XML Schema Transformations

The ELaX Approach

Thomas Nösinger, Meike Klettke, and Andreas Heuer

Database Research Group
University of Rostock, 18051 Rostock, Germany
(tn, meike, ah)@informatik.uni-rostock.de

Abstract. In this article the transformation language ELaX (Evolution Language for XML-Schema) for modifying existing XML Schemas is introduced. This domain-specific language was developed to fulfill the crucial need to handle modifications on an XML Schema and to express such modifications formally. The language has a readable, simple, base-model-oriented syntax, but it is able to also express more complex transformations by using add, delete and update operations. A small subset of operations of the whole language is presented and illustrated partially by dealing with a real life XML Schema of the WSWC (Western States Water Council). Finally, the idea of integrating an ELaX interface into an existing research prototype for dealing with the co-evolution of corresponding XML documents is presented.

1 Introduction

The eXtensible Markup Language (XML) [2] is one of the most popular formats for exchanging and storing structured and semi-structured information in heterogeneous environments. To assure that well-defined XML documents can be understood by every participant (e.g. user or application) it is necessary to introduce a document description, which contains information about allowed structures, constraints, data types and so on. XML Schema [6] is one commonly used standard for dealing with this problem. An XML document is called valid, if it fulfills all restrictions and conditions of an associated XML Schema.

After using an XML Schema the requirements against exchanged information can change over time. To meet these requirements the schema has to be adapted, for example if additional elements are added into an existing content model, the data type of information are changed or integrity constraints are introduced. All in all, every possible structure of an XML Schema definition (XSD) can be changed. The occurring problem is: how can adaptions be described and formulated under consideration of the underlying XML schema definition in a descriptive, intuitive and easy-understandable way? The definition of a schema update language is absolutely necessary; we introduce such a language in this paper.

A further issue, not covered in this paper, but important in the overall context of exchanging information, is the validity of XML documents. The resulting

problem of modifying an XML Schema is, existing XML documents, which were valid against the former XML Schema, have to be adapted as well (also known as co-evolution). One unpractical way to handle this problem is to introduce different versions of an XML Schema, but in this case all versions have to be stored and every participant in the heterogeneous environment has to understand all different document descriptions. An alternative solution is the evolution of the XML Schema, so that just one document description exists at one time. The above mentioned validity problem of XML documents is not solved, but with the standardized description of the adaptions (e.g. a sequence of operations of an update language) it is possible to automatically derive necessary XML document transformation steps based on these adaptions [14].

The new evolution language for XML Schema (ELaX - Evolution Language for XML-Schema) is our answer to the above mentioned problems. With ELaX it is possible to describe and formulate XML Schema modifications under consideration of the underlying model (XSD). Furthermore, it is an essential prerequisite for the here not in detail but incidentally handled process of the evolution of XML Schema.

This paper is organized as follows. **Section 2** introduces a running example, which defines a realistic scenario for the use of ELaX. **Section 3** gives the necessary background of XML Schema and corresponding concepts. **Section 4** and **section 5** present our approach, by first specifying the basic statements of ELaX and then showing how our approach can contribute to the scenario discussed in **section 2**. In **section 6** we describe the practical use of ELaX in our prototype, which was developed for handling the co-evolution of XML Schema and XML documents. In **section 7** we discuss some related transformation language for XML Schema. Finally, in **section 8** we draw our conclusions.

2 Running Example

Information exchange specifications usually provide some kind of XML Schema which contains information about allowed structures, constraints, data types and so on. One example is the WSWC (Western States Water Council), an organization which accomplishes effective cooperation among 18 western states in the conservation, development and management of water resources. Another purpose is the exchange of views, perspectives and experiences among member states - summarized the exchange of information. The expected format for XML data exchange is defined in a set of XML Schemas (current draft v.02), one XML Schema for reports is presented in the following. Due to space limitations, the chosen example is a simple one, annotations were folded and some parts were deleted and replaced by "[..]".

The original XML Schema is illustrated in figure 1. According to the complex type ("ReportDataType") a report (the component "Report") is a sequence of an obligatory report identifier ("WC:ReportIdentifier"), an optional name ("WC:ReportName") and a set of reported units ("WC:ReportingUnit"). The element declarations not given in this schema are specified in external XML

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:WC="http://www.exchangenetwork.net/schema/WC/0"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.exchangenetwork.net/schema/WC/0"> [..]
   <xsd:include schemaLocation="WC_ReportingUnit_v0.2.xsd"/>
   <xsd:annotation/>
   <xsd:element name="Report" type="WC:ReportDataType">[..]</xsd:element>
   <xsd:complexType name="ReportDataType">
      <xsd:annotation/>
      <xsd:sequence>
         <xsd:element ref="WC:ReportIdentifier"/>[..]
         <xsd:element ref="WC:ReportName" minOccurs="0"/>[..]
         <xsd:element ref="WC:ReportingUnit" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:schema>
```

Fig. 1. The original WSWC report XML Schema

Schemas, which are represented by the "xsd:include" component. This original report is adapted; the result is the XML Schema presented in figure 2 (changed and added parts are highlighted by rectangles).

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:WC="http://www.exchangenetwork.net/schema/WC/0"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   targetNamespace="http://www.exchangenetwork.net/schema/WC/0"> [..]
   <xsd:include schemaLocation="WC_ReportingUnit_v0.2.xsd"/>
   <xsd:annotation/>
   <xsd:element name="Report" type="WC:ReportDataType">[..]</xsd:element>
   <xsd:complexType name="ReportDataType">
      <xsd:annotation/>
      <xsd:sequence>
         <xsd:element ref="WC:ReportIdentifier"/>[..]                3
         <xsd:element ref="WC:ReportName" minOccurs="1"/>[..]
         <xsd:element ref="WC:ReportingUnit" maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>                                                1
   <xsd:complexType name="ReportListDataType">
      <xsd:sequence>
         <xsd:element ref="WC:Report" minOccurs="0" maxOccurs="10"/>
      </xsd:sequence>
   </xsd:complexType>                                                    2
   <xsd:element name="ReportList" type="WC:ReportListDataType"></xsd:element>
</xsd:schema>
```

Fig. 2. The adapted WSWC report XML Schema

The modifications have the purpose to summarize information in only one report that otherwise would be spread over multiple small reports. This is for example possible by combining different reports. In general, a new complex type ("ReportListDataType") and a new element declaration ("ReportList") are added. Furthermore, the name of a report is no longer optional, the minimum occurrence is changed to one (i.e. it is obligatory). The question is how can these modifications be described formally? Before presenting one possibility (i.e. ELaX), some background information and notations are presented in the following chapter.

3 Technical Background

In this section we present a common notation used in the remainder of this paper. At first, we will shortly introduce the *abstract data model* and *element information item* of XML Schema, before further details concerning different modeling styles are given.

The XML Schema *abstract data model* consists of different components or node types[1]. Basically, these are: type definition components (simple and complex types), declaration components (elements and attributes), model group components, constraint components, group definition components and annotation components [4]. Additionally, the *element information item* serves as an XML representation of these components. The *element information item* defines which content and attributes can be used in an XML Schema. The following table 1 gives an overview about the most important components and their concrete representation.

Abstract Data Model	**Element Information Item**
declarations	<element>, <attribute>
group-definitions	<attributeGroup>
model-groups	<all>, <choice>, <sequence>, <any>, <anyAttribute>
type-definitions	<simpleType>, <complexType>
N.N.	<include>, <import>, <redefine>, <overwrite>
annotations	<annotation>
constraints	<key>, <unique>, <keyref>, <assert>, <assertion>
N.N.	<schema>

Table 1. Abstract Data Model and XML representation

The <include>, <import>, <redefine> and <overwrite> items are not explicitly given in the *abstract data model* (N.N. - Not Named), but they are important components for embedding externally defined XML Schemas (esp. element declarations, attribute declarations and type definitions). In the remaining parts of the paper, we will summarize them under the node type "module". The <schema> item "is the document (root) element of any W3C XML Schema. It's both a container for all the declarations and definitions of the schema and a place holder for a number of default values expressed as attributes" [17].

Analyzing the possibilities of specifying declarations and definitions leads to four different modeling styles of XML Schema [13]. These modeling styles mainly influence the re-usability of element declarations or defined data types but also the flexibility of an XML Schema in general. The scope of element and attribute

[1] An XML Schema can be visualized as a directed graph with different nodes (components); an edge represents the hierarchy between two nodes

declarations as well as the scope of type definitions is global iff the corresponding node is specified as a child of the <schema> item and can be referenced (e.g. by knowing the name and namespace). In contrast, locally specified nodes are not directly defined under the <schema> item, therefore the re-usability is low. Table 2 summarizes the modeling styles according to [13]:

	Scope	**Russian Doll**	**Salami Slice**	**Venetian Blind**	**Garden of Eden**
element and attribute declaration	local	x		x	
	global		x		x
type definition	local	x	x		
	global			x	x

Table 2. Overview of XSD modeling styles according to [13]

An XML Schema in the *Garden of Eden* style just contains global declarations and definitions. If the requirements against exchanged information change and the underlying schema has to be adapted then this modeling style is the most suitable one. That is, because all components can be easily identified by knowing the *QNAME* (qualified name). Furthermore, the position of components within an XML Schema is obvious. A qualified name is a colon separated string of the target namespace of the XML Schema followed by the name of the declaration or definition. The name of a declaration and definition is a string of the data type *NCNAME* (non-colonized name), a string without colons. Due to the characteristics of the *Garden of Eden* style, it is the basic modeling style which is considered in this paper, a transformation between different styles is easily possible.[2]

4 XML Schema Transformation Language

In order to handle modifications on an XML Schema and to express these modifications formally, an adequate transformation language is absolutely essential. Therefore, we developed ELaX (Evolution Language for XML-Schema) which lets the user describe modifications in a simple, easily-understandable and explicit manner. The following criteria were important through the development process, parts were already mentioned above:

1. Consideration of the underlying data model (i.e. the *abstract data model* and *element information item* of the XML Schema definition)
2. Adequate and complete realization of the common operations ADD, UPDATE, DELETE

[2] A student thesis to address the issue of converting different modeling styles into each other is in progress at our professorship; this is not covered in this paper.

3. Definition of an descriptive and readable interface for creating, changing and deleting XML Schema
4. Intuitive and simple syntax of operation steps

The *abstract data model* and the *element information item* consist of different node types, which have to be adapted. These node types are substantially summarized: elements and attributes (declarations), content models (model group components), data types (simple and complex type definitions), modules (externally defined schema, e.g. included, redefined or imported schema), annotations, constraints and the schema itself. On these node types the operations ADD, DELETE and UPDATE have to be executed; the first ELaX statements are in an EBNF (Extended Backus-Naur Form) like notation:

$$elax ::= ((< add > | < delete > | < update >) \text{ ";" }) + \; ; \tag{1}$$

$$\begin{aligned} add ::= \text{"add"} \; (&< addannotation > | < addattributegroup > \\ &| < addgroup > | < addst > | < addct > | < addelement > \\ &| < addmodule > | < addconstraint >) \; ; \end{aligned} \tag{2}$$

$$\begin{aligned} delete ::= \text{"delete"} \; (&< delannotation > | < delattributegroup > \\ &| < delgroup > | < delst > | < delct > | < delelement > \\ &| < delmodule > | < delconstraint >) \; ; \end{aligned} \tag{3}$$

$$\begin{aligned} update ::= \text{"update"} \; (&< updannotation > | < updattributegroup > \\ &| < updgroup > | < updst > | < updct > | < updelement > \\ &| < updmodule > | < updconstraint > | < updschema >) \; ; \end{aligned} \tag{4}$$

An ELaX statement always starts with "add", "delete" or "update" followed by one of the alternative components for modifying the different node types. Every component of rule (1) can optionally be repeated one or more times (i.e. "+"), consequently an encapsulation or ordered sequence of operations is possible. The operations are separated by ";". By using the rules (1), (2), (3) and (4), complex modifications of an XML Schema can be expressed formally. In the following subsections the statements for adding (<addelement>), deleting (<delelement>) and updating (<updelement>) elements are presented in detail.[3]

4.1 Adding elements

According to the *Garden of Eden* modeling style, elements are either defined as element declarations in the global scope of an XML Schema or as references to such declarations. Furthermore, it is possible to define wildcards, which provide "for validation of attribute and element information items dependent on their

[3] The statements not presented (i.e. the whole update language ELaX) are available at: www.ls-dbis.de/elax

namespace name, but independently of their local name" [4]. Wildcards are one reason for the high extensibility of XML. The following statements realize the add operation for elements:

$$addelement ::= \ < addelementdef > | < addelementref > \\ | < addelementwildard > \ ; \tag{5}$$

$$addelementdef ::= \ "element" \ "name" \ \text{ncname} \ "type" \ \text{qname} \\ (("default"|"fixed") \ \text{string} \ )? \ ("final" \ ("\#all"|"restriction" \\ |"extension"))? \ ("nillable" \ ("true"|"false"))? \ ("id" \ \text{id} \ )? \\ ("form" \ ("qualified"|"unqualified"))? \ ; \tag{6}$$

$$addelementref ::= \ "element" \ "ref" \ \text{qname} \ ("minoccurs" \ \text{int} \ )? \\ ("maxoccurs" \ \text{int} \ )? \ ("id" \ \text{id} \ )? < position > \ ; \tag{7}$$

$$addelementwildard ::= \ "any" \ ("not" \ (\ \text{qname} \ | \ ("\#\#definedsibling" \\ |"\#\#defined")) +)? \ ("namespace" \ ("\#\#any"|"\#\#other"| \ (\ \text{anyuri} \\ |"\#\#local"|"\#\#targetnamespace")+) \ ("not" \ (\ \text{anyuri} \ | \ ("\#\#local" \\ |"\#\#targetnamespace"))+)?)? \ ("processcontent" \ ("lax"|"skip" \\ |"strict"))? \ ("minoccurs" \ \text{int} \ )? \ ("maxoccurs" \ \text{int} \ )? \ ("id" \ \text{id} \ )? \\ "in" < locator > \ ; \tag{8}$$

Before going into detail, further components are necessary to localize or identify elements and node types in general. It is possible to localize node types in a content model under consideration of the node neighborhood with statement (9) and to identify a node type itself by using an absolute addressing (11). An absolute addressing always begins at the "document (root) element" whereas a relative address starts at the current selected node type.

$$position ::= ("after"|"before"| \ ("as"("first"|"last") \ "into") \ |"in") \\ < locator > \ ; \tag{9}$$

$$locator ::= < xpathexpr > | \ \text{emxid} \ ; \tag{10}$$

$$xpathexpr ::= ("/"("." \ | \ ("node()" \ | \ ("node()[@name='"\text{ncname}"']")) \\ ("[" \ \text{int} \ "]")? \ ) \ ) + \ ; \tag{11}$$

An element reference statement (7) starts with "element ref", followed by the qualified name of the referenced element declaration (qname) and other, optional attributes for the frequency of occurrence ("minoccurs", "maxoccurs") or an XML Schema id ("id"). An element reference can be added "after", "before", "as first into", "as last into" or "in" the model-group node type (i.e. sequence, choice, all) with consideration of the node neighborhood and using statement (9). The identification of nodes is possible with a unique identifier of our conceptual model

(emxid)[4]. Alternatively, a subset of XPath expressions (<xpathexpr>) can be used to create an absolute path. The subset of XPath contains the navigation steps child ("child::node()" resp. "/"), self ("self::node()" resp. "."), and also the general navigation without a predicate ("node()"), with a specified name in a predicate ("node()[..]") or by using the position ("[int]"). The position is always needed if an XPath expression returns a set of nodes instead of a single node - otherwise the identified node would not be unique (e.g. same named nodes in a content model sequence). The given subset is sufficient for the simple localization or identification of every node type in the *Garden of Eden* style. This is possible, because every declaration or definition has a global scope and the maximum nesting depth is limited.

In the *Garden of Eden* style, element declarations (6) are added under the *element information item* of the schema itself. Furthermore, the order and neighborhood of element declaration nodes do not influence the XML Schema, so no special positioning is necessary.

Element wildcards are added at the end of a content model node (i.e. sequence or choice) with statement (8) according to the XML Schema specification. The identification of the parent node is sufficient for adding wildcards.

4.2 Deleting elements

It is possible to add element declarations, references and wildcards with statement (2). Following, the removal of such parts is described. Compared to the add operation, deleting an element basically just requires some information of identification. The qualified name in general and in the case of references and wildcards also the position of an element is sufficient. The following statements realize the delete operation for elements:

$$\begin{aligned} delelement ::= &< delelementdef > \mid < delelementref > \\ &\mid < delelementwildcard > \; ; \end{aligned} \tag{12}$$

$$delelementdef ::= \text{"}element\text{" "}name\text{" qname} \; ; \tag{13}$$

$$\begin{aligned} delelementref ::= &\text{"}element\text{" "}ref\text{" qname} \\ &(\text{"}at\text{"} < locator > \mid < refposition >) \; ; \end{aligned} \tag{14}$$

$$delelementwildcard ::= \text{"}any\text{" "}at\text{"} < locator > \; ; \tag{15}$$

$$\begin{aligned} refposition ::= &((\text{"}first\text{"}\mid\text{"}last\text{"}\mid\text{"}all\text{"}\mid (\text{"}at\text{" "}position\text{" int })) \\ &\text{"}in\text{"} < xpathexpr >) \mid \text{emxid} \; ; \end{aligned} \tag{16}$$

The element reference statement (14) starts with "element ref", followed by the qualified name and information about the locator (introduced in section 4.1, statement (10)) or about the position of the reference (<refposition>). The reference position statement (16) enables the localization of one reference if

[4] Our conceptual model is EMX (Entity Model for XML Schema)[11], in which every node of a model has its own, global identifier; see also section 6

more than one is given in the same group-model node type. With this statement the "first", the "last", "all" of them or a reference at a specific position ("at position") can be deleted. If the unique identifier of the conceptual model is known, the emxid can be used instead of the XPath expression.

The element declaration is uniquely identified with its qualified name, because it is located directly under the *element information item* of the schema itself.

An element wildcard needs the localization of the parent node, according to the adding statement (8). More than one wildcard is not valid according to the XML Schema specification.

4.3 Updating elements

Updating elements is implemented by rule (4). Basically, all added elements with their given information can be updated afterwards. Also, adding new information to an existing element is considered, so the specification of an update of elements is similar to the adding of them. In general, the qualified name, the new values or additional information and the position of an element when dealing with references or wildcards are needed. The following statements realize the update operation for elements:

$$\begin{aligned} updelement ::=& < updelementdef > \mid < updelementref > \\ & \mid < updelementwildcard > \ ; \end{aligned} \tag{17}$$

$$\begin{aligned} &updelementdef ::= \ \text{"}element\text{"} \ \text{"}name\text{"} \ \text{qname} \ \text{"}change\text{"} \\ &\quad (\text{"}name\text{"} \ \text{ncname} \ )? \ (\text{"}type\text{"} \ \text{qname} \ )? \ ((\text{"}default\text{"}|\text{"}fixed\text{"} \\ &\quad \text{string} \ )? \ (\text{"}final\text{"} \ (\text{"}\#all\text{"}|\text{"}restriction\text{"}|\text{"}extension\text{"}))? \\ &\quad (\text{"}nillable\text{"} \ (\text{"}true\text{"}|\text{"}false\text{"}))? \ (\text{"}id\text{"} \ \text{id} \ )? \\ &\quad (\text{"}form\text{"} \ (\text{"}qualified\text{"}|\text{"}unqualified\text{"}))? \ ; \end{aligned} \tag{18}$$

$$\begin{aligned} &updelementref ::= \ \text{"}element\text{"} \ \text{"}ref\text{"} \ \text{qname} \ ((\text{"}at\text{"} < locator >) \\ &\quad \mid < refposition >) \ (\text{"}change\text{"} \ (\text{"}ref\text{"} \ \text{qname} \ )? \\ &\quad (\text{"}minoccurs\text{"} \ \text{int} \ )? \ (\text{"}maxoccurs\text{"} \ \text{int} \ )? \ (\text{"}id\text{"} \ \text{id} \ )? \ )? \\ &\quad (\text{"}move\text{"} \ \text{"}to\text{"} < position >)? \ ; \end{aligned} \tag{19}$$

$$\begin{aligned} &updelementwildcard ::= \ \text{"}any\text{"} \ \text{"}at\text{"} < locator > \text{"}change\text{"} \\ &\quad (\text{"}not\text{"} \ (\ \text{qname} \mid (\text{"}\#\#defined\text{"}|\text{"}\#\#definedsibling\text{"}))+)? \\ &\quad (\text{"}namespace\text{"} \ (\text{"}\#\#any\text{"}|\text{"}\#\#other\text{"}| \ (\text{"}\#\#local\text{"}| \ \text{anyuri} \\ &\quad |\text{"}\#\#targetnamespace\text{"})+) \ (\text{"}not\text{"} \ (\ \text{anyuri} \ |(\text{"}\#\#targetnamespace\text{"} \\ &\quad |\text{"}\#\#local\text{"}))+)?)? \ (\text{"}processcontent\text{"} \ (\text{"}lax\text{"}|\text{"}skip\text{"}|\text{"}strict\text{"}))? \\ &\quad (\text{"}minoccurs\text{"} \ \text{int} \ )? \ (\text{"}maxoccurs\text{"} \ \text{int} \ )? \ (\text{"}id\text{"} \ \text{id} \ )? \ ; \end{aligned} \tag{20}$$

Element references are adapted with statement (19). Starting with "element ref", the qualified name and information about the position element references

can be updated. The newly given or changed information are specified after "change". This information comprises a list of tuples of an identifier and the corresponding value, they are always optional (i.e. "?"). For example the minimal occurrence can be changed to the value 5 using "*[..] change minoccurs 5*". Furthermore, it is possible to move an element reference, that is why the "move to" component was inserted at the end of the statement (19). The move operation is a short form for deleting and inserting an element reference completely. The information regarding the localization of references was mentioned above (<locator> and <position> in section 4.1, <refposition> in section 4.2).

Element declarations and wildcards are adapted with statements (18) and (20). In both cases the corresponding statement is a sequence of an element identifier (qualified name or location), followed by "change" and completed with optional tuples of identifier and value. After introducing some basic statements of ELaX, we want to pursue with the running example (section 2) and use those statements for adapting the given XML Schema.

5 Example

In section 2 an XML Schema for exchange reports of the WSWC (Western States Water Council) was introduced. The basic schema of figure 1 should be adapted to the XML Schema of figure 2. The main idea of the adaptions was the grouping of ten reports in a list in order to summarize information in one report that otherwise would be spread over multiple small reports. In general, three different steps are necessary to modify the old schema (the steps are visualized in labelled rectangles in figure 2):

1. Insert a new type, which contains up to ten reports
2. Insert a new element, which has the new introduced type of *step 1*
3. Update the "ReportDataType" type so that the report name is obligatory

Following, the necessary ELaX operations for the above mentioned steps are described, the syntax of statements which are not directly introduced in section 4 are mentioned in footnotes[5]. Furthermore, the replaced values of the data types

[5] Syntax of components <addct> and <addgroup> of rule (2):

addct ::= "*complextype*" "*name*" ncname ("*mixed*" ("*true*"|"*false*"))?
("*final*" ("*#all*"|"*restriction*"|"*extension*"))? ("*mode*" ("*extension_cc*"|
"*extension_sc*"|"*restriction_cc*"|"*restriction_sc*") "*with*" "*base*" qname)?
("*id*" id)? ("*defaultattributesapply*" ("*true*"|"*false*"))? (< *assert* > ∗)? ;
assert ::= "*assert*" string (< *xpathdefaultnamespace* >)? ("*id*" id)? ;
xpathdefaultnamespace ::= "*xpathdefaultnamespace*" (anyuri |
("*##defaultnamespace*"|"*##targetnamespace*"|"*##local*"));
addgroup ::= "*group mode*"("*sequence*"|"*choice*"|"*all*")("*with*" < *groupdefault* >)?
("*minoccurs*" int)? ("*maxoccurs*" int)? ("*id*" id)? "*in*" < *locator* > ;
groupdefault ::= "*first*" | "*last*" | int ;

are boldfaced in every following operation, e.g. the values of QNAME, NCNAME or XPath statements.

Step 1: First of all, a new type has to be inserted. This is a complex type, because the new type has a complex content model which contains different elements (the reports of the "old" XML Schema). The new complex type gets the NCNAME "ReportListDataType". After specifying the complex type, a group-model node type is inserted as a child of the new complex type. In our example, this is defined as a sequence. The last operation within the first step is performed by inserting an element reference into the sequence. The necessary element declaration is "Report", this element can be referenced by the QNAME "WC:Report". This approach is possible because the XML Schema is modeled in *Garden of Eden* style and moreover, because the target namespace is known (i.e. "WC"). Information about the occurrence is also given, up to ten reports can be collected in the report list. According to that the maximum occurrence value is fixed to ten, the minimum occurrence is fixed to zero (i.e. optional). The following ELaX operations have to be executed, the sequence of applied rules are listed below every operation:

add complextype name **ReportListDataType** *;* (21)
Sequence of rules: (1), (2), addct[note 5 (page up)]

add group mode sequence in
/node()/node()[@name='ReportListDataType'] *;* (22)
Sequence of rules: (1), (2), addgroup[note 5 (page up)], (10), (11), (11)

add element ref **WC:Report** *minoccurs* **0** *maxoccurs* **10** *in*
/node()/node()[@name='ReportListDataType']/node() *;* (23)
Sequence of rules: (1), (2), (5), (7), (9), (10), (11), (11), (11)

The correct order of the operations (21), (22) and (23) is specified in the *element information item* of XML Schema and represents the result of the implicitly given relationships of node types. For example it is necessary to define a complex type (type-definition node type) before a model-group node type can be inserted into it. The XSD corresponding execution of operations can easily be guaranteed by using simple rules; this approach is not covered in this paper.

Step 2: After specifying a new complex type in *step 1*, a new element declaration with this complex type has to be defined. The NCNAME of this new element is "ReportList", the qualified name of the complex type is "WC:ReportList-DataType". Further information are not required, because element declarations are inserted into the global scope of the XML Schema and other attributes (e.g. default values) are not given.

add element name **ReportList** *type* **WC:ReportListDataType** *;* (24)
Sequence of rules: (1), (2), (5), (6)

Step 3: The last adaption of the XML Schema in figure 1 contains changing the occurrence of the report name. In our example it is changed from "minOccurs = '0'" to "minOccurs = '1'". Consequently, the name is no longer optional but obligatory. The corresponding operation is applied as follows:

update element ref **WC:ReportName** *at* **/node()/node()**
[@name='ReportDataType']/node()[2] *change minoccurs* **1** *;* (25)

Sequence of rules: (1), (4), part I (19), (10), (11), (11), (11), part II (19)

The XML Schema of figure 1 is adapted with operations (21), (22), (23), (24) and (25). This simple example illustrates how modifications can be described formally by using ELaX statements introduced in section 4. In general, three different steps were necessary to fulfill the new requirements against the XML Schema of exchanged reports according to section 2. More complex examples are easy to construct, but due to the limitation aspects this is not covered in this paper.

6 Practical Use of ELaX

The transformation language ELaX was specified for dealing with XML Schema modifications. It is useful to describe and formulate different adaptions of an XML Schema. In general, this reflects all add, delete and update operations which are possible considering the underlying base model (XSD). In section 1, the co-evolution of XML documents was already mentioned. If an XML Schema is adapted then possibly the XML documents have to be adapted as well to recover their validity against the new XML Schema. The question is how can these adaptions of XML Schema be used to derive all necessary XML document transformations automatically?

At the University of Rostock a research prototype named CodeX (Conceptual design and evolution for XML Schema) was developed for dealing with co-evolution. The idea behind CodeX is simple and straightforward at the same time: A given XML Schema is transformed to the specifically developed conceptual model (EMX - Entity Model for XML Schema [15]). With the help of this simplified model, the desired modifications are defined and logged (i.e. user interaction) and then used to automatically create transformation steps for adapting the XML documents (by using XSLT - Extensible Stylesheet Language Transformations [1]). The mapping between EMX and XSD is unique, so it is possible to describe modifications not only on the EMX but also on the XSD. ELaX is useful to unify the collected information and additionally provides an interface for dealing directly with an XML Schema. The picture 3 illustrates the component model of CodeX, primarily published in [14] but now extended with the ELaX interface.

The component model illustrates the different parts for dealing with the co-evolution. The main parts consist of an import and export component for

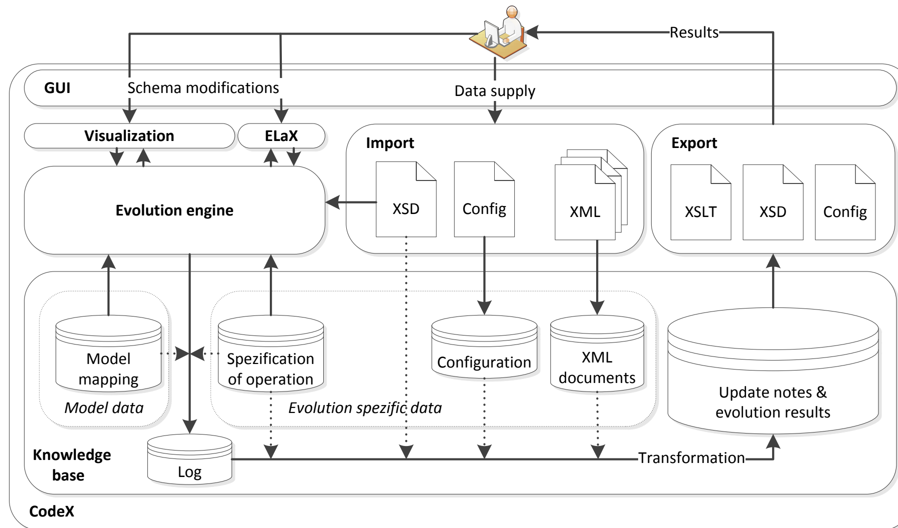

Fig. 3. The integration of ELaX into prototype CodeX [10]

collecting and providing data of an user (XML Schemas, configuration files, XML document collections, XSLT files), a knowledge base for storing information (model data, evolution specific data and co-evolution results) and especially the logged ELaX statements ("Log"). Moreover, the CodeX prototype also provides a graphical user interface ("GUI"), a visualization component for the conceptual model and an evolution engine, in which the transformations are derived. The new ELaX interface for modifying imported XML Schemas communicates directly with the evolution engine.

Because the ELaX interface is implemented for the user-driven adaption of XML Schema and ELaX is used for internal logging of modification steps, the criteria for this transformation language (listed in section 4) are more important than they seem to be at the beginning of this article: consideration of the base model; adequate and complete realization of common operations; descriptive, readable and without ignoring the other criteria a simple and intuitive syntax.

7 State of the Art

Regarding other transformation languages, there are some which may possibly be used instead of ELaX. The most commonly to mention are XQuery and XSLT.

XSLT is "a language for transforming XML documents into other XML documents" [1] so it can also deal with XML Schema (an XML Schema is basically an XML document). XSLT works with templates and matching rules for identifying structures and creates a new document (target) based on the original, old document (source). That is why, every node type of an XML Schema, e.g. all type definitions and declarations, have to be copied from the source to the target

although the nodes are not updated or changed. XSLT is very complex and difficult to understand, so the use and also the understanding of its results implies a huge overhead. This language is neither suitable for describing modifications nor for unifying the internal collected information within our context.

XQuery [3] as a query language for different XML data sources and especially the extension of it through the update facility [5] "provide expressions that can be used to make persistent changes" to instances of the *abstract data model* (see section 3). In general, the different introduced operations are integrated in the return clause of an XQuery statement, so it is possible to insert, delete, replace, rename or transform parts of an XML Schema. But by using all features of XQuery update, modifications can be produced that lead to non-regular XML Schema [16]. This is why restrictions are required (e.g. avoiding loops and conditional statements [16]). However, by focussing on which parts of a complex update language (i.e. XQuery update) are suitable or which parts lead to non-valid XML Schema, again an unintentional overhead is produced. Furthermore, it is questionable whether XQuery statements are suitable for the unified logging of modification steps.

In section 6 the integration of ELaX into our prototype was illustrated. Examples of prototypes developed by other working groups dealing with the evolution of XML Schema include for example X-Evolution [9], EXup [7], the GEA Framework [8] and XCase [12].[6] To our knowledge, XSchemaUpdate (used in X-Evolution and EXup) is the only XML Schema modification language which is comparable to ELaX. XSchemaUpdate was also developed to modify given XML Schema per interface. However, ELaX is closer to the base-model. ELaX considers wildcards, constraints and attributes, which are explicitly allowed in a node type considering the *element information item* of XML Schema. Moreover, it distinguishes between element declarations and references (see rules (6) and (7) in section 4.1) amongst others, so more fine-grained operations are possible, which simplifies the analyzing of modification steps. Importantly, the other prototypes do not contain such an XML Schema modification language like ELaX, even though it is a good and practicable way for direct modifications of a given XML Schema.

8 Conclusion

Valid XML documents need an XML Schema which specifies the possibilities and usage of declarations, definitions and of structures in general. In a heterogeneous and dynamic environment (e.g. the information exchange scenario mentioned above), also "old" and longtime used XML Schema have to be modified to meet new requirements and to be up-to-date. Modifications of XML Schema documents urgently need a description and a formalism in order to be traceable. Therefore, we developed the new language ELaX (Evolution Language for XML-Schema).

[6] A detailed list containing also e.g. the State of the Art of databases are listed in [14]

ELaX is a base-model-oriented transformation language, which can be used to modify a given XML Schema. These modifications are in general add, delete and update operations on the node types of XML Schema which are specified in the *abstract data model* and implemented in the *element information item*. ELaX has a simple, intuitive and easy-understandable syntax, but nevertheless it is powerful enough to describe more complex transformations by combining the given operations. Moreover, it can be used to log modifications for the adaptions of XML documents associated with a given XML Schema, which represents an essential prerequisite for the co-evolution.

We are confident, that ELaX is user-friendly and easy to use for users with some basic knowledge of XML Schema. In general, every ELaX operation is something like: operation, node type, node identification or localization followed by a list of optional tuples of information (attribute-value pairs).

One remaining step is the implementation of the language and the integration into our research prototype CodeX (Conceptual design and evolution for XML Schema), a student thesis is in progress at our professorship to deal with these remaining steps. Afterwards, a final evaluation is planned.

References

1. XSL Transformations (XSLT) Version 2.0. http://www.w3.org/TR/2007/REC-xslt20-20070123/, January 2007. Online; accessed 26-March-2013.
2. Extensible Markup Language (XML) 1.0 (Fifth Edition). http://www.w3.org/TR/2008/REC-xml-20081126/, November 2008. Online; accessed 26-March-2013.
3. XQuery 1.0: An XML Query Language (Second Edition). http://www.w3.org/TR/2010/REC-xquery-20101214/, December 2010. Online; accessed 26-March-2013.
4. XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). http://www.w3.org/TR/2010/REC-xpath-datamodel-20101214/, December 2010. Online; accessed 26-March-2013.
5. XQuery Update Facility 1.0. http://www.w3.org/TR/2011/REC-xquery-update-10-20110317/, March 2011. Online; accessed 26-March-2013.
6. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. http://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/, April 2012. Online; accessed 26-March-2013.
7. Federico Cavalieri. EXup: an engine for the evolution of XML schemas and associated documents. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 21:1–21:10, New York, NY, USA, 2010. ACM.
8. Eladio Domínguez, Jorge Lloret, Beatriz Pérez, Áurea Rodríguez, Angel Luis Rubio, and María Antonia Zapata. Evolution of XML schemas and documents from stereotyped UML class models: A traceable approach. *Information & Software Technology*, 53(1):34–50, 2011.
9. Giovanna Guerrini and Marco Mesiti. X-Evolution: A Comprehensive Approach for XML Schema Evolution. In *DEXA Workshops*, pages 251–255, 2008.
10. Meike Klettke. Conceptual XML Schema Evolution - the CoDEX Approach for Design and Redesign. In *BTW Workshops*, pages 53–63, 2007.

11. Meike Klettke. *Modellierung, Bewertung und Evolution von XML-Dokumentkollektionen*. Habilitation, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2007.
12. Jakub Klímek, Lukás Kopenec, Pavel Loupal, and Jakub Malý. XCase - A Tool for Conceptual XML Data Modeling. In *ADBIS (Workshops)*, pages 96–103, 2009.
13. Eve Maler. Schema design rules for ubl...and maybe for you. In *XML 2002 Proceedings by deepX*, 2002.
14. Thomas Nösinger, Meike Klettke, and Andreas Heuer. Evolution von XML-Schemata auf konzeptioneller Ebene - Übersicht: Der CodeX-Ansatz zur Lösung des Gültigkeitsproblems. In *Grundlagen von Datenbanken*, pages 29–34, 2012.
15. Thomas Nösinger, Meike Klettke, and Andreas Heuer. A Conceptual Model for the XML Schema Evolution - Overview: Storing, Base-Model-Mapping and Visualization. In *Grundlagen von Datenbanken*, 2013.
16. Alessandro Solimando, Giorgio Delzanno, and Giovanna Guerrini. Static Analysis of XML Document Adaptations. In Silvana Castano, Panos Vassiliadis, LaksV. Lakshmanan, and MongLi Lee, editors, *Advances in Conceptual Modeling*, volume 7518 of *Lecture Notes in Computer Science*, pages 57–66. Springer, 2012.
17. Eric van der Vlist. *XML Schema*. O'Reilly & Associates, Inc., 2002.