Building Adaptable and Reusable XML Applications with Model Transformations

Ivan Kurtev
Software Engineering Group, University of Twente
P.O. Box 217, 7500 AE, Enschede, the Netherlands
kurtev@ewi.utwente.nl

Klaas van den Berg
Software Engineering Group, University of Twente
P.O. Box 217, 7500 AE, Enschede, the Netherlands
k.g.vandenberg@ewi.utwente.nl

ABSTRACT
We present an approach in which the semantics of an XML language is defined by means of a transformation from an XML document model (an XML schema) to an application specific model. The application specific model implements the intended behavior of documents written in the language. A transformation is specified in a model transformation language used in the Model Driven Architecture (MDA) approach for software development. Our approach provides a better separation of three concerns found in XML applications: syntax, syntax processing logic and intended meaning of the syntax. It frees the developer of low-level syntactical details and improves the adaptability and reusability of XML applications. Declarative transformation rules and the explicit application model provide a finer control over the application parts affected by adaptations. Transformation rules and the application model for an XML language may be composed with the corresponding rules and application models defined for other XML languages. In that way we achieve reuse and composition of XML applications.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks*. D.3.4 [**Programming Languages**]: Processors – *Interpreters*. I.7.2 [**Document and Text Processing**]: Document Preparation – *Markup languages*.

General Terms
Documentation, Design, Languages.

Keywords
XML, XML Processing, MDA, Transformation Language, Model Transformations.

1. INTRODUCTION
Extensible Markup Language (XML) is nowadays a dominant data representation format used in many areas in computer science and industry such as World Wide Web (WWW), eCommerce and Web Services Architecture. As a result many XML markup languages emerged focusing on a particular problem domain. This opens the possibility for reuse of existing languages into new ones (known as hybrid languages) and creating compound documents based on more than one vocabulary. This trend is clearly exemplified by the recent standards created within W3C based on composition and reuse of modules defined for the popular Web languages such as XHTML, SMIL, MathML, SVG.

The wide acceptance of XML motivates the need for techniques and tools that support the development of XML-based applications. Today, XML technology offers mature standards and tools that mainly facilitate the definition and processing of the syntactical part of XML applications. These are the XML Schema for definition of markup language syntax, XSLT for defining document transformations, XPath/XQuery for navigation and extraction over documents, and a large set of high quality XML parsers.

Apart from the traditional tasks of syntax definition and parsing, an XML application requires processing that reflects the semantics of the markup used in the documents. Since the semantics is specific to the application it is much more difficult to standardize the application-specific processing phase in contrast with the syntax parsing phase. The application usually has to transform XML documents into application-specific structures that implement the concepts in the domain for which XML is used. That is a recurring task and is a candidate for at least a partial automation. Furthermore, today's applications must satisfy certain quality properties. The first property we consider is the adaptability of the application that allows it to be easily adapted at low cost when the syntax of the markup language changes. The second property is the reusability of the processing application. This is motivated by the need of compound documents based on multiple vocabularies. The ability to reuse the vocabulary is naturally followed by the need to reuse processing logic for that vocabulary. One possible reuse is in the composition of several XML applications in a new one. In that respect reusability is a prerequisite for the composability of the applications.

Generally, the programmer may choose between two technologies to process XML documents with a programming language: generic document interfaces (such as DOM and SAX), and data binding. Simple API for XML (SAX) [24] and Document Object Model (DOM) [26] provide interfaces to documents that reflect the document syntax. It is acknowledged that this approach is too low level and error-prone. Moreover, the application is often designed in an ad-hoc manner and hardly possesses the adaptability and reusability properties. For instance, a change in the syntax may lead to many changes in the code and recompilation of the whole application.

In data binding [23] a document schema is compiled into a set of classes in a given language and the processing of documents is

WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

automated (a process known as unmarshaling). This approach, however, is not applicable if the application classes already exist and differ significantly from the document syntax structure. Reusability and adaptability are deteriorated because every change in the schema requires schema recompilation.

The problems of adaptability and reusability caused by the need for redesign and recompilation of applications are strongly related to the fact that the XML technology does not provide a standard means for specifying semantics of markup languages. In current XML applications the relation between syntax and its intended meaning is not explicit. It is often hard-coded in the application and it is difficult to reuse and maintain it. On the other hand, the domain of programming language specification offers a number of frameworks for defining language semantics in an explicit way [30]. Issues like the evolution and composition of languages and their translators have been on the research agenda for a long time [12][13][3]. Clearly, the experience gained in that area can be used to develop tools and techniques required for XML applications.

In this paper we propose an approach for XML processing based on a declarative specification and execution of a model transformation from the language syntax structures (the source model) to the application structures (the target model). That transformation can be regarded as a semantic specification for the markup language syntax. We assume that the document syntax is either defined in an XML schema or as a set of elements and attributes (schema-less approach). If a schema is present it is treated as a model of XML documents. The application classes form the target model. A given transformation contains declarative rules that encode how the syntax constructs defined in the source schema represent components in the target model. Transformations are specified in a domain-specific model transformation language. We present a language that has been developed for another problem domain in software engineering: the OMG's Model Driven Architecture (MDA) [17] approach and show how the language is applied in the context of XML processing.

By using transformations we achieve a better separation of concerns. XML applications are decomposed in three components: syntax definition (schema), transformation specification and application classes. Application classes do not contain syntax processing code; this is captured in the transformation specification.

The benefits of our approach are the following:

- developers are freed from writing a low level syntax processing code;
- it opens a possibility for automatic generation of language translators similar to the compiler-compiler approach;
- syntax and application code may evolve independently;
- transformation rules can be designed at the granularity that provides good adaptability of the application. Only rules that reflect changes in the syntax are updated;
- reusability of the applications is improved. Using multiple vocabularies in a document is achieved by composing corresponding transformation rules and application classes.

This paper is organized as follows. Section 2 gives a detailed overview of the approach. Section 3 presents an example used further in Section 4 to present the features of our transformation language. Section 5 discusses related work. Section 6 gives conclusions and directions for future work.

2. OVERVIEW OF THE APPROACH

Our approach is based on specification and execution of transformations between models. We borrow this technique from the Model Driven Architecture (MDA) approach for software development. The application of model transformations to XML processing shown here is an elaboration of our previous work reported in [9].

In an MDA-based process the development of a software system starts with making a detailed model of it. That model is a system's specification at an abstract level that does not contain information about the technologies that will be used for the system implementation (so called Platform Independent Model, PIM). When the implementation technologies are chosen the PIM is (semi) automatically transformed to a model that contains implementation information known as Platform Specific Model (PSM). The PSM has to contain enough information to allow automatic code generation of the system. As we can see transformations between models is the main operation in an MDA-based process. Recent activities in the area are focused on development of domain-specific transformation languages and supporting tools [18].

In our approach to XML processing we benefit from the ability to express and execute transformations for specifying, in a declarative and explicit way, the actions that an application takes during the processing of XML documents. We use a transformation language developed in the context of an MDA process [10].

Fig. 1 shows the basic model transformation pattern in MDA.

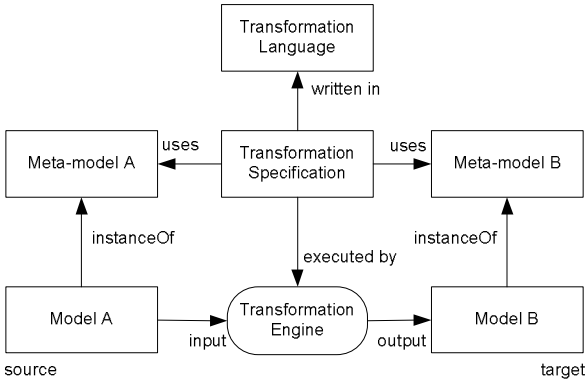

Figure 1. The basic model transformation pattern

In this pattern a transformation is executed by the transformation engine taking model A as a source and producing model B as a target. A transformation specification is written in a transformation language. In MDA, models are conforming to (or are instances of) meta-models that define the rules of the modeling languages used to create the models. An example of a modeling language is Unified Modeling Language (UML) [19]. A transformation specification is based on the knowledge of the meta-models (in Fig. 1 meta-model A and meta-model B). A

transformation can be executed on every input model that conforms to the meta-model A.

To apply this approach to XML processing we adapt the pattern in that context (see Fig. 2). The transformation engine takes an XML document as input and generates an output. This output can be a set of rows in a relational database, another XML document or objects instances of classes written in a given programming language (e.g. Java). In this paper we focus on applications that instantiate objects on the base of XML documents. These objects may be implemented in any programming language. Our transformation language is independent of concrete languages used to specify the models. We chose Java to illustrate our approach. To apply the approach we must identify the meta-models that will be used to specify the transformation. Fig. 2 shows the transformational pattern applied in the context of XML processing.

First we have to identify the model of XML documents. Available alternatives are the Document Object Model (DOM) and the XML Information Set. Both reflect the XML grammar that is used to check if an XML document is well-formed. In this paper we choose DOM as a more popular standard among the developers but any other model that reflects the notion of well-formedness may be used. Furthermore, XML documents may conform to a schema. Most of the today's XML languages are defined by an XML schema. The schema can be perceived as a model of the class of documents that are valid against that schema.

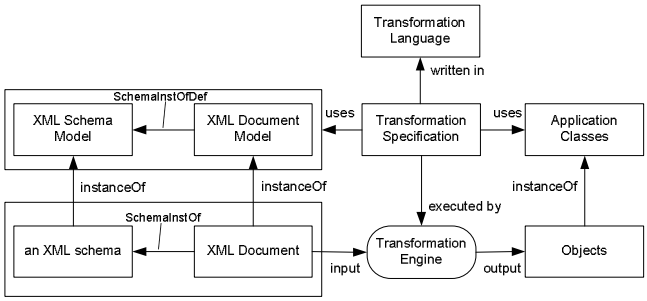

Figure 2. Transformation pattern for XML processing

We assume that the schema may be available and the processing may use the schema constructs. Moreover, the presence of schema does not inhibit the conformance to the generic XML document model. It only imposes additional constraints. Therefore the documents may be considered as instances of two different models: the generic document model and the document schema. The *instanceOf* relationships are defined in different ways in these cases and may exist together. Working with both models is important and should be available in the transformations. A software engineer should be able to specify both generic document processing reflecting DOM and document processing that uses type information based on schema types. To employ schemas in our approach we include a model of XML Schema that can be derived from the specification. In that way the source meta-model (meta-model A in Fig. 1) is split into two separate models in the case of XML processing: the XML Schema Model and the XML Document Model. The transformation specification may use both the schema and the generic document types. The XML Document Model is defined in UML and shown in Fig. 3. The XML Schema Model is referred to [27].

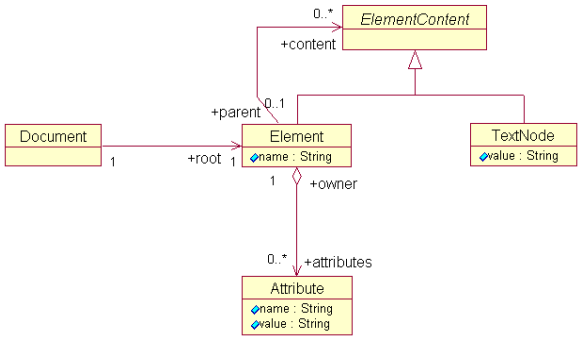

Figure 3. The XML document model

A concrete schema and a concrete XML document are used as a source of the transformation but only the document is transformed. The schema is used only for purposes of selection of concrete document nodes on the base of their types and associated element or attribute declarations. This is done by using a Post Schema Validation Infoset (PSVI).

The target meta-model in the pattern in Fig. 2 consists of the application-specific classes. The output model (corresponding to model B in Fig.1) is therefore a set of objects instances of the application classes.

The structure of an XML application based on model transformations is shown in Fig. 4. The static part of the application consists of the components surrounded by the gray area. In this part the optional XML schema, the transformation specification and the classes are the application specific components. Application classes implement the intended meaning of the markup syntax constructs and the transformation specification specifies how the syntax is related to that meaning. Application classes do not contain syntax processing functionality. This functionality is captured in the transformation specification.

The dynamic part of the application contains the components surrounded by the white rectangle in the lower right corner of Fig.4.

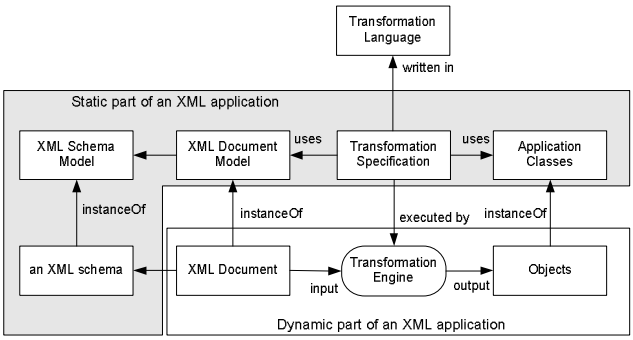

Figure 4. Static and runtime part of an XML application

The objects are part of the dynamic state of the XML application and are instantiated at runtime after the execution of the transformation.

3. RUNNING EXAMPLE

We will illustrate our approach on the base of an example presented in this section. Then in Section 4, we show the transformation specification for the example and explains the transformation language constructs.

The example uses a simplified version of the SMIL timing synchronization module [28] intended to be used together with other markup languages such as XHTML. A set of application classes written in Java is used to implement the behavior of the time dependency graph nodes according to the time model of SMIL. The structure of the example is shown in Fig. 5. It is an instance of the general pattern for XML processing in Fig. 2.

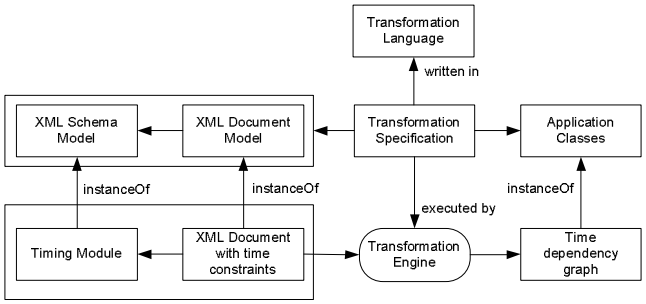

Figure 5. Structure of the example application

It should be noted that the SMIL timing model is rather complex and includes many capabilities. It is beyond the scope of the paper to provide a detailed description of that model and the way to design a time scheduler program that executes the model. We limit ourselves to a very simplified subset of the timing module that uses a set of attributes indicating the type of the time element (interval, parallel or sequence) and the start, end and duration properties. Our source schema therefore contains four attributes taken from the SMIL specification. The following schema snippet shows the attribute definitions.

```
<attribute name='begin' type='string'/>
<attribute name='end' type='string'/>
<attribute name='dur' type='string'/>
<attribute name='timeContainer' type='string'/>
```

The attribute *timeContainer* may assume 3 values: *none*, *par* and *seq*. The first value *none* indicates that an element that has an attribute with value *none* is an atomic timed element (interval). The two values *par* and *seq* determine the element as a time container with a parallel and sequential scheduling of its children.

The processing of an XML document that uses these attributes results in creation of a time dependency graph that captures the timing constraints and dependencies expressed in the document. The nodes of that graph reflect the semantics of interval and container nodes and implement their functionality. Time graph nodes are instances of application classes written in Java. A sketch of the classes is given below. We focus on building the time dependency graph, and do not include an implementation of the timing functionality.

```
public interface ControlledObject {
  public void activate();
  public void deactivate();
}
```

```
public abstract class TimedElement{
  public int begin;
  public int end;
  public int dur;
  public ControlledObject ctrlObject;
  public void abstract start();
  public void abstract stop();
}
public class Interval extends TimedElement{
  public void start()  {//concrete implementation}
  public void stop()  {//concrete implementation}
}
public abstract class TimeContainer extends TimedElement{
  public Vector components;
}
public class Parallel extends TimeContainer{
  public void start()  {//concrete implementation}
  public void stop()  {//concrete implementation}
}
public class Sequence extends TimeContainer{
  public void start()  {//concrete implementation}
  public void stop()  {//concrete implementation}
}
```

Class *TimedElement* is the abstract root class of the application hierarchy. Every node in the time graph is an indirect instance of that class. It has fields for the begin, end and the duration of the timed element. A node in the graph manipulates the behavior of an object. That object could be a text, picture, an audio clip or any other element. Timed objects must implement the interface *ControlledObject*. At the time of activation/deactivation of a node it invokes the operations *activate()/deactivate()* on the controlled object.

We have three types of time nodes: *Interval*, *Parallel* and *Sequence*. The latter two are time operators that specialize the abstract class *TimeContainer*. Time containers have other nodes as children and impose a sequential or parallel order on their activation. Time containers may be nested. The execution semantics of the time graph is described in the SMIL specification [28].

4. TRANSFORMATION LANGUAGE

The transformation language presented here is developed according to the requirements formulated in the Query/Views/Transformations Request for Proposals by OMG [18]. According to this OMG document transformations describe relationships between a source meta-model and a target meta-model in a declarative way. Another requirement is for mechanisms that support reusability and extensibility of transformation definitions. We shall see in Section 4.5 that this requirement is essential in achieving the adaptability and reusability of XML applications.

4.1 Transformation Specification

The transformation language is used for the specification of transformations. The following code is the transformation specification for processing XML documents that use the timing attributes in the example to impose timing constraints upon other objects. After the execution of this transformation an XML application will build a time graph that captures the timing constraints specified in the document. It does not process the

concrete controlled objects which are described by another markup language. An example of such a markup language that uses timing constraints is given in section 4.5.

The transformation is explained throughout this section in combination with an explanation of the transformation language constructs. Language keywords are given in bold. A more detailed description of the language can be found in [10].

```
1. timedElementMapping abstract ModelElementRule {
2.   source[e:Element link-to(node),
3.                    condition{XPath($e[@timeContainer])} ]
4.   target [node: TimedElement{begin, end, dur, ctrlObject}]
5.   SlotRules{
6.      beginValue
7.         source[beg:Attribute=XPath($e/@begin)]
8.         target [begin=toInt(beg.value)]
9.      endValue
10.        source[end:Attribute=XPath($e/@end)]
11.        target [end=toInt(end.value)]
12.     durValue
13.        source[duration:Attribute=XPath($e/@dur)]
14.        target [dur=toInt(duration.value)]
15.  }
16. }
17. parallelContainer ModelElementRule inherits
18.                          timedElementMapping{
19.   source[ condition{XPath($e[@timeContainer='par'])} ]
20.   target[ node: Parallel{components} ]
21.   SlotRules{
22.      componentsValue
23.         source[timedChild:Element=XPath($e/*[@timeContainer])]
24.         target[components=target(timedChild, node)]
25.   }
26. }
27. intervalNode ModelElementRule inherits
28.                                timedElementMapping{
29.    source[ condition{XPath($e[@timeContainer='none'])}]
30.    target[ node: Interval]
31. }
```

4.2 Language Overview

A transformation specification is written in the transformation language being described here and is based on the meta-models of the source model and the target model. In the case of XML processing the source and target models and their meta-models are shown in Fig. 2.

A transformation specification is a set of rules. There are two types of rules: model element rules and slot rules. Model element rules select elements in the source model and execute actions. Actions are creation of elements in the target model, update of existing elements and deletion of elements. In our approach the model elements in the source model are XML document nodes and the model elements in the target model are Java objects. Slot rules are used to relate the elements by setting their slot values. For Java objects slots are defined by the class fields. For XML nodes slots are defined by the attributes and association roles in the XML Document Model in Fig. 3. Both types of rules have rule source that selects elements in the source model.

Our example transformation consists of 3 model element rules which in turn have associated slot rules.

4.3 Language Constructs

4.3.1 Model Element Rules

Model element rules create new elements in the target model or modify existing ones in the source and the target models. The creation of new elements is done by instantiating the types in the target meta-model, in our example these are the Java classes.

The syntax of model element rules is specified below in a pseudo EBNF notation. Non-terminals are in italic.

ruleName **ModelElementRule** *InputParameters*? {
  RuleSource
  target [*Action* +]
  *SlotRule**
}

Every model element rule has a name, a source, a target, an optional list of input parameters and is associated with a number of slot rules. Model element rules specify a correspondence between elements enumerated in the rule source and elements in the rule target. When a rule is executed elements in the rule target are instantiated for every tuple that matches the rule source. Model element rules may be defined as abstract. If a rule is abstract it cannot be executed directly. It can be inherited by other rules and provides its components for reuse.

Rule source specifies the characteristics of the elements in the source model that will be selected by a transformation rule. Rule source is an expression that is evaluated to a set of tuples containing elements in the source model.

A rule source enumerates at least one component. An optional condition may be imposed on the components. The components of a rule source are two kinds: a model element identifier that uniquely identifies an element in the source model and variable that can be bound to more than one source element. Each variable has a type. The type is a model element from the source meta-model. The variable matches the instances of this type in the source model. Variables can be initialized by an expression written in Object Constraints Language (OCL). For the purposes of XML processing we allow the usage of XPath expressions. The condition of a rule source is a Boolean expression. The result of the evaluation of a rule source is a set of tuples formed by the Cartesian product of the matches for each component. Tuples that do not satisfy the condition of the rule source are excluded.

Consider the first model element rule named *timedElementMapping* (lines 1-16). The source contains one component, which is a variable of type *Element* with an imposed condition. The evaluation of the rule source will produce a set of element nodes in the source document that satisfy the condition, that is, all the XML elements that have an attribute *timeContainer* no matter what the attribute value is. Note that the condition is written in XPath and refers to the variable *e* by using the notation *$e* (line 3). In that way we select all the elements on which some time constraints are imposed.

The target of a model element rule contains a set of actions. Two types of actions are supported: instantiation and update. Only instantiation action will be explained here. An instantiation specifies a type in the target meta-model that will be instantiated. The element created by an instantiation might be assigned with

an identifier. Instantiations enumerate the names of the slots that will be assigned with value after the instantiation. Slot values are determined from an optional expression specified in the slot list and an optional set of slot rules.

The rule *timedElementMapping* contains one instantiation action based on class *TimedElement* and is assigned with the identifier *node* (line 4). Since this is an abstract class the rule cannot be executed and is declared as abstract. The instantiation enumerates the slots that must be assigned with values: *begin*, *end*, *dur* and *ctrlObject*.

The transformation language supports single inheritance among model element rules. The inheriting (or child) rule inherits from the inherited (or parent) rule its source, target and the associated slot rules. Inheriting rule may define its own source, target and slot rules and may override the corresponding inherited components.

Rules *parallelContainer* (lines 17-26) and *intervalNode* (lines 27-31) inherit the rule *timedElementMapping*. They specify the classes that will be instantiated for the parallel time operator and interval node by overriding the instantiation labeled *node* in the parent rule. The identifier is preserved, however, the classes are changed (lines 20 and 30). New classes are concrete and may be instantiated. Slot rules for obtaining the values of slots *begin*, *end* and *dur* are inherited. In this example rule inheritance follows the inheritance in the target meta-model. It allows reusing of the logic for calculating slot values defined in the super class. The rule for processing of sequential time operator is skipped. It is similar to the rule for the parallel operator.

Inheriting rules also inherit the source element and add new conditions. The conditions specified in the inheriting rules are logically and-ed to the inherited condition. Therefore rule *parallelContainer* will be applied on all elements that have attribute *timeContainer* with value 'par' (see the condition in line 19) and rule *intervalNode* will be applied on elements with attribute value 'none' (condition in line 29).

4.3.2 Slot Rules

Slot rules are always associated to a model element rule and specify how to obtain the values of the slots of its instantiations. The syntax of the slot rules is given below:

ruleName RuleSource **target**[(*slotName=Expression*)+]

Every slot rule has a name, a source and a target. Rule target enumerates the slots to be set up with a value. Rule source specifies the elements in the source model that will be used to obtain the value of the slots. A given slot may have more than one slot rule for the calculation of the value. Expressions in the rule source may refer to variables defined in the source of the owner model element rule. In many cases the source of a slot rule is determined relatively to the source of its owner model element rule. For example, the value of the slot *begin* is determined by the rule *beginValue* (lines 6-8). The source of the rule specifies that the value must be taken from an attribute that is located by the XPath expression *$e/@begin* where *e* is the component in the owner model element rule.

To determine the value of a slot the transformation engine first evaluates the expression assigned to the slot in the instantiation. If there is no expression then the value is obtained by executing the associated slot rules. For every match of the source of a slot rule the expression assigned to the slot is evaluated. Results obtained from the matches are united in a set. The sets obtained from the slot rules are united and the result is used as value of the slot. Multiplicity and type constraints are checked.

It should be noted that our example transformation is not complete since there is no slot rule for the slot *ctrlObject*. The reason is that it is not known in advance what the controlled object is and how it is located. This is determined when the timing module is used together with another module to form a full language. Only in that case the information about the controlled object is available. Therefore, a new slot rule should be added based on the specific composition between the timing module and the other module. This is explained in section 4.5 where our example is completed.

4.3.3 Linking Source and Target Elements

Whenever a model element rule is executed the execution engine establishes an association link between the elements matched by the source and the elements instantiated by the target of the rule.

The created target model elements may be located via this association and used as slot values of other model elements created by other rules. They are accessed by querying the source element for the associated elements in the target model. The linking is done by the *link-to* construct that instructs the transformation engine to establish a link between an element of the source and the instantiations in the target of the rule. An example usage of this construct is shown in line 2 where the timed element is associated to the time graph node created for it. This association is used in rule *parallelContainer* to obtain the children nodes of the container. This is done by the slot rule *componentsValue* (line 22-24) where for every child XML element with imposed timing constraints the corresponding time graph node is located by using the built-in function *target* (line 24). This function has two arguments. The first is the variable that holds the source node (in our example *timedChild*) and the second is the identifier of the element in the target model (*node*). The element in the target model may be created by any rule. The important point here is the usage of the same identifier across the rules.

4.4 Transformation Execution

Generally, there are two ways in the transformation engine to execute transformations: by interpretation and by compilation. Currently, rules are executed by interpretation and a prototype of an interpreter has been developed.

Rules are declarative and there is no predefined execution order among them. A single source element may be processed by many rules. The execution of a model element rule is a sequence of instantiations of its target classes. The main problem here is the dependency among the instantiations introduced by the fact that an object may require another object as a slot value. In case of a constructor of a Java class that requires parameters the values of the parameters must be available before the class instantiation. Again, this leads to a dependency among instantiations. Instantiations and their dependencies form a graph. An execution order is derived after a topological sort over the graph. A transformation is executable if the graph does not contain cycles.

4.5 Composing Transformations

In this section we show how our approach supports reuse of XML applications in the context of processing of compound

documents. For this purpose we introduce another simple XML language that will be composed with the timing constructs in our previous example. For this new language we specify a second transformation that will be composed with the first one to form a new transformation capable of processing compound documents that use markup from both languages.

4.5.1 Our Second Example Language

The second language contains primitive elements that encode widgets and container elements that organize the widgets horizontally and vertically. We have two simple widgets for labels and images and two containers. The element names are *label*, *image*, *hbox*, and *vbox* respectively. For simplicity we skip the details about the size, font and color of the elements. A document written in that widget language is visualized by building and showing a hierarchy of layout objects following the nesting hierarchy of the document. Layout objects are again instances of Java classes. The following code gives a sketch of the application class hierarchy.

```
public abstract class LayoutElement {
  public boolean visible;
  public boolean displayed;
  public abstract void draw();
  public abstract void refresh();
}
public abstract class Container extends LayoutElement{
  public Vector components;
}
public class Label extends LayoutElement{
  public String labelText;
  public void draw() {//implementation}
}
public class Image extends LayoutElement{
  public String imageFile;
  public void draw() {//implementation}
}
public class HBox extends Container{
  public void draw() {//implementation}
}
public class VBox extends Container{
  public void draw() {//implementation}
}
```

4.5.2 The Transformation Specification

The transformation specification used to transform documents into a hierarchy of layout widget objects is given below. Some slot rules are omitted to safe space.

```
labelRule ModelElementRule {
   source [e: Element link-to(widget),
             condition{e.name='label'}]
   target [widget: Label{labelText}]
}
imageRule ModelElementRule{
   source [e:Element link-to(widget),
            condition{e.name='image'}]
   target [widget: Image{imageFile}]
}
containerRule abstract ModelElementRule {
   source [e: Element link-to(widget)]
   target [widget: Container{components}]
   SlotRules{
      componentsRule
         source[child:Element=XPath($e/*)]
         target[components=target(child, widget)]
   }
}
hBoxRule ModelElementRule inherits containerRule{
   source[condition{e.name='hbox'}]
   target[widget: HBox]
}
vBoxRule ModelElementRule inherits containerRule{
   source[condition{e.name='vbox'}]
   target[widget: VBox]
}
```

4.5.3 Composing Languages

The structure of the application that processes documents with time constrained widgets is shown in Fig. 6. The schema of the hybrid language contains constructs in the timing module and the widget language. Transformation specification is a composition of the transformations for the two languages. The application model is a composition of the widget classes and time dependency graph classes.

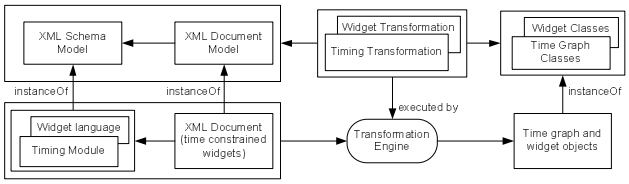

Figure 6. Composition of timing and widget markup languages and their processors

The required step for integration of the two markup languages is to establish an interpretation of the activation and deactivation events in terms of the hosting language. Following the ideas of XHTML+SMIL [29] we choose two possible interpretations of activation. The first one affects the visibility of an element and the second one affects the display of an element. Making an element invisible means that it is not shown on the screen but generates a layout element and still affects the layout of other elements. Turning of the display property means that the element is not shown and no layout element is created for it. This interpretation is specific for the composition of the two languages and needs an explicit indication. Therefore a new attribute will be introduced to indicate the exact time action. Similarly to XHTML+SMIL the attribute name is *timeAction* with two possible values: *visibility* and *display*.

The next step is to integrate the two applications in order to obtain a processor for compound documents. This requires composition of the transformations and the application classes. We first discuss the integration of the application classes.

4.5.4 Composing Application Classes

Every node in the time dependency graph must hold a reference to the object it controls. After the integration of the languages it is known what the controlled objects are. Controlled objects must implement the *ControlledObject* interface, however, class *LayoutElement* does not. Therefore we have a type compatibility problem. To overcome the problem and to perform the required composition between the classes we turn to the Adapter design pattern [6]. We introduce a new Java class that implements

ControlledObject interface and holds a reference to the actual layout element being controlled. The invocations of *activate* and *deactivate* methods are transformed to changing the *visible* and *displayed* properties of the layout objects. Since there are two possible interpretations of activation/deactivation we create one adapter class per interpretation:

```
public class ChangeVisibility implements ControlledObject{
  public LayoutElement obj;
  public void activate(){
    obj.visible=true;
    obj.refresh();
  }
  public void deactivate(){
    obj.visible=false;
    obj.refresh();
  }
}
```

The second class *ChangeDisplay* is implemented in a similar way.

4.5.5 Composing Transformations

Now we can turn to the composition of the transformations. All the rules are reused and new rules must be added to instantiate the glue classes shown above. Two model element rules are added for that purpose and one slot rule that determines the value of the slot *ctrlObject*. Rules are shown below.

```
visibilityRule ModelElementRule{
   source[e:Element link-to(adapter),
           condition{XPath($e[@timeAction='visibility'])}]
   target[adapter:ChangeVisibility{obj=target(e, widget)}]
}
displayRule ModelElementRule{
   source[e:Element link-to(adapter),
           condition{XPath($e[@timeAction='display'])}]
   target[adapter: ChangeDisplay {obj=target(e, widget)}]
}
ctrlObjectValue SlotRule owner=timedElementMapping {
   target[ctrlObject=target(e, adapter)]
}
```

The slot rule is associated to the *timedElementMapping* rule from the first transformation. Apart from the simple additions of rules the transformation language also allows overriding of existing rules.

5. RELATED WORK

Explicit specification of the XML semantics can be done in one of the formalisms used in the area of programming language specification. A number of papers adapt techniques for specification of computer language semantics in the context of XML as a syntactical framework. In [21] the semantics of an XML language is given in the form of an attribute grammar [8]. This opens the possibility for applying the results and tools of extensive research available in that area. In attribute grammars translation is performed over attributed trees. The difference with our approach is that in our approach translation is performed as a transformation from a document tree to a graph.

RelaxNGCC [15] is based on compiler-compiler techniques to build a processor for a language conforming to a RelaxNG schema. This provides more flexibility in bridging between the application model and document syntax and in associating behavior with XML documents. It allows reuse of already existing classes and deals better with structural differences between the syntax and classes. In addition, it generates a dedicated parser for a given XML language. Our transformation rules can be seen as similar schema annotations for W3C schemas and for schema-less documents. The approach for modular and extensible processors presented in [25] is inspired by denotational semantics of computer languages. Processors operate on document trees and do not rely on a schema. Our approach permits both types of processing: document-based and schema based.

Another dimension of the work presented in the paper is the explicit specification of the relation between the syntax and another type of structure (a model, a database, an ontology). The approach presented in [1] maps XML documents to domain ontologies. Mapping rules rely on XPath. The primary goal of mapping is to allow translation of queries over the ontology to queries over the source documents. In our work the rules are used to transform source documents into a set of objects. Other papers that discuss the problem of bridging between XML syntax and ontology are [16] where a mapping ontology is presented that transforms XML documents to their RDF representation and [20] where the authors suggest a unification approach for XML and RDF based on model-theoretic semantics. In [2] a framework for expressing the semantics of markup is defined. The semantics of markup is a set of inferences that can be drawn from the document. PROLOG is used as an implementation language for inference rules. In the context of this work our transformation rules are particular types of inference rules. However, we rely on a domain-specific language for transformation specification and aim at a closer integration with object-oriented programming languages.

There exist a number of languages dedicated to XML processing: XSLT, XDuce [7], XL [5]. All of them transform XML documents to other XML documents and their type system is based on XML types. Our approach is focused on transformation to application objects and uses types from a programming language, in our case Java.

The problem of reuse of language processors and building new languages by composing existing modules has been addressed in research of programming language development and the problem proved to be hard. Existing work studies the composability properties of frameworks for semantics specification: attribute grammars [3], denotational semantics, operational and action semantics [13][12]. These techniques rely on mathematical formalisms to specify the semantics. Transformation rules in our approach may be perceived as specification of the language semantics in a domain-specific transformation language that has features closer to programming languages. In this respect it is more familiar to software developers than the enumerated formal techniques.

There are tools supporting XML processing in browsers. XSmiles [22] is a browser that supports some of the today's popular web languages. Mozilla browser [14] provides a framework for client-side web applications relying on a set of XML languages. Both tools provide an extensible architecture for XML applications. In contrast, our approach does not define an architecture nor a tool but stresses on the explicit specification of the language semantics that can be further employed in a tool. XVM [11] is an extensible architecture for XML processing

based on an association between XML elements and software components that implement their behavior. Our approach allows similar association of processing logic to more complex structures in the document (e.g. a tuple of elements and attributes). In XVM many aspects of the processing may remain hard-coded in the components while our approach declares this explicitly.

6. CONCLUSION AND FUTURE WORK

We presented an approach for development of XML applications based on specification of model transformations from a document model (or document schema) to a set of application-specific classes. In our approach transformations are specified in a model transformation language developed in the context of MDA.

The approach frees the software engineer from writing low level and error-prone code against the generic syntax document model such as DOM and SAX. Instead, a declarative transformation specification is written that establishes the relations between syntax constructs and application structures. Transformation specification has an operational semantics and is executed by an interpreter. We plan to apply a compilation on transformation specifications to produce a dedicated XML processor for a given language. This is inspired by the compiler-compiler approach mentioned in the paper. Transformation rules are considered as semantic annotations.

The main purpose of the approach is to improve adaptability and reusability properties of XML applications. The strength of the approach is that it makes processing logic of the application explicit by expressing it in a set of rules that can be manipulated. However, we observe also problems in achieving these properties.

Adaptability is required to respond to changes in the application in an easy and cheap way without a redesign and recompilation of the whole application. Changes may occur in the language syntax, in the transformation specification and in the application classes. These three aspects are clearly separated from each other and may evolve independently. The most difficult case is the change of the syntax since it usually brings changes in the transformation specification and in the application classes. In our approach the required changes can be isolated in specific rule(s) and classes. The adaptability in that case is derived from the finer control over the application components. Additive adaptations seem to be easier for handling. This type of adaptation usually requires additions of new rules and classes that must be integrated with their counterparts. Deletions and replacement of components may be more difficult since this will require replacement of classes and refactoring of the transformation rules. However, these changes are still isolated and may be done without recompilation and redeployment of the whole application.

Another quality property being pursuit is the reusability of XML applications motivated by the need of hybrid languages and compound documents. Usually an application is reused and composed together with other applications. In that case there are two distinct problems: the composition of transformation rules and the composition of application classes. They are driven by the composition of the XML languages. Composition of transformations is achieved by the available operators in the transformation language. It provides inheritance among rules, additions of both model element and slot rules and modules for reusing a set of rules. Rules may also be overridden. Every language, however, has limitations with respect to the available composition operators. Transformation technology provides us with a solution for composition operators non-supported by the transformation language: transformations may be considered as models and may be manipulated by another transformation. We investigate the required compositional operators for a transformation language and methods for introduction of new operators if required.

The second problem related to the composition of XML applications is the composition of the application classes which is a case of software composition. In general, this is a problem that proved to be difficult. In our example it was easily solved by using the Adaptor design pattern. This is possible if the composition is anticipated and the application is properly designed. In most cases, however, the composition is not anticipated and the application classes are not composable. Composition may be done on source code and on already compiled classes. We can benefit from research in the area of aspect-oriented software development that provides advanced software composition techniques beyond the aggregation and inheritance [4]. This is the main direction for future research.

One problem that remains open is the scalability of the approach. It works on examples that are relatively simple but it is not clear what happens in complex cases with larger number of languages involved. A possible approach is to start with a small stable set of languages and to create modular processors for them that can be composed with each other. This could be the domain of the Web languages. The composition of markup languages is complicated further by the requirement that software engineers should know the details of the language semantics which is not always simple.

In this paper we focused on the possibility to utilize transformations for the specification of XML language semantics. We plan to go further by developing a tool that supports the tasks not discussed here such as deploying, updating and composing transformations and classes in a browser-like environment. This is similar to the architectures that XSmiles and Mozilla provide.

We believe that another benefit of our approach may come from the increasing popularity of MDA and model transformations and the standard set of languages and tools that are expected.

7. REFERENCES

[1] Amann, B., Beeri, C., Fundulaki, I., Scholl, M. Querying XML Sources Using an Ontology-Based Mediator. In proceedings of CoopIS/DOA/ODBASE, 2002

[2] Dubin, D. Object mapping for markup semantics. In B. T Usdin, editor, Proceedings of Extreme Markup Languages 2003, Montreal, Quebec, August 2003

[3] Farrow, R., Marlowe, T.J., and Yellin, D.M., Composable attribute grammars: support for modularity in translator design and implementation. 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Albuquerque, US. 1992

[4] Filman, R., Elrad, T., Clarke, S., and Aksit, M. Aspect-Oriented Software Development. Addison-Wesley. 2004

[5] Florescu, D., Grunhagen, A., and Kossmann, D. XL: an XML programming language for web service specification and composition. 11th international conference on WWW, Honolulu, Hawaii, USA, 2002

[6] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley. 1995

[7] Hosoya, H., Pierce, B. XDuce: A typed XML processing language. In Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pages 226–244, 2000

[8] Knuth, D. Semantics of context free languages. 1968

[9] Kurtev, I., van den Berg, K. Model Driven Architecture Based XML Processing, ACM Symposium on Document Engineering, Grenoble, France, 2003

[10] Kurtev, I., van den Berg, K. A Language for Model Transformations in the MOF Meta-modeling Architecture. Workshop on Model Driven Architecture: Foundations and Applications, Linkoping, Sweden, 2004

[11] Li, Q., Kim, M.Y., So, E. Wood, S. XVM: a Bridge between XML Data and Its Behavior, 13th international conference on WWW, New York, USA, 2004

[12] Mosses, P. Action semantics. Cambridge University Press. 1992

[13] Mosses, P. Pragmatics of Modular SOS, 9th International Conference on Algebraic Methodology and Software Technology, pp. 21-40. 2002

[14] Mozilla Organization, http://www.mozilla.org

[15] Okajima, D. RelaxNGCC - Bridging the Gap Between Schemas and Programs, Available at: http://www.xml.com

[16] Omelayenko B. and Fensel D., A Two-Layered Integration Approach for Product Information in B2B E-commerce, In: K. Bauknecht, S. -K. Madria, G. Pernul (eds.), Electronic Commerce and Web Technologies, Proceedings of the 2nd Int. Conference on Electronic Commerce and Web Technologies, Germany, 2001

[17] OMG. MDA Guide version 1.0.1. OMG document omg/2003-06-01, 2003

[18] OMG. MOF 2.0 Query/Views/Transformations RFP. OMG document ad/2002-04-10, 2002

[19] OMG/Unified Modeling Language Specification. 2001

[20] Patel-Schneider, P., Siméon, J., The Yin/Yang Web: XML Syntax and RDF Semantics, 11th International WWW Conference, Hawaii, USA, 2002

[21] Psaila, G. and S. Crespi-Reghizzi. Adding Semantics to XML. In Second Workshop on Attribute Grammars and their Applications, WAGA'99, 1999

[22] Pihkala K., Honkala M. and Vuorimaa P., A browser framework for hybrid XML documents. 6th International Conference on Internet and Multimedia Systems and Applications, pp 164-169, Kauai, Hawaii, USA, 2002

[23] Reinhold, M. An XML Data-Binding Facility for the Java Platform. 1999

[24] SAX Project Home Page: http://www.saxproject.org/

[25] Sierra, J., L., Fernandez-Manjon, B., Fernandez-Valmayor, A., Navarro, A. An extensible and modular processing model for document trees. Extreme Markup Languages 2002, Montreal, Canada, 2002

[26] W3C. DOM Level 1 Specification, October 1999

[27] W3C. XML Schema Part 0: Primer, Part 1: Structures. 2001

[28] W3C. Synchronized Multimedia Integration Language (SMIL 2.0), 2001

[29] W3C. XHTML+SMIL, 2002

[30] Zhang, Y., and Xu, B. A survey of semantic description frameworks for programming languages. SIGPLAN Notices, vol. 39, 3, pp. 14-30, 2004