published in: Proceedings of Short Papers of the 7th East European Conference on Advances in Databases and Information Systems (ADBIS 2003), pp 38-47

Composition of Transformations for XML Schema Based Documents

Johann Eder and Marek Lehmann

University of Klagenfurt
Dep. of Informatics-Systems
{eder,marek}@isys.uni-klu.ac.at

Abstract. XML is gaining acceptance as a universal format for data interchange and data publication. The proliferation of different document type descriptions or XML Schemas, however, requires still adequate treatment of heterogeneity, in particular the transformation of documents between different schemas by XSLT programs. We propose a new way to define XSLT transformations for XML Schema based documents. Transformations are based on types defined in XML Schemas. Transformations of complex documents may be composed of transformations on subcomponents of these documents. A library stores information about available transformations and their relation to schemas and types and is instrumental in decomposing a document in a way that promotes the application of already available transformations on subcomponents. Therefore the tedious and error-prone work of programming transformations can be made easier and more efficient through reuse.

1 Introduction

The eXtensible Markup Language (XML) [3] has been widely accepted as a universal format for data interchange and publication, in particular for data published on the web or transmitted through web services. As XML is a meta-language, everyone can define his or her own language in XML using document type descriptions (DTD) or XML Schema [8]. The homogeneity XML brings on lower levels is contrasted with the proliferation of different document types, different schemas, different semantics on upper levels of a communication channel. Due to differences in interests, different requirements, progress and optimization with different constraints, to hope for generally accepted standard format for business documents is wishful thinking, but unrealistic at least. Information systems dealing with receiving information from other parties or sending information to them will have to manage heterogeneity. In particular, it is frequently necessary to transform documents between different document types or schemas.

The most widely used and popular way to transform an XML document from one schema to another are XSLT transformations [5]. XSLT allows one to write simple transformations for small documents quite easily, but complexity and error rates increase dramatically with the size of documents. For this reason writing such transformations is very time consuming and tedious. Maintenance and reuse of XSLT code is also very difficult. Therefore, support for programmers is highly desirable. In [7] an approach was proposed to solve this problem by composing and decomposing XML transformations stored in a library and attaching them to components of DTD based XML documents.

In this paper we extend this approach by taking into account the concepts introduced by XML Schema. In particular, we can make use of the typing concept of XML Schema for describing transformations. We propose to build up libraries of well tested transformations between types of XML Schema. In our proposal we emphasize reusing of existing transformations and making use of them in defining new, more complex transformations.

The rest of the paper is organized as follows: in section 2 we briefly review the concepts of XML, DTD, XML Schema and XSLT as much as is necessary for presenting our transformation system. We present our library of elements, types and transformations for XML Schema. Section 3 presents a top-down process for composition and execution of transformations, and introduces the notion of stubs, a concept necessary for composing XSLT transformations without side effects. In section 4 we give an extensive example for stub aware transformations to illustrate our approach. Finally, we draw some conclusions in section 5.

2 Transformation Library

XML is a W3C standard [3] that allows the sharing of data across applications, platforms, and the Internet. In principle, an XML document consist of nested elements. Any well-formed XML document can be represented as a tree in which the element names (tags) describe the nodes. The DOM Model [9], a standard interface for accessing and manipulating XML documents, uses the tree model.

XML is a metalanguage, which can be instantiated with the XML Schema or DTD. Extensibility of XML by defining new grammars was crucial for its success. It also causes problems, if two cooperating parties use different schemas for their documents. This heterogeneity requires transformations of XML documents. We can convert XML data from one representation to another by using XSLT [5].

XSLT is very powerful, but writing and maintaining big XSLT programs is very arduous. The main problem is the lack of support for code reuse. In [7] a proposal was presented to attach XSLT transformations to components. This decomposition was based on DTDs. An XML component is a semantically meaningful unit of the problem domain. In an XML document the tags can be interpreted as delimiters of a component defined in such a way. In the tree model the same component is defined as a subtree identified through its root element.

Due to DTD's weaknesses, there are several other proposals of schema definition methods for XML [2]. One of the most promising is the XML Schema [8]. The most significant difference between DTDs and XML Schema is a possibility do derive new user defined types in the manner similar to the object oriented languages. We can distinguish two kinds of datatypes: simple and complex. Simple types contain neither child elements nor attributes. Complex types have such possibility. Every element has to be of a certain type (simple or complex). At-

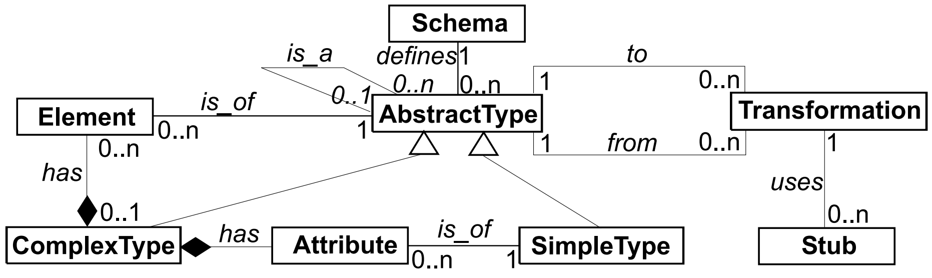

Fig. 1. Metamodel for a Transformation Library

tributes always have a simple type. To simplify this idea we have introduced an *AbstractType* class, which is not specified in [1]. A metamodel of the basic concepts of XML Schema is the basis for our library of transformations (Fig. 1).

The composition of XML documents allows us to write a separate transformation for every component. In our proposal the XML Schema enables us to attach a source type and a target type to XSLT transformations (Fig. 1). Our motivation is to have a library of well tested and reusable transformations between various types which describe often used components. Special techniques described in section 3 combine such transformations.

In the XML Schema a complex type defines the structure of an element. Every such element will be a root element of a DOM subtree corresponding to a component described by this complex type. The DOM Level 2 does not support type information. However, by having type information stored in some external repository we can label each element in the DOM tree with its type.

XML Schema gives the programmers flexibility in defining schemas for the same documents in many different ways. There have been some efforts to define design patterns for the XML Schema [4]. From our point of view the most useful pattern is called the Venetian Blind Design. In this pattern all meaningful components of an XML document (e.g. an address) are defined as named complex types. Such complex type can be easily reused in many parts of a document (e.g. elements *personalAddress* and *officeAddress* both of type *addressT*).

3 Top-Down Transformations

We propose a top-down approach to transformations of XML documents. The input to the transformation is a source document, a type and a name of a target document into which the source document has to be transformed. In the first step the transformer looks for a direct transformation between a source tree and a target tree. If there is no transformation found, the transformer decomposes the source document into subcomponents and tries to transform every subcomponent of the source document into a subcomponent of the target tree. Every transformation treats a transformed component as a separate subtree. In this

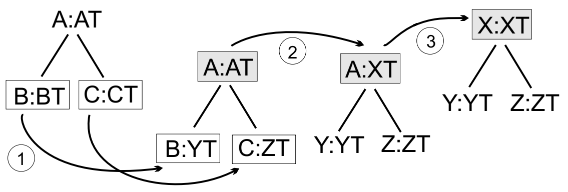

Fig. 2. Transformation with renaming component names at the higher level

way it is guaranteed that the transformation will not see the environment of this component. As the result of all these transformations we will obtain several transformed trees corresponding to desired subcomponents. In the next step the transformer has to glue together these partial results with an additional transformation, which should be simpler than a complex transformation between the source and target type, because it will use well defined partial results.

In our metamodel transformations are attached to the types. But in one document we can have several differently named elements of the same type. Every transformation knows the structure of a type being transformed, but it does not know the root name of a subtree representing this type. This problem concerns both the source and the target type. A transformation can only change the structure of a component from the source into the target type, but it cannot change the root name of a tree corresponding to this component. The name of the root of the transformed subtree can be changed by a transformation at the upper level of the document tree, where this information is available.

In the first step Fig. 2 shows the transformation of two subcomponents: one from type BT into type YT and the other from type CT into type ZT. Both transformations do not change the root names of the transformed subtrees. In the second step we transform their parent element A of the AT type into an element of the XT type. This transformation is possibly very simple, involving just renaming and rearranging of subcomponents. At this level the information about root names of subtrees is available both in the source type AT and in the target type XT. In the last step we just rename the root of the whole tree. This name was given in a user's call to the transformer.

When we apply transformations to all or some subcomponents of a given component, we change a type of this component. The programmer of the transformation for a component modified in such a way has to be aware of changes in its structure. In the example in Fig. 3 a transformation of type AT cannot use any more information from the subtree pointed by the transformed element B. This leads to an idea of employing stubs. A stub is a placeholder for the transformed subcomponents. If a subtree was transformed by a separate partial transformation, then it is replaced by a stub in the source document. In the process of transformation of its parent the stub is a black box. Every transformation of the element on the upper level of the tree has to be aware of stubs. Subsequently every transformation registered in the system has to declare which stubs

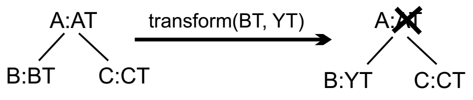

Fig. 3. Type change of A after the transformation of its children

(if any) it is going to use. In every stub description is an XPath [6] localization of a subcomponent which is to be replaced in the source tree, a target type of this subcomponent and a label of this component in the target tree. Localization of the stubs allows one to easily extract all subtrees which correspond to them in the source tree. In the source tree every subtree extracted in such a manner is replaced by a special tag with the stub description. At the end all the stubs will be again replaced by the correctly transformed subcomponents.

In a document which is being transformed the stubs replace all separately transformed subcomponents. A transformation on the upper level does not have access to the content of these subcomponents. This prevents side effects during XSLT transformations such as accidental matching of patterns in the template rules.

Once the subcomponents are extracted from the source tree, they are treated as separate documents. The transformer has to determine their source type by using the schema information and the given XPath expression. The target type of every subcomponent corresponding to the stub is given in the stub description. Having acquired this information the transformer may try to apply recursively the whole procedure for each subcomponent. A transformation between types can change only the structure of the component being transformed. However, it cannot change the name of the root element of this component. Since the label of this root element was given in the stub description, the transformer may just replace this name.

After these operations we have a source document with stubs replacing subcomponents, as well as a set of these subcomponents transformed by other transformations. In the next step the transformer applies the stub aware transformation from the source to the target document. It is important that this transformation does not interfere with the content of the stubs. It copies the stubs into the target document probably rearranging their order. As a result we obtain the target document with stubs located in correct places. Finally we have to replace all stubs by the transformed subcomponents. All steps are presented in Fig. 4.

4 Example of the Stub Aware Top-Down Transformation

In this section we illustrate the presented ideas with an example. We show a process of a stub aware transformation from the source type aT into target type xT. The XML Schema definitions of the source and target types are given in Appendix A and B. The extended notation of the DOM trees of the source and target type can be found in Fig. 5. Our source document is looks as follows:

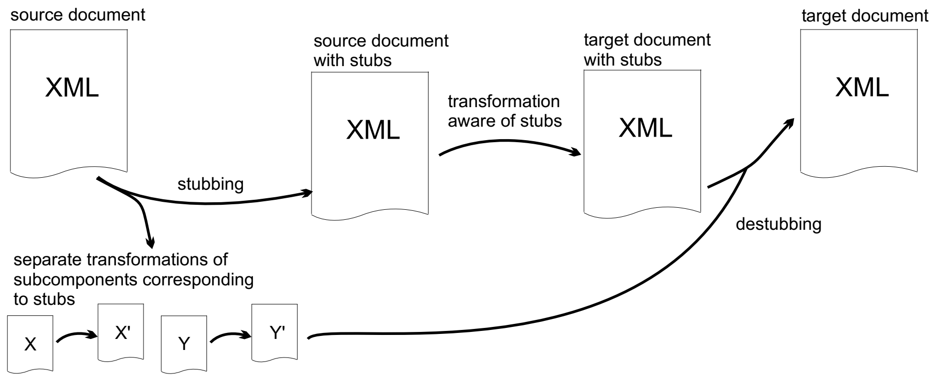
source document
XML
source document with stubs
XML
transformation aware of stubs
target document with stubs
XML
target document
XML
stubbing
destubbing
separate transformations of subcomponents corresponding to stubs
X
X'
Y
Y'

Fig. 4. Stub aware top down transformation

```
<a>
  <b>
    <b1>Element b1</b1> <b2>Element b2</b2>
  </b>
  <c>
    <d>
      <d1>Element d1</d1> <d2>Element d2</d2> <d3>Element d3</d3>
    </d>
    <c1>1</c1>
  </c>
</a>
```

In the analyzed scenario the user has requested our system to transform a document *'a'* from a given schema to a document *'x'* from another schema. The transformer will use schema information to determine the type of both

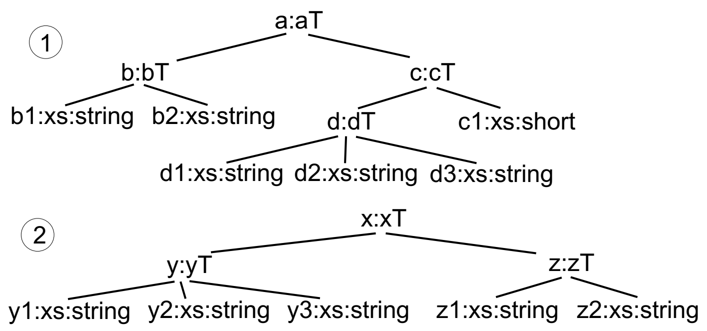

Fig. 5. Hierarchy of the source (1) and target (2) types in our example

documents and attempt to find, among the registered transformations, a transformation between them. There is only a stub aware transformation registered between types aT and xT. Stubs are localized in the source tree aT by an XPath expression: */*/c/d*. An asterisk in this expression matches root names of all differently named elements of the type aT. The description of the transformation is presented as follows:

```
<transformation id="aT2xT">
  <URL>aT2xT.xsl</URL>
  <sourceSch>letters.xsd</sourceSch> <sourceType>aT</sourceType>
  <targetSch>letters.xsd</targetSch> <targetType>xT</targetType>
    <stubs>
      <stub id="stub_dT2yT" targetRoot="y" targetType="yT"
       localization="/*/c/d"/>
    </stubs>
</transformation>
```

In the source document the transformer extracts all components corresponding to the stub description and replaces them with a special tag. As the result we obtain a stubbed source document and a set of separate trees corresponding to extracted components. The transformer has to maintain the association between stubs and a component corresponding to every stub. A stubbed source document and extracted component are shown as follows:

```
<a>
  <b>
    <b1>Element b1</b1> <b2>Element b2</b2>
  </b>
  <c>
    <stub type="stub_dT2yT" id="001"/>
    <c1>1</c1>
  </c>
</a>

<d>
  <d1>Element d1</d1> <d2>Element d2</d2> <d3>Element d3</d3>
</d>
```

After stubbing the transformer can independently work with the stubbed source tree and every tree representing extracted subcomponents. It can apply the following stub aware transformation to the stubbed source document:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="xml" indent="yes"/>
    <xsl:template match="/">
        <xsl:element name="{name(./*)}">
            <xsl:apply-templates select="./*/c/stub"/>
            <xsl:apply-templates select="./*/b"/>
        </xsl:element> </xsl:template>
    <xsl:template match="b">
        <xsl:element name="z">
            <xsl:element name="z1">
                <xsl:value-of select="./b2"/>
            </xsl:element>
            <xsl:element name="z2">
                <xsl:value-of select="./b1"/>
            </xsl:element>
        </xsl:element> </xsl:template>
    <xsl:template match="stub">
        <xsl:copy-of select="."/> </xsl:template>
</xsl:stylesheet>
```

There are a few important issues concerning this transformation. First of all it is a transformation between types. For this reason no assumptions can be made about the root names of the source and target trees. During the process of transformation the root name is simply copied from the source tree to the target tree. Another important issue is the stub awareness. Transformation cannot interfere with the stub content. As a result of this transformation we obtain the following stubbed target tree:

```
<a>
  <stub type="stub_dT2yT" id="001"/>
  <z>
    <z1>Element b2</z1>  <z2>Element b1</z2>
  </z>
</a>
```

The transformer should also transform the extracted components. First it has to determine the type of every component. Since it has already obtained the information about the source schema, this will not be difficult. The target type and the root label of the component were given in the stub description. With the information about the source and target types of components, the transformer can repeat the whole transformation procedure for every component.

In our example we assume that the transformer finds a direct transformation between type dT and yT. Once again it is a transformation between types and there is no awareness of the root names of documents being transformed. But since the root label of a target component was given in the stub description, after performing the transformation, the transformer can rename the whole component tree.

After the transformation of the stubbed source tree and all extracted components, the transformer can start the process of destubbing. Every stub in the transformed document should be replaced by a corresponding component. In the final phase the transformer should rename the root of the resulting tree due to the name given in the user call. As a result we obtain the target document:

```
<x>
    <y>
       <y1>Element d3</y1> <y2>Element d2</y2> <y3>Element d1</y3>
    </y>
    <z>
      <z1>Element b2</z1> <z2>Element b1</z2>
    </z>
</x>
```

5 Conclusions

We have presented a new approach to transformations of XML Schema based documents. Our proposal consists of the following:

– Assigning source types and target types to transformation
– Composition of transformations out of transformations for subtrees using stubs to avoid side effects
– A meta structure for storing XML Schemas and associated transformations in a library
– A general decomposition and transformation process

We have described how to compose new transformations for complex documents by using the existing transformations of the subcomponents belonging to these documents. In order to do that we have introduced the idea of stubs. In our approach a stub is a placeholder for partially transformed subcomponent of a bigger document.

The method introduced in this paper is intended to increase the efficiency of the development process for creating transformations between XML documents. This can be achieved through reuse of existing transformations and the tendency to write smaller transformations for well defined components. These are then used to define transformations for more complex documents. The way an XML Schema is designed will have influence on how easily the transformations can be applied. We believe that the design model of XML schemas called the Venetian Blind Model is the most suitable.

References

1. P. V. Biron, A. Malhotra: *XML Schema Part 2: Datatypes.* W3C Recommendation
2. A. Bonifati, D. Lee: *Technical Survey of XML Schema and Query Languages* Technical report, 2001
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler: *Extensible Markup Language (XML) 1.0 (Second Edition).* W3C Recommendation
4. K. Cagle, J. Ducket et al.: *Professional XML Schema.* Wrox Press, 2001
5. J. Clark *XSL Transformations (XSLT) v1.0.* W3C Recommendation
6. J. Clark, S. DeRose: *XML Path Language (XPath) v1.0.* W3C Recommendation
7. J. Eder, W. Strametz: *Composition of XML-Transformations.* LNCS 2115
8. D. C. Fallside: *XML Schema Part 0: Primer.* W3C Recommendation
9. A. Le Hors et al.: *Document Object Model (DOM) Level 2 Core Specification v1.0.* W3C Recommendation

Appendix A: The Source Type in the Example

```
<xs:element name="a" type="aT"/> <xs:complexType name="aT">
  <xs:sequence>
    <xs:element name="b" type="bT"/>
    <xs:element name="c" type="cT"/>
  </xs:sequence>
</xs:complexType> <xs:complexType name="bT">
  <xs:sequence>
    <xs:element name="b1" type="xs:string"/>
    <xs:element name="b2" type="xs:string"/>
  </xs:sequence>
</xs:complexType> <xs:complexType name="cT">
  <xs:sequence>
    <xs:element name="d" type="dT"/>
    <xs:element name="c1" type="xs:short"/>
  </xs:sequence>
</xs:complexType> <xs:complexType name="dT">
  <xs:sequence>
    <xs:element name="d1" type="xs:string"/>
    <xs:element name="d2" type="xs:string"/>
    <xs:element name="d3" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Appendix B: The Target Type in the Example

```
<xs:element name="x" type="xT"/> <xs:complexType name="xT">
  <xs:sequence>
    <xs:element name="y" type="yT"/>
    <xs:element name="z" type="zT"/>
  </xs:sequence>
</xs:complexType> <xs:complexType name="yT">
  <xs:sequence>
    <xs:element name="y1" type="xs:string"/>
    <xs:element name="y2" type="xs:string"/>
    <xs:element name="y3" type="xs:string"/>
  </xs:sequence>
</xs:complexType> <xs:complexType name="zT">
  <xs:sequence>
    <xs:element name="z1" type="xs:string"/>
    <xs:element name="z2" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```