Evolving XML Schemas and Documents Using UML Class Diagrams*

Eladio Domínguez[1], Jorge Lloret[1], Ángel L. Rubio[2], and María A. Zapata[1]

[1] Dpto. de Informática e Ingeniería de Sistemas,
Facultad de Ciencias. Edificio de Matemáticas,
Universidad de Zaragoza. 50009 Zaragoza. Spain
{noesis, jlloret, mazapata}@unizar.es
[2] Dpto. de Matemáticas y Computación. Edificio Vives,
Universidad de La Rioja. 26004 Logroño. Spain
arubio@dmc.unirioja.es

Abstract. The widespread use of XML brings new challenges for its integration into general software development processes. In particular, it is necessary to keep the consistency between different software artifacts and XML documents when evolution tasks are carried out. In this paper we present an approach to evolve XML schemas and documents conceptually modeled by means of UML class diagrams. Evolution primitives are issued on the UML class diagram and are automatically propagated down to the XML schema. The XML documents are also automatically modified to conform to the new XML schema. In this way, the consistency between the different artifacts involved is kept. This goal is achieved by using an intermediate component which reflects how the UML diagrams are translated into the XML schemas.

1 Introduction

XML [17] is increasingly used as a standard format for data representation and exchange across the Internet. XML Schema [16] is also the preferred means of describing structured XML data. These widespread uses bring about new challenges for software researchers and practitioners. On the one hand, there is a need for integrating XML schemas into general software development processes. The production of XML schemas out of UML models [1,8,14] or the binding of XML schemas to a representation in Java code [7] are examples of the relationships between XML and development processes. On the other hand, XML documents (and, in particular, XML schemas) are not immutable and must change over time for various varied reasons, as for example widening the scope of the application or changes in the requirements [15].

In these circumstances, it seems highly valuable to have a framework where XML evolution tasks can be performed while ensuring that consistency between

* This work has been partially supported by DGES, project TIC2002-01626, by the Government of La Rioja, project ACPI2002/06, by the Government of Aragón and by the European Social Fund.

K.V. Andersen, J. Debenham, and R. Wagner (Eds.): DEXA 2005, LNCS 3588, pp. 343–352, 2005.

the different artifacts involved (documents, models, code) is kept. The general objective of our research is to obtain such a complete framework. As a step in this direction, in this paper we present our approach in order to evolve UML–modeled XML data (XML schemas and documents) by means of applying evolution operations on the UML class diagram.

This work relies on our own scaffolding architecture, presented in [4,5], that contributes to the achievement of a satisfactory solution to analogous problems in the database evolution setting. One of the main characteristics of this architecture is an explicit translation component that allows properties of traceability and consistency to be fulfilled when evolution tasks are carried out. In the present paper we use this architecture as a framework to perform XML evolution activities and, in particular, we explain with a certain degree of detail the algorithms associated to that translation component.

The remainder of the paper is as follows. In section 2, we present an overview of our architecture for evolution. Section 3 is devoted to the algorithm for translating a UML class diagram into an XML schema while section 4 deals with the algorithm for propagating changes from the UML class diagram to the XML schema and XML documents. In section 5 we review related work and finish with the conclusions and future work.

2 Evolution Architecture Overview

As it is said in the introduction, the scaffolding of our approach is constituted by an architecture we presented in [4,5] applied within a database evolution setting. Although the architecture has been proven within this setting, it was designed with the aim of being independent of any particular modeling technique. This fact has allowed us to apply the same architectural pattern to the XML context.

The architecture is shaped along two dimensions. On the one hand, the different artifacts of the architecture are divided into three abstraction levels which fit with the metamodel, model and data layers of the MOF metadata architecture [11]. On the other hand, the architecture is also layered on the basis of several structures that model different development phases. More specifically, the architecture includes a *conceptual component*, a *translation component*, a *logical component* and an *extensional component*. We will describe briefly the meaning and purpose of each component (see [4,5] for details).

The *conceptual component* captures machine–independent knowledge of the real world. For instance, in the case of database evolution, this component would deal with entity–relationship schemas. In the XML evolution approach proposed in the present paper, the conceptual component deals with UML class diagrams modeling the domain. The *logical component* captures tool–independent knowledge describing the data structures in an abstract way. In database evolution, this component would deal with schemas from the relational model, as for instance by means of standard SQL. In the case of XML evolution, the logical models univocally represent the XML schemas. The *extensional component* captures tool dependent knowledge using the implementation language. In databases, it

would deal with the specific database in question, populated with data, and expressed in the SQL of the DBMS of choice. Within the XML context, the textual structure of data is represented using XML Schema and the data are specified in textual XML documents conforming to an XML schema. One of the main contributions of our architecture is the *translation component*, that not only captures the existence of a transformation from elements of the conceptual component to other of the logical one, but also stores explicit information about the way in which concrete conceptual elements are translated into logical ones.

More specifically, the way of working of our architecture within the XML context is as follows: given a UML class diagram representing a data structure, it is mapped into a XML schema applying a translation algorithm. XML documents conforming to the resultant XML schema can be created. For various reasons, the data structure may need to be changed. In this case, the data designer must issue the appropriate evolution transformations to the conceptual UML diagram. The existence of the translation component allows these changes to be automatically propagated (by means of the propagation algorithm) to the other components. In this way the extensional XML schema is changed, and consequently the XML documents are also changed so as to conform them to the new XML schema.

3 Translation Algorithm

There are several papers [8,14] where the generation of XML schemas from UML class diagrams is proposed. Paper [8] proposes a generation based on transformation rules and in Table 1 we offer a summary of this approach.

Table 1. Rules for generating XML schemas from UML schemas

UML block	XML item(s)
class	element, complex type, with ID attribute, and key
attribute	subelement of the corresponding class complex type
association	reference element, with IDREF attribute referencing the associated class and keyref for type safety (key/keyref references)
generalization	complex type of the subclass is defined as an extension of the complex type of the superclass

Our goal is not only the generation of the XML schema but also the automatic management of its evolution. For this purpose, we have defined an intermediate component which allows us to maintain the traceability between the UML and XML schemas.

In order to deal with this intermediate component, we have enriched the notion of transformation by developing the notion of *translation rule*. The translation rules are used inside the *translation algorithm*. When this algorithm is applied to the UML class diagram it produces not only the XML schema but also a set of elementary translations stored in the intermediate component. An elementary translation is the smallest piece of information reflecting the correspondence between the UML elements and the XML items.

elementary_translation

elem_transl_id	type	conceptual_element	logical_item
1	ETT20	employee.name	name element of employeeType type
2	ETT20	employee.department	department element of employeeType type
3	ETT25	employee	idEmployee attribute of employeeType type
4	ETT60	-	key of employee element of complexType of root element
5	ETT01	employee	employee element of complexType of root element
6	ETT05	employee	employeeType complexType
7	ETT00	enterprise	enterprise root element

Fig. 1. Elementary translations after applying the translation to our running example

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="enterprise">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="employee" type="employeeType"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:key name="keyEmployee">
   <xsd:selector xpath="employee"/>
   <xsd:field xpath="@idEmployee"/>
  </xsd:key>
 </xsd:element>
<xsd:complexType name="employeeType">
 <xsd:sequence>
  <xsd:element name="name"
   minOccurs="1" maxOccurs="1"/>
  <xsd:element name="department"
   minOccurs="0" maxOccurs="1"/>
 </xsd:sequence>
 <xsd:attribute name="idEmployee"
  type="xsd:ID" use="required"/>
</xsd:complexType>
</xsd:schema>
```

Fig. 2. Initial extensional XML schema for our running example

Translation rule. We have defined our translation rules taking the transformation rules proposed in [8] as a starting point. Each translation rule basically includes procedures for creating the XML items of the XML schema enriched with procedures for creating the elementary translations. For example, a translation rule for classes defines, among other things, the name of the XML element into which each class is translated as well as the elementary translations to be added to the translation component.

Translation algorithm. This algorithm takes as input conceptual building block instances of the UML schema and creates 1) the elementary translations 2) the logical elements of the *logical XML schema* and 3) the *extensional XML schema*. More details about the translation algorithm can be found in [4].

As we can see, in our setting an XML schema admits two different views. In the first, the XML schema is an instance of the metamodel for XML and each item of the XML schema is an instance of a metaclass of the XML metamodel. In the second view, it is a sequence of characters encoding tree–structured data following rules specified by the XML standard. From now on, when we refer to the first view, we use the term *logical XML schema* while for the second view we use the term *extensional XML schema*.

Example. We consider a UML schema of a company where there is a class `employee` with attributes `name` and `department`. When we apply the translation algorithm to this UML schema, we obtain the elementary translations shown in Figure 1, the corresponding items of the logical XML schema (not shown in this paper) and the initial extensional XML schema shown in Figure 2. For example, inside this algorithm, when the translation rule for classes is applied to the class

employee, some of the obtained results are: the name of the XML element for the class *employee* is also *employee* (as can be seen in line 5 in Figure 2) and the elementary translation numbered 5 in Figure 1 is added to the translation component. This elementary translation reflects the fact that the class *employee* is translated into the element *employee* of the root element.

4 Propagation Algorithm

The propagation algorithm propagates changes made in the UML schema to the XML schema and XML documents. It is split into propagation subalgorithms for the intermediate, for the logical and for the extensional levels. In this paper, we briefly describe the first two subalgorithms while concentrating on the latter because this subalgorithm is responsible for the evolution of the extensional XML schema and documents.

Propagation subalgorithm for the intermediate and logical levels. In order to change automatically the XML schema and the XML documents, the data designer issues appropriate evolution primitives to the UML diagram. These primitives are basic operations such as addition or deletion of modeling elements (class, attribute, association), transformation of an attribute into a class and so on.

For example, to transform the attribute `employee.department` into a class, the primitive `attribToClass('employee.department')` is executed. This transformation (1) adds to the UML class diagram a `department` class described by the attribute `department`, (2) adds a binary association `employee has department` and, (3) deletes the attribute `employee.department`.

The conceptual changes are the input for the propagation subalgorithm for the intermediate level, which updates the elementary translations of the intermediate component to reflect these changes. After applying this subalgorithm in our running example, the resulting intermediate component is shown in Figure 3 where the elementary translation number 2 has been deleted and the elementary translations from 8 to 18 have been added.

The information about the changes performed in the intermediate component is the input for the propagation subalgorithm for the logical level, which changes the logical XML schema by triggering a set of procedures. Let us see a general description of the procedures which are executed for the `attribToClass` primitive:

(a) **addType**. Creates a new type in the logical XML schema.
(b) **addRootChildElement**. Creates a new child element of the root element.
(c) **addAttributeForType**. Adds a new attribute to a type.
(d) **addKey**. Adds a key to an element of the root type.
(e) **emptyElement**. A nested element is transformed into a non–nested element.
(f) **addKeyref**. Creates a new keyref.

Propagation subalgorithm for the extensional level. This subalgorithm (see sketch in Table 2) takes as input the changes produced in the logical XML

elementary_translation

	elem_transl_id	type	conceptual_element	logical_item
	1	ETT20	employee.name	name element of employeeType type
	~~2~~	~~ETT20~~	~~employee.department~~	~~department element of employeeType type~~
	3	ETT25	employee	idEmployee attribute of employeeType type
	4	ETT60	-	key of employee element of complexType of root element
	5	ETT01	employee	employee element of complexType of root element
	6	ETT05	employee	employeeType complexType
	7	ETT00	enterprise	enterprise root element
added elementary translations	8	ETT20	department.department	department element of departmentType type
	9	ETT25	department	idDepartment attribute of departmentType type
	10	ETT60	-	key of department element of complexType of root element
	11	ETT01	department	department element of complexType of root element
	12	ETT05	department	departmentType complexType
	13	ETT02	binaryAssociation employee has department	department element of employeeType type
	14	ETT21	department	idDepartment attribute of department element of employeeType type
	15	ETT75	multiplicity constraint 0..1	minOccurs and maxOcurrs in the department element of employeeType type
	16	ETT75	multiplicity constraint 0..*	-
	17	ETT65	exists constraint exist1	-
	18	ETT65	exists constraint exist2	keyref from employee to department

Fig. 3. Elementary translations after applying the attribToClass primitive to our running example

Table 2. Sketch of the propagation subalgorithm for the extensional level

```
INPUT: Set of operations on the logical XML schema
OUTPUT: Set of XSLT stylesheets to be applied to the old extensional XML
schema and to the XML documents
 For each operation o of the INPUT
  If the operation o is to add on the logical metaclass metaclass_i and
     the conceptual evolution primitive is concept_primit_i1 then
      XML_sch_i11;
      ...
      XML_sch_i1r1;
      XML_doc_i1r1+1;
      ...
      XML_doc_i1n1;
  endif
  If the operation o is to add on the logical metaclass...
 endfor
```

schema and updates the extensional XML schema as well as the XML documents in order to reflect these changes.

In order to do such updates, each change of the logical XML schema triggers one or more XSLT stylesheets which, on the one hand, change the extensional XML schema and, on the other hand, change the XML documents to conform to the new extensional XML schema. The XML stylesheets are executed by procedures of which we distinguish two kinds: the XML_sch* procedures execute stylesheets that act on the extensional XML schema and the XML_doc* procedures execute stylesheets that act on the XML documents. Every procedure has been designed to maintain the consistency between the XML documents and the XML schema.

In our running example, the changes in the logical XML schema produced by the procedures (a) to (f) mentioned above, as applied to our running example, are the input for this subalgorithm. For these changes, the algorithm executes

Fig. 4. XML stylesheets applied to the extensional XML schema and to the XML documents after adding a new element to the root element in our running example

the corresponding XML_sch* or XML_doc* procedures, which apply their XML stylesheets. In total, five schema stylesheets and four document stylesheets are applied. An example of the applied stylesheets for the logical procedure (b) is shown in Figure 4, where the identity template as well as the headers have been omitted. Let us explain the meaning of each procedure triggered by the changes made by the (b) procedure.

XML_sch_addRootChildElement(element:string, type:string)

Precondition: The type exists in the XML schema.

Semantics: Modifies the extensional XML schema in order to add to it a new element in the sequence of the complex type of the root element. The type of the new element is `type`.

Effect in the running example: Generates and executes on the extensional XML schema the stylesheet (1) of Figure 4. As a result, the element `department` is added to the extensional XML schema (see sixth line in Figure 5).

XML_doc_addRootChilds(d:xpath_expression,rootelement:string)

Precondition: d is an xpath expression of the form $rootelement\backslash element_1\backslash ...\backslash element_n$ and $element_n$ is a terminal element.

Semantics: Copies each node of the node set defined by the xpath expression d as a child of the root node. Moreover, there are no two nodes among the just copied nodes with the same value.

Effect in the running example: Generates and executes on the XML documents the stylesheet (2) of Figure 4.

XML_doc_addParentElement(d:xpath_expression,element_name:string)

Precondition: d is an xpath expression of the form $rootelement\backslash element_1\backslash ...\backslash element_n$

Semantics: Each node of the node set defined by the xpath expression d is included as a content of a new element node with the name element_name.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="enterprise">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="employee" type="employeeType"
     minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element name="department" type="departmentType"
     minOccurs="0" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:complexType>
  <xsd:key name="keyEmployee">
   <xsd:selector xpath="employee"/>
   <xsd:field xpath="@idEmployee"/>
  </xsd:key>
  <xsd:key name="keyDepartment">
   <xsd:selector xpath="department"/>
   <xsd:field xpath="@idDepartment"/>
  </xsd:key>
  <xsd:keyref name="ref1" refer="keyDepartment">
   <xsd:selector xpath="./employee/department"/>
   <xsd:field xpath="@idDepartment"/>
  </xsd:keyref>
 </xsd:element>
<xsd:complexType name="employeeType">
 <xsd:sequence>
  <xsd:element name="name" minOccurs="1"
   maxOccurs="1"/>
  <xsd:element name="department"
   minOccurs="0" maxOccurs="1">
    <xsd:complexType>
     <xsd:attribute name="idDepartment"
      type="xsd:IDREF" use="required"/>
    </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
 <xsd:attribute name="idEmployee"
  type="xsd:ID" use="required"/>
</xsd:complexType>
<xsd:complexType name="departmentType">
 <xsd:sequence>
  <xsd:element name="department"
   minOccurs="1" maxOccurs="1"/>
 </xsd:sequence>
 <xsd:attribute name="idDepartment"
  type="xsd:ID" use="required"/>
</xsd:complexType>
</xsd:schema>
```

Fig. 5. Final extensional XML schema (in bold, modified parts from the initial extensional XML schema)

There is a new element for each node of the node set. The xpath expression for the new nodes will be $rootelement\backslash element_1\backslash ...\backslash element\_name\backslash element_n$

Effect in the running example: Generates and executes on the XML documents the stylesheet (3) of Figure 4.

In Figure 5 we can see the final extensional XML schema that is obtained after applying the XML schema stylesheets generated by the procedures triggered by the (a) to (f) procedures.

We have implemented our approach with Oracle 10g Release 1 and PL/SQL. In particular, we have used the DBMS_XMLSCHEMA package and its CopyEvolve() procedure. This procedure allows us to evolve XML schemas registered in the Oracle XML DB database in such a way that existing XML instance documents continue to be valid.

5 Related Work

There exist in the literature various proposals for managing the evolution of XML documents, [15] being the most sound proposal since it provides a minimal and complete taxonomy of basic changes which preserve consistency between data and schema. The problem with these proposals is that the data designer has to perform the evolution changes working directly with the XML documents, so that (s)he is concerned with some low-level implementation issues [14].

Like other authors [1,3,8,14], we advocate using a conceptual level or a platform independent level for XML document design. However, we also consider that the possibility of performing evolution tasks at a conceptual level is advisable, since it allows the data designer to work at a higher degree of abstraction. The problem is that, to our knowledge, the approaches that propose a conceptual

modeling language for data design, generating the XML documents automatically, do not take into account evolution issues [8,14]. Furthermore, the authors that deal with evolution tasks at a conceptual level do not apply them for the specific case of XML documents [6]. For this reason, as far as we are aware, our proposal is the first framework including a conceptual level for managing XML document evolution tasks.

With regard to other evolution frameworks that consider a conceptual level [6,9], most of these are proposed for the specific database evolution field. The main challenge of these proposals is to maintain the consistency between models of different levels that evolve over time. We tackle this problem, as [6] does, by ensuring the traceability of the translation process between levels. But, although the traceability of transformation executions is a feature required in several proposals (see, for example, QVT [12]), there is no agreement about which artifacts and mechanisms are needed for assuring this traceability [13,18]. In [6] the traceability is achieved storing the sequence (called history) of operations performed during the translation of the conceptual schema into a logical one. In this way the mappings affected by the changes can be detected and modified, whereas the rest can be reexecuted without any modification. The main difference between this approach and ours is the type of information stored for assuring traceability. Whereas in [6] the idea is to store the history of the process performed (probably with redundancies), in our case the goal of the elementary translations is to reflect the correspondence between the conceptual elements and the logical ones, so there is no room for redundancies.

6 Conclusions and Future Work

The main contribution of this work is the presentation of a framework for managing XML document evolution tasks. This framework includes a conceptual level and a logical one, and the consistency between them is kept ensuring the traceability of the translation process between levels. More specifically, we have described, by means of an example, the component that reflects the correspondence between conceptual and logical elements. For this purpose, elementary translations that reflect the relations between the conceptual elements and the logical ones and that facilitate evolution tasks are used. Furthermore the propagation algorithm which guarantees the consistency between the XML schema and documents has been explained.

There are several possible directions for future work. Our solution has been implemented using a particular tool, while approaches such as MDA [10] promise the future development of general model–driven tools that will provide further automatized support to evolution tasks. Because of that we will work on approaching our solution to these other model–driven proposals. In particular, the specification of the transformations involved in our proposal by means of a unified transformation language such as it is demanded in the QVT request for proposal [12] is a goal for further development. Besides, the present proposal takes a forward maintenance perspective, and how to apply our ideas for a

round–trip perspective [2] remains an ongoing project. Another direction is how to apply the architecture to other contexts, such as, for example, for managing the binding of XML schemas to a representation in Java code [7].

References

1. M.Bernauer, G. Kappel, G. Kramler, Representing XML Schema in UML – A Comparison of Approaches, in N. Koch, P. Fraternali, M. Wirsing, Martin (Eds.) *Web Engineering - ICWE 2004* LNCS 3140, 2004, 440–444.
2. P. A. Bernstein, Applying Model Management to Classical Meta Data Problems, *First Biennial Conference on Innovative Data Systems Research- CIDR 2003*.
3. R. Conrad, D. Scheffner, J. C. Freytag, XML Conceptual Modeling Using UML, in Alberto H. F. Laender, Stephen W. Liddle, Veda C. Storey (Eds.) *Conceptual Modeling - ER 2000* LNCS 1920, 2000, 558–571.
4. E. Domínguez, J. Lloret, A. L. Rubio, M. A. Zapata, Elementary translations: the seesaws for achieving traceability between database schemata, in S. Wang et al, (Eds.), *Conceptual modeling for advanced application domains- ER 2004 Workshops*, LNCS 3289 , 2004, 377–389.
5. E. Domínguez, J. Lloret, M. A. Zapata, An architecture for Managing Database Evolution, in A. Olivé et al. (eds) *Advanced conceptual modeling techniques- ER 2002 Workshops*, LNCS 2784 , 2002, 63–74.
6. J.M. Hick, J.L. Hainaut, Strategy for Database Application Evolution: The DB-MAIN Approach, in I.-Y. Song et al. (eds.) *ER 2003*, LNCS 2813, 291–306.
7. *Java Architecture for XML Binding (JAXB)*, available at `http://java.sun.com/xml/jaxb/`.
8. T. Krumbein, T. Kudrass, Rule-Based Generation of XML Schemas from UML Class Diagrams, in Robert Tolksdorf, Rainer Eckstein (Eds.), *Berliner XML Tage 2003* XML-Clearinghouse 2003, 213–227
9. J. R. López, A. Olivé, A Framework for the Evolution of Temporal Conceptual Schemas of Information Systems, in B. Wangler, L. Bergman (eds.), *Advanced Information Systems Eng.- CAiSE 2000*, LNCS 1789, 2000, 369–386.
10. J. Miller, J. Mukerji (eds.), MDA Guide Version 1.0.1, Object Management Group, Document number omg/2003-06-01, May, 2003.
11. OMG, *Meta Object Facility (MOF) specification*, version 1.4, formal/02–04–03, available at http://www.omg.org, April, 2002.
12. OMG, *MOF 2.0 Query / Views / Transformations RFP*, ad/2002–04–10, available at http://www.omg.org, 2002.
13. B. Ramesh, Factors influencing requirements traceability practice, *Communications of the ACM*, 41 (12), December 1998, 37-44.
14. N. Routledge, L. Bird, A. Goodchild, UML and XML schema, in Xiaofang Zhou (Ed.), *Thirteenth Australasian Database Conference, 2002*, 157–166
15. H. Su, D. Kramer, L. Chen, K. T. Claypool, E. A. Rundensteiner, XEM: Managing the evolution of XML Documents, in K. Aberer, L. Liu(Eds.) *11th Intl. Workshop on Research Issues in Data Engineering*, IEEE 2001, 103-110.
16. W3C XML Working Group, *XML Schema Parts 0–2 (2nd ed)*, available at http://www.w3.org/XML/Schema♯dev.
17. W3C XML Working Group, *Extensible Markup Language (XML) 1.0 (3rd ed)*, available at http://www.w3.org/XML/Core/♯Publications.
18. W. M. N. Wan-Kadir, P. Loucopoulos, Relating evolving business rules to software design, *Journal of Systems Architecture*, 50 (7), july 2004, 367-382.