 <g rdf:datatype="XMLSchema#float">
    6.672e-11
 </g>
 <g rdf:datatype="XMLSchema#float">
    intentional error
    (string is not float)
 </g>
</CelestialBody>

<!--
 instance declaration of Body
-->
<Body rdf:ID="body_instance">
 <name rdf:datatype="XMLSchema#string">
   some body
 </name>
 <mass rdf:datatype="XMLSchema#float">
   7.382e22
 </mass>
 <--
  intentional error
  (Body does not have a radius)
 -->
 <radius rdf:datatype="XMLSchema#float">
   1.738e6
 </radius>
</Body>
</rdf:RDF>
```

In the OWL representation of the Modelica model we first define **Body** as being an **owl:Class** with "Generic body" as label. The attributes of **Body**, namely: **mass** and **name** are represented as **owl:DatatypeProperty**. The datatype is a binary relation having a **range** (type) and a **domain** (in our case the **Body** concept). As **range** we use the datatypes from XML-Schema, in our case, for **mass** we use "float" and for **name** we use "string".

The class **CelestialBody** is defined as **owl:subclassOf** the **Body** class according to the "extends" statement from our Modelica model. As an OWL feature in the definition of **CelestialBody** we show a local cardinality restriction placed on the **g** relation. This means that in the instances of **CelestialBody**, the **g** component has to appear exactly once. The representation of **g** or **radius** components is similar to the representation of **mass** or **name**.

The **moon** instance of the **CelestialBody** class sets the values of the components. We intentionally added the **g** component twice and with a wrong type. We also declare an instance of the **Body** class that has a **radius** component (which is an error).

To verify the model, our file: "inheritance.owl" was fed into an OWL Validator [32].

The validator, as expected, reports the following errors:

- For the g component that has a string as value: "Range Type Mismatch. Use of this property implies that object is of type XMLSchema#float".
- For the radius component in the body_instance declaration: "Domain Type Mismatch. Use of this property implies that subject is of type #CelestialBody. Subject is declared type [Body]"
- For the moon instance: "Cardinality Violation. Resource #moon violates the cardinality restriction on class #CelestialBody for property #g. Resource has 2 statements with this property. Maximum cardinality is 1".

The OWL language has more constructs than our example has covered. One can consult the OWL website [16] for more details.

5.2 The roadmap to a Modelica representation using Semantic Web Languages

In the example above we have presented a small ontology that models our Modelica model, consisting of both classes and instances. With a clever parser, such ontologies could be generated from Modelica libraries and then used for composing Modelica models.

The roadmap to a Modelica representation in OWL has the following steps:

- Define an RDFS vocabulary for Modelica source code constructs. Such a vocabulary should include concepts like class, model, record, block, etc.
- Transform the Modelica libraries in their OWL representation using the above vocabulary.
- An OWL validator can then check the correctness of both the concepts and the instances of these concepts.

At the end of this roadmap we would have Modelica represented in OWL. The future benefits of such a representation were underlined in the Introduction section. Here, we briefly explain how they could be achieved.

The Autonomous Models

In the OpenModelica Project [24], the Modelica compiler is built from the formal specification (expressed in Natural Semantics [26]) of the Modelica Language. This specification can be compiled to executable form using the Relational Meta-Language (RML) tool [30, 31]. The rules from Natural Semantics could be translated to OWL or RuleML [12] and shipped together with the model. Using the rules from the model a normal browser could compile and simulate the Modelica model. We assume that the platform should have a C compiler.

The Software Information System (SIS)

Having the Modelica ontologies that model the source code one could use the approach detailed in [33] and build the domain model of the problem. Merging them together would result in a Software Information System.

Using such a Software Information System users can ask queries about the Modelica source code concepts (components, classes, etc) that are classified according to the domain model concepts of the problem.

Model consistency could be checked using Description Logic

Modelica models represented in OWL (Description Logics) can be fed into a reasoning tool like FaCT [25] for consistency checking.

Moreover, such support would be of great help to the Modelica library designers that could formally check relevant properties of the class hierarchies.

The checks one can do using Description Logics on the Modelica OWL representation are the following:

- Ensure that the classes and the class hierarchy are consistent (ensure that a class can have instances and is not over-constrained).
- Find the explicit relations between classes, regarding for example sub-typing or equivalence.

Translation of Models to/from Unified Modeling Language (UML)

The UML language has its XML representation called XMI [1]. Translation from Modelica models conforming to a Modelica ontology to XMI could be possible using XSLT.

6 Conclusion and future work

We have presented the ModelicaXML language and some applications of XML technologies. We have shown that there are some missing capabilities with such XML representation and we addressed some of them. We have presented a roadmap to an alternative representation of Modelica in OWL and the use of representation together with the Semantic Web technology.

As future work, we consider completing the ModelicaXML with the definition of all the intermediate steps representations from Modelica to flat Modelica and further to the code generation. This complete representation would allow various open-source tools to act at these formally defined levels, independent of each other. More information could be added in the future to such XML representation, like: model configuration, simulation parameters, etc.

Further insights in the direction of Semantic Web Languages and their use to express Modelica semantics is necessary. Compilation in both directions between OWL and the Relational Meta-Language (RML) is worth considering.

7 Acknowledgements

We would like to thank the anonymous reviewers for their valuable and insightful comments or suggestions.

8 References

1. *CORBA, XML and XMI Resource Page*, http://www.omg.org/xml/.

2. Description Logics Website. *Description Logics*, http://dl.kr.org/.

3. World Wide Web Consortium (W3C). *Document Object Model (DOM)*, http://www.w3.org/DOM/.

4. Dynasim. *Dymola*, http://www.dynasim.se/.

5. Word Wide Web Consortium (W3C). *Extensible Markup Language (XML)*, http://www.w3.org/XML/.

6. Word Wide Web Consortium (W3C). *The Extensible Stylesheet Language Family (XSL/XSLT/XPath/XSL-FO)*, http://www.w3.org/Style/XSL.

7. MathCore. *MathModelica*, http://www.mathcore.se/.

8. *Modelica: A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification version 2.1*, Modelica Association, 2003.

9. *Qexo - The GNU Kawa implementation of XQuery*,http://www.gnu.org/software/qexo

10. World Wide Web Consortium (W3C). *RDF Vocabulary Description Language (RDFS/RDF-Schema)*, http://www.w3.org/TR/rdf-schema/.

11. Word Wide Web Consortium (W3C). *Resource Description Framework (RDF)*, http://www.w3c.org/RDF.

12. *The Rule Markup Initiative*, http://www.dfki.uni-kl.de/ruleml/.

13. *Semantic Web Community Portal*, http://www.semanticweb.org/.

14. World Wide Web Consortium (W3C). *Standard Generalized Markup Language (SGML)*, http://www.w3.org/MarkUp/SGML.

15. UML Website. *Unified Modeling Language (UML)*, http://www.uml.org/.

16. Word Wide Web Consortium (W3C). *Web Ontology Language (OWL)*, http://www.w3.org/TR/2003/CR-owl-features-20030818/.

17. Word Wide Web Consortium (W3C). *XML Query (XQuery)*, http://www.w3.org/XML/Query.

18. Word Wide Web Consortium (W3C). *XML Schema (XSchema)*, http://www.w3.org/XML/Schema.

19. Altova. *XmlSpy*, http://www.xmlspy.com/.

20. Greg Badros. *JavaML: A Markup Language for Java Source Code*, in *Proceedings of The 9th International World Wide Web Conference*,May 15-19, 2000, Amsterdam, Nederlands.

21. Johansson Björn, Jonas Larsson, Magnus Sethson and Petter Krus. *An XML-Based Model Representation for model management, transformation and exchange*, in *ASME International Mechanical Engineering Congress*,November 17-20, 2002, New Orleans, USA.

22. Wolfgang Freiseisen, Robert Keber, Wihelm Medetz, Petru Pau and Dietmar Stelzmueller. *Using Modelica for testing embedded systems*, in *Proceedings of The 2th International Modelica Conference*,March 18-19, 2002, Munich, Germany.

23. Peter Fritzson. *Principles of Object-Oriented Modeling and Simulation with Modelica*, Wiley-IEEE Press, 2003.

24. Peter Fritzson, Peter Aronsson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson and Andreas Karstöm. *The Open Source Modelica Project*, in *Proceedings of The 2th International Modelica Conference*,March 18-19, 2002, Munich, Germany.

25. Ian Horrocks. *The FaCT System*, http://www.cs.man.ac.uk/~horrocks/FaCT/.

26. Gilles Kahn. *Natural Semantics*, in *Programming of Future Generation Computers*, Fuchi K. and Niva M., Editors, 1988, Elsevier Science Publishers: North Holland. p. 237-258.

27. Jonas Larsson, Björn Johansson, Petter Krus and Magnus Sethson. *Modelith: A Framework Enabling Tool-Independent Modeling and Simulation*, in *European Simulation Symposium*,October 23-26, 2002, Dresten, Germany.

28. Terence Parr. *ANTLR Practical Computer Language Recognition and Translation*, http://www.antlr.org/book/.

29. Peter Aronsson Peter Fritzson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, Andreas Karstöm. *The Open Source Modelica Project*, in *Proceedings of The 2th International Modelica Conference*,March 18-19, 2002, Munich, Germany.

30. Mikael Pettersson. *Compiling Natural Semantics*, Lecture Notes in Computer Science (LNCS) 1549, Springer-Verlag, 1999.

31. Mikael Pettersson. *Compiling Natural Semantics*, *Department of Computer and Information Science*, Linköping University, Linköping, Dissertation No. 413, 1995.

32. Dave Rager. *OWL Validator*. 2003, http://owl.bbn.com/validator/#www.

33. Christopher Welty. *An Integrated Representation for Software Development and Discovery*, 1996.