Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 114 (2005) 137–152
www.elsevier.com/locate/entcs

Composition of XML Dialects: A ModelicaXML Case Study

Adrian Pop Ilie Savga Uwe Aßmann Peter Fritzson

Programming Environments Laboratory (PELAB)
Department of Computer and Information Science (IDA)
Linköping University, Linköping, Sweden

Abstract

This paper investigates how software composition and transformation can be applied to domain specific languages used today in modeling and simulation of physical systems. More specifically, we address the composition and transformation of the Modelica language. The composition targets the ModelicaXML dialect which is the XML representation of the Modelica language. By extending the COMPOST concrete composition layer with a component model for Modelica, we provide composition and transformation of Modelica. The design of our COMPOST extension is presented togheter with examples of composition programs for Modelica.

Keywords: Composition of XML dialects, XML, Domain Specific Languages, Modelica, ModelicaXML, COMPOST

1 Introduction

Modelica [10] [4] [5] is an object-oriented modeling language used for modeling of multi-domain (i.e. mechanical, electrical, electronic, hydraulic, etc) complex physical systems. Modeling with Modelica has a component-oriented approach where components can be connected togheter to form a complex system. To have access to the structure of a model, ModelicaXML [17] has been developed as an XML representation (serialization) of Modelica language.

Commercial software products as MathModelica [8] and Dymola [7] as well as open-source as OpenModelica [11] can be used for modeling with the

[1] {adrpo,ilisa,uweas, petfr}@ida.liu.se

doi:10.1016/j.entcs.2004.02.071

Modelica language. While all these tools have high capabilities for compilation and simulation of Modelica models, they:

- Provide little support for configuration and generation of components and models from external data sources (databases, XML, etc).
- Provide little support for security, i.e. protection of "intellectual property" through obfuscation of components and models.
- Do not provide automatic composition of models using a composition language. This would be very useful for automatic generation of models from various CAD products.
- Provide little support for library designers (no automatic renaming of components in models, no support for comparison of two version of the same component at the structure level, etc)

We address these issues by extending the COMPOST framework with a Modelica component model that acts on the ModelicaXML representation.

The use of XML technology for software engineering purposes is highly present in the literature today. The SmartTools system [6] uses XML technologies to automatically generate programming environments specially tailored to a specific XML dialect that represents the abstract syntax of some desired language. The use of Abstract Syntax Trees represented as XML for aspect-oriented programming and component weaving is presented in [18]. The OpenModelica [11] project investigates some transformations on Modelica code like meta-programming [1]. The bases of uniform composition for XML, XHTML dialect and the Java language were developed in the European project Easycomp [9]. However, the possibilities of this framework can be further extended and tested by supporting composition for an advanced domain specific language like Modelica.

The paper is structured as follows. The next section introduces Modelica, ModelicaXML, and COMPOST. Section 3 presents our COMPOST extension and its usage through various examples of composition and transformation programs for Modelica. Conclusion and future work can be found in Section 4. Section 5, the appendix, gives the ModelicaXML representation for some of the examples.

2 Background

In this section we briefly introduce the Modelica language and its XML representation: ModelicaXML, followed by a short description of the COMPOST framework.

2.1 Modelica and ModelicaXML

Modelica has a structure similar to the Java language, but with equation and algorithm sections for specifying behavior instead of methods. Also, in contrast to Java, where one would use assignment statements, Modelica is primary an equation-based language. Equations are more powerful than assignments because they do not specify a certain control and data flow direction. Since the flow direction is not explicitly specified, the Modelica classes are more reusable than the classes from traditional programming languages, which use assignment statements for which the data flow direction is always from the right to the left-hand side.

We introduce Modelica by an example:

```
class HelloWorld "HelloWorld comment"
   Real x(start = 1);
   parameter Real a = 1;
 equation
   der(x) = -a*x;
end HelloWorld;
```

In the example we have defined a class called `HelloWorld`, that has two components and one equation. The first component declaration (line 02) creates a component `x`, with type `Real`. All Modelica variables have a `start` attribute, which can be initialized using a modification equation like `(start = 1)`.

The second declaration declares a so called `parameter` named `a`, of type `Real` and set equal to an integer with value `1`. The parameters are constant during simulation; they can be changed only during the set-up phase, before the actual simulation.

The software composition is not performed directly on the Modelica code, but instead, on an alternative representation of it: ModelicaXML [17].

As an example, the `HelloWorld` class translated to ModelicaXML would have the following representation:

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!DOCTYPE modelica SYSTEM "modelica.dtd">
<program>
  <definition ident="HelloWorld" restriction="class"
              string_comment="HelloWorld comment">
    <component visibility="public" type="Real"
               ident="x">
      <modification_arguments>
       <element_modification>
         <component_reference ident="start"/>
         <modification_equals>
           <integer_literal value="1"/>
         </modification_equals>
       </element_modification>
```

```
      </modification_arguments>
    </component>
    <component visibility="public"
               variability="parameter"
               type="Real" ident="a">
      <modification_equals>
        <integer_literal value="1"/>
      </modification_equals>
    </component>
    <equation>
      <equ_equal>
        <call>
         <component_reference ident="der"/>
         <function_arguments>
             <component_reference ident="x"/>
         </function_arguments>
        </call>
        <sub operation="unary">
         <mul>
          <component_reference ident="a"/>
          <component_reference ident="x"/>
         </mul>
        </sub>
      </equ_equal>
    </equation>
  </definition>
</program>
```

The translation of the Modelica into ModelicaXML is straightforward. The abstract syntax tree (AST) of the Modelica code is serialized as XML using the ModelicaXML format. [2] ModelicaXML is validated using `modelica.dtd` Document Type Definition (DTD). Using the XML representation for Modelica, generation of documentation, translation to/from other modeling languages can be simplified.

2.2 Compost

COMPOST is a composition framework for components such as code or document fragments, with special regard to construction time. Its interface layer called UNICOMP for universal composition provides a generic model for fragment components in different languages and different concrete component models. [3]

Components are composed by COMPOST as follows. First, the compo-

[2] This paper does not present how the Modelica is transformed to ModelicaXML. The reader is referred to [17] for more information.

[3] COMPOST and its interface layer UNICOMP can also model runtime and other types of component models, which are not the subject of this paper.

nents, i.e., templates containing declared and implicit hooks, are read from file. Then, a *composition program* in Java applies composition operations to the templates, and transforms them towards their final form. (The transformations rely on standard program transformation techniques.) After all hooks have been filled, the components can be pretty-printed to textual form in a file again. They should no longer contain declared hooks so that they can be compiled to binary form.

2.2.1 The notions of components and composition

Fragment-based composition with COMPOST is based on the observation that the features of a component can be classified in several dimensions. These dimensions are the language of the component, the model of the component, and abstract component features. The dimensions depend on each other and can be ordered into a layer structure of 5 layers (Fig. 1):

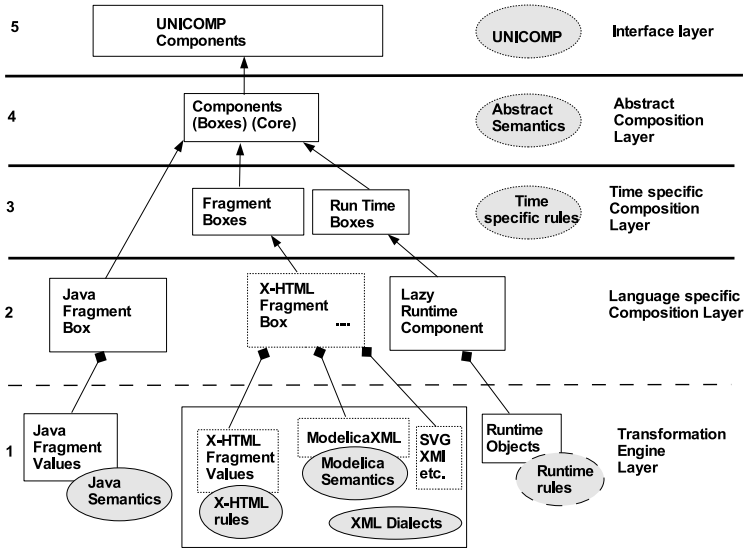

Fig. 1. The layers of COMPOST.

1 **Transformation Engine Layer** The most basic layer encapsulates knowledge about the contents of the components, i.e., about the concrete language of the component. Fragment-based component composition need a transformation engine that transforms the representation of components [2]. For such transformation engines, COMPOST reuses external tools, such as the Java refactoring engine RECODER [14]. This transformation engine layer contains adapters between COMPOST and the external

tools.

2 **Concrete Composition Layer** On top of the pure fragment layer, this layer adds information for a concrete component model, e.g., Java fragment components, or ModelicaXML fragment components. Concrete composition constraints are incorporated that describe valid compositions, which can refer to the contents of the components. For instance, a constraint could be defined that disallows to encapsulating a Java method component into another Java method component.

3 **Time Specific Composition Layer** On this layer the time of the composition is taken into account: static or runtime composition.

4 **Abstract Composition Layer** In this layer, knowledge is modeled that does not depend on the concrete component language, nor on the concrete component model. General constraints are modeled, for instance, that each component has a list of subcomponents, the component hierarchy is a tree, or composition expressions employ the same type of component, independently of the concrete type.

5 **UNICOMP Interface Layer** The interfaces of the abstract composition layer have been collected into a separate interface layer, UNICOMP. This set of interfaces provides a generic fragment component model, from which different concrete component models can be instantiated.

For COMPOST applications, UNICOMP hides underlying concrete information about the component model to a large extent. An application uses COMPOST in a similar way as a component framework with an Abstract Factory [12]. When a component is created, its concrete type is given to the COMPOST factory. However, after creation, the application only uses the UNICOMP generic interfaces. Hence, generic applications can be developed that work for different component models, but use generic composition operations. Already on the Abstract Composition Level, the following uniform operations for fragment components are available:

(i) *Other uniform basic operations.* COMPOST composition operators can address hooks and adapt them during composition for a context. As a basic set of abstract composition operators, *copy*, *extend*, and *rename* are available.

(ii) *Uniform parameterizations.* Template processing works for completely different types of component models. After a semantics for composition points and bind operations has been defined, generic parameterization programs can be executed for template processing.

(iii) *Uniform extensions.* The extension operator works on all types of components.

(iv) *Uniform inheritance.* On the abstract composition layer COMPOST defines several inheritance operators that can be employed to share components, be it Java, or XML-based components. Inheritance is explained as a copy-and-extend operation, and both copy and extend operations are available in the most abstract layer.

(v) *Uniform connection.* COMPOST allows for uniform connection operations, as well for topologic as well as concrete connections [2].

(vi) *Uniform aspect weaving.* Based on these basic uniform operations, uniform aspect weaving operations [13], can be defined.

The great advantage of the layer structure is that new component models, e.g., for XML languages, can be added easily as we show in this paper. In fact, COMPOST is built for extension: adding a new component model is easy, it consists of adding appropriate classes in the concrete composition levels, subclassing from the abstract composition level as we show in Section 3.

2.2.2 Composition Constraints

Each COMPOST layer contains constraints for composition. These constraints consist of code that validates components and compositions.

(i) *Composite component constraints.* A component must be composite, i.e., the composed system is a hierarchy of subsystems. A component is the result of a composite composition expression or a composition program.

(ii) *Composition typing constraints.* Composition operations must fit to components and their composition points. For instance, a composer may only bind appropriate values to composition points (fragments to fragments, runtime values to runtime values), or use a specific extension semantics.

(iii) *Constraints on the content of components.* For instance, for a Java composition system, this requires that the static semantics of Java is modeled, and that this semantics controls the composition. For an XML dialect, semantic constraints can be modeled, for instance, that all links in a document must be valid, i.e., point to a reasonable target. Our extended framework presented in this paper provides parts of the Modelica semantics in top of the ModelicaXML format.

With these constraints, it should be possible to type-check composition expressions and programs in the UNICOMP framework. Many of these constraints can be specified in a logic language, such as Datalog or OWL, and can be generated to check objects on every layer.

2.2.3 Support for staged composition

COMPOST supports staged composition as follows. Firstly, the UNICOMP layer has been connected to the Component Workbench, the visual component editor of the VCF [15]. Composition programs for fragment component models can be edited from the Component Workbench, and executed via COMPOST.

So far, a case study has been build for a web-based conference reviewing system that requires Java and XHTML composition. This paper shows how to *compose Modelica components* by using its alternative XML representation: ModelicaXML.

Secondly, COMPOST can be used to prepare components such that they fit into component models of stage 2 and 3. For instance, COMPOST connectors can prepare a Java class for use in CORBA context [3]. They can also be used to insert event-emitting code, to prepare a class for Aspect-Oriented Programming.

3 COMPOST extension for Modelica

This section describes the Modelica component model. The architecture of our system is presented. Modelica Box and Hook hierarchies are explained. Finally, various composition programs are given as examples.

3.1 Overview

The architecture of the composition system is given in Fig. 2. A Modelica parser is employed to generate the ModelicaXML representation. ModelicaXML is fed into the COMPOST framework where it can be composed and transformed. The result is transformed back into Modelica code by the use of a ModelicaXML unparser.

In order to compose and transform the Modelica code the XML Fragment Box of COMPOST is augmented with with the Modelica structure and semantics to form the Modelica component model. The component model consists of a Modelica Box (templates) hierarchy and Modelica Hook Hierarchy (parameterization) described below.

3.2 Modelica Box Hierarchy

Besides general classes, Modelica uses so called restricted class constructs to structure information and behavior: models, packages, records, types, functions, connectors and blocks. Restricted classes have most properties in common with general classes, but have some restrictions, e.g. there are no equations in records.

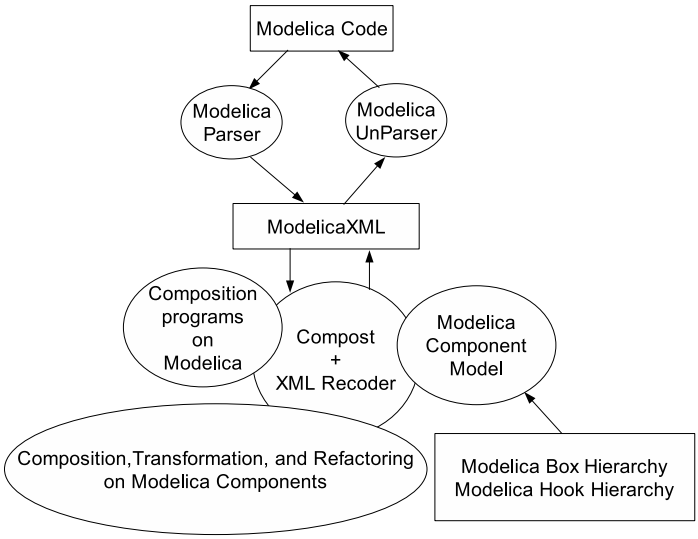

Fig. 2. The XML composition. System Architecture Overview.

Modelica classes are composed of elements of different kinds, e.g.:

- `import` or `extends` declarations.
- public or private variable declarations.
- equation and algorithm sections.

Each of the Modelica restricted classes and each of the element types have their corresponding box class in the Modelica Box hierarchy (Fig. 3).

In our case the boxes (templates) are mapped to their specific element types in the ModelicaXML representation. For example, the `ModelicaClass` box is mapped to a `<define ident="ClassName">..</define>` element. The `ModelicaClass` box can contain several `ModelicaElement` boxes and can contain itself in the case that one Modelica class is declared inside another class.

The boxes that inherit from `ModelicaContainer` represents the usual constructs of the Modelica language. The boxes that inherit from `ModelicaElement` are defining the contents of the boxes that inherits from `ModelicaContainer`.

The boxes incorporate constraints derived from Modelica static semantics. For example, one constraint specifies that inside a `ModelicaRecord` no `ModelicaEquationSection` is allowed.

While these constraints in our case were specified in the Java code, a future extension will automatically generate these constraints from external specifications expressed in formalisms such as DTD, OWL or Relational Meta-Language (RML) [16].

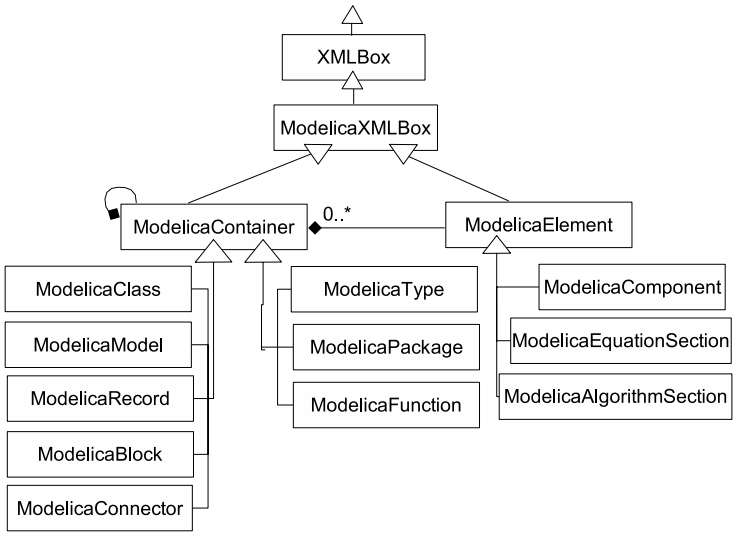

Fig. 3. The Modelica Box Hierarchy. The Box Hierarchy defines a set of templates for each language structure.

3.3 Modelica Hook Hierarchy

Implicit Hooks are fragments of Modelica classes that have specific meaning according to Modelica code structure and semantics. By using Hooks one can easily change/extract parts of the code. In the Modelica Hook Hierarchy presented in (Fig 4) only Implicit Hooks are defined for the Modelica code. There is no need to define Declared Hooks specially for Modelica, because the `XMLDeclaredHook` already performs this operation. One can have an XML declared hook that extracts from the XML document the contents of an element with a specified tag, i.e., `extract`.

Hooks are used to configure parts of boxes. The `XMLImplicitHook` is specialized as `ModelicaParameterHook` or `ModelicaModificationHook`.

`ModelicaParameterHook` binds variable components in ModelicaXML that have `variability` attribute set to `"parameter"`. To provide typing constraints, specific hooks for `real_literal`, `integer_literal`, `string_literal` types have been declared. This constraints the binding of the parameters to values of proper type.

`ModelicaModificationHook` targets component declarations that have their elements changed by modifiers. In the `HelloWorld` example in Section 2, the modifier is imposing on component `x` to change its `start` value. At the ModelicaXML level the `ModelicaModificationHook` is searching for XML elements of the form:

```
<component ident="ComponentName">
```

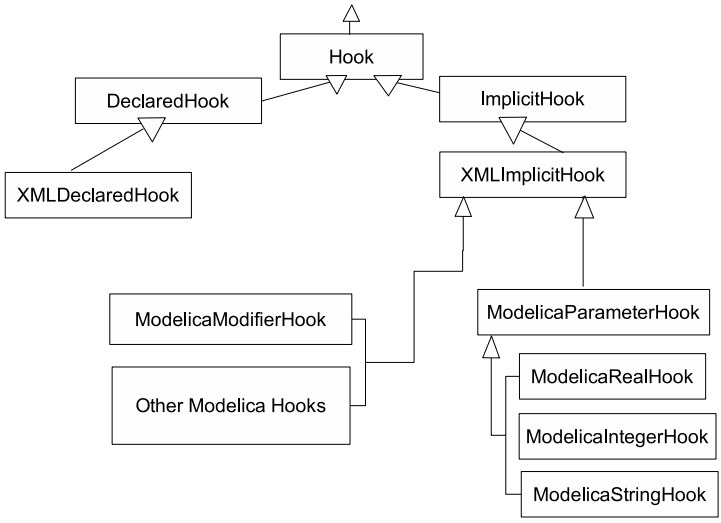

Fig. 4. The Modelica Hook Hierarchy.

```
<modification_arguments>
 <element_modification>
  <component_reference ident="element"/>
  <modification_equals>
    value initialization e.g.
    <integer_literal>1</integer_literal>
  </modification_equals>
 </element_modification>
</modification_arguments>
</component>
```

This hook will bind proper values to the modified elements.

Also, other types of implicit hooks can be specified like hooks for the left hand side or the right hand side of an equation, hooks that change types of components, hooks that change the documentation part of a class declaration, etc.

3.4 Examples of composition and transformation programs

This subsection gives concrete examples on the usages of our framework. The examples are written in Java, but they could easily be performed using a tool that has visual abstractions for the composition operators. For presentation issues only the Modelica code is given in the examples below and their corresponding ModelicaXML representation is presented in Section 5.

3.4.1 Generic parameterization with type checking

To be able to reuse components into different contexts they should be highly configurable. Configuration of parameters in Modelica is specified in class definitions and can be modified in parameter declaration. The values can be read from external sources using external functions implemented in C or Fortran. In the example below we show how the parameters of a Modelica component can be configured using implicit hooks. Because we use Java, the parameter/value list can be read from any data source (XML, SQL, files, etc). The example is based on the following Modelica class:

```
class Engine
   parameter Integer cylinders = 4;
   Cylinder c[cylinders];
   /* additional parameters, variables and equations */
end Engine;
```

Different versions of the `Engine` class can be automatically generated using a composition script. Also, the parameter values are type checked before they are bound to ensure their compatibility. The composition script is given below partially in Java, partially in pseudo-code:

```
ModelicaCompositionSystem cs =
     new ModelicaCompositionSystem();
ModelicaClass templateBox =
     cs.createModelicaClass("Engine.mo.xml");
/* read parameters from configuration file, XML or SQL */
foreach engine entry X {
  ModelicaClass engineX =
     templateBox.cloneBox().rename("Engine_"+X);
   foreach engine parameter {
    engineX.findHook("parameterName").bind(parameterValue);
      /* typed parameterization */
   }
   engineX.print();
}
```

Using a similar program, the modification of parameters can be performed in parameter declarations.

3.4.2 Class Hierarchy Refinement using Declared Hooks

When designing libraries one would like to split specific classes into a more general part and a more specific part. As an example, one could split the class defined below into two classes that inherits from each other, one more generic and one more specific in order to exploit reuse. Also if one wants to add a third class, e.g. `RectangularBody`, to the created hierarchy the transformation above would be beneficial. The specific class that should be modified is given below:

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

The desired result, the two split classes where one inherits from the other, is shown below:

```
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;
```

One can see that this transformation extracts parts of classes and inserts them into a new created class. Also, the old class is modified to inherit from the newly created class.

This transformation is performed with the help of one declared hook (for the extraction part) and an implicit hook for the superclass, with its value bound to the newly created class. The user will guide this operation by specifying, with a declared hook or visually, which parts should be moved in the new class. The composition program that performs this transformations is as follows:

```
ModelicaCompositionSystem
      cs = new ModelicaCompositionSystem();
ModelicaClass bodyBox =
      cs.createClass("Body.mo.xml");
ModelicaClass celestialBodyBox =
      cs.createModelicaClass("Celestial.mo.xml");
ModelicaElement extractedPart =
      celestialBody.findHook("extract").getValue();
celestialBody.findHook("extract").bind(null);
bodyBox.append(extractedPart)
bodyBox.print();
celestialBody.findHook("superclass").bind("Body");
      /* or findSuperclass().bind("Body"); */
celestialBody.print();
```

Similar transformations can be used to compose Modelica models based on the interpretation of other modeling languages. During such composition some classes need to be wrapped to provide a different interface. For example, when there is only a force specified for moving a robot arm, but the available library of components only provides electrical motors that generate a force proportional to a voltage input.

3.4.3 Composition of classes or model flattening

Mixin composition of the entire contents of two or more classes into one another is performed when the models are flattened i.e. as the first operation in model obfuscation or at compilation time. The content of the classes composed below is not relevant for this particular operation. The composition

program that encapsulates this behavior is as follows:

```
ModelicaCompositionSystem cs =
     new ModelicaCompositionSystem();
ModelicaClass resultBox =
     cs.createModelicaClass("Class1.mo.xml");
ModelicaClass firstMixin =
     cs.createModelicaClass("Class2.mo.xml");
ModelicaClass secondBox =
     cs.createModelicaClass("Result.mo.xml");
resultBox.mixin(firstMixin);
resultBox.mixin(secondMixin);
resultBox.print();
```

It first reads the two classes from files, creates a new result class and pastes the contents of the first classes inside the new class.

4 Conclusion and future work

We have shown how composition on Modelica, using its alternative the ModelicaXML representation, can be achieved with a small extension of the COMPOST framework. While this is a good start, we would like to extend our work in the future with some additional features like:

- More composition operators and more transformations, i.e., obfuscation, symbolic transformation of equations, aspect oriented debugging of component behavior by weaving `assert` statements in equations, etc.
- Implementation of full Modelica semantics to guide the composition, based on the already existing Modelica compiler implemented in the OpenModelica project [11].
- Validation of the composed or transformed components with the OpenModelica compiler.
- Automatic composition of Modelica models based on interpretation of other modeling languages.

Modelica should provide additional constraints on composition, based on the domain knowledge. These constraints are specifying, for example, that specific components should not be connected even if their connectors allows it. We would like to further investigate how these constraints could be specified by library developers.

5 Appendix

`CelestialBody` in ModelicaXML format before transformation:

```
<definition ident="CelestialBody" restriction="class"
```

```
            string_comment="Celestial Body"/>
  <component visibility="public" ident="mass" type="Real" />
  <component visibility="public" ident="name" type="String" />
  <component visibility="public" variability="constant"
             ident="g" type="Real">
    <modification_equals>
      <real_literal value="6.672e-11"/>
    </modification_equals>
  </component>
  <component visibility="public" variability="parameter"
             ident="radius" type="Real" />
</definition>
```

`CelestialBody` and `Body` in ModelicaXML format after transformation:

```
<definition ident="Body" restriction="class"
            string_comment="Generic Body"/>
  <component visibility="public" ident="mass" type="Real" />
  <component visibility="public" ident="name" type="String" />
</definition>

<definition ident="CelestialBody" restriction="class"
            string_comment="Celestial Body"/>
  <extends type="Body"/>
  <component visibility="public" variability="constant"
             ident="g" type="Real">
    <modification_equals>
      <real_literal value="6.672e-11"/>
    </modification_equals>
  </component>
  <component visibility="public" variability="parameter"
             ident="radius" type="Real" />
</definition>
```

The `Engine` class representation in ModelicaXML.

```
 <definition ident="Engine" restriction="class">
  <component visibility="public" variability="parameter"
             type="Integer"
             ident="cylinders">
   <modification_equals>
     <integer_literal value="4"/>
   </modification_equals>
  </component>
  <component visibility="public" type="Cylinder" ident="c">
    <array_subscripts>
     <component_reference ident="cylinders"/>
    </array_subscripts>
  </component>
 </definition>
```

References

[1] Peter Aronsson, Peter Fritzson, Levon Saldamli, Peter Bunus, and Kaj Nyström. Meta Programming and Function Overloading in OpenModelica. In *Proceedings of the 3th International Modelica Conference*, 3-4 October 2003.

[2] Uwe Aßmann. *Invasive Software Composition*. Springer-Verlag, February 2003.

[3] Uwe Aßmann, Thomas Genßler, and Holger Bär. Meta-programming Grey-box Connectors. In R. Mitchell, editor, *Proceedings of the International Conference on Object-Oriented Languages and Systems (TOOLS Europe)*. IEEE Press, Piscataway, NJ, June 2000.

[4] The Modelica Association. Modelica. http://www.modelica.org.

[5] The Modelica Association. Modelica - A Unified Object-Oriented Lanugage for Physical System Modeling. http://www.modelica.org/.

[6] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Didier Parigot, and Claude Pasquier. SmartTools: a Generator of Interactive Environments Tools. In *International Conference on Compiler Construction CC'01*, April 2001.

[7] Dynasim Company. Dymola. http://www.dynasim.se.

[8] MathCore Company. Mathmodelica. http://www.mathcore.se.

[9] The EASYCOMP Consortium. EASYCOMP home page. http://www.easycomp.org, August 2000.

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

[11] Peter Fritzson, Peter Bunus, Vadim Engelson, Levon Saldamli, Henrik Johansson, and Andreas Karström. The Open Source Modelica Project. In *Proceedings of the 2nd International Modelica Conference*, 18-19, March 2002.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[13] Mattias Karlsson. Component-Based Aspect Weaving Through Invasive Software Composition. Master's thesis, Department of Computer and Information Science, Linköpings Universitet, September 2003.

[14] Andreas Ludwig. The RECODER Refactoring Engine. http://recoder.sourceforge.net, September 2001.

[15] Johann Oberleitner and Thomas Gschwind. Composing distributed components with the Component Workbench. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware (SEM 2002)*, volume 2596 of *Lecture Notes in Computer Science*. Springer-Verlag.

[16] Mikael Petterson. *Compiling Natural Semantics*. PhD thesis, Linköping University, 1995. Dissertation No. 413, also in Lecture Notes in Computer Science (LNCS) 1549, Springer-Verlag, 1999.

[17] Adrian Pop and Peter Fritzson. ModelicaXML: A Modelica XML Representation with Applications. In *Proceedings of the 3rd International Modelica Conference*, 3-4 October 2003.

[18] Stefan Schonger, Elke Pulvermüller, and Stefan Sarstedt. Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees. In *Second Workshop on Aspect-Oriented Software Development*, February 2002.