Optimization of Sequences of XML Schema Modifications - The ROfEL Approach

Thomas Nösinger, Meike Klettke, Andreas Heuer
Database Research Group
University of Rostock, Germany
(tn, meike, ah)@informatik.uni-rostock.de

ABSTRACT

The transformation language ELaX (Evolution Language for XML-Schema [16]) is a domain-specific language for modifying existing XML Schemas. ELaX was developed to express complex modifications by using add, delete and update statements. Additionally, it is used to consistently log all change operations specified by a user. In this paper we present the rule-based optimization algorithm ROfEL (Rule-based Optimizer for ELaX) for reducing the number of logged operations by identifying and removing unnecessary, redundant and also invalid modifications. This is an essential prerequisite for the co-evolution of XML Schemas and corresponding XML documents.

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 are valid it is necessary to introduce a document description, which contains information about allowed structures, constraints, data types and so on. XML Schema [4] is one commonly used standard for dealing with this problem. After using an XML Schema a period of time, the requirements can change; for example if additional elements are needed, data types change or integrity constraints are introduced. This may result in the adaptation of the XML Schema definition.

In [16] we presented the transformation language ELaX (Evolution Language for XML-Schema) to describe and formulate these XML Schema modifications. Furthermore, we mentioned briefly that ELaX is also useful to log information about modifications consistently, an essential prerequisite for the co-evolution process of XML Schema and corresponding XML documents [14].

One problem of storing information over a long period of time is, that there can be different unnecessary or redundant modifications. Consider modifications which firstly add an element and shortly afterwards delete the same element. In the overall context of an efficient realization of modification steps, such operations have to be removed. Further issues are incorrect information (possibly caused by network problems), for example if the same element is deleted twice or the order of modifications is invalid (e.g. update before add).

The new rule-based optimizer for ELaX (ROfEL - Rule-based Optimizer for ELaX) had been developed for solving the above mentioned problems. With ROfEL it is possible to identify unnecessary or redundant operations by using different straightforward optimization rules. Furthermore, the underlying algorithm is capable to correct invalid modification steps. All in all, ROfEL could reduce the number of modification steps by removing or even correcting the logged ELaX operations.

This paper is organized as follows. **Section 2** gives the necessary background of XML Schema, ELaX and corresponding concepts. **Section 3** and **section 4** present our approach, by first specifying our ruled-based algorithm ROfEL and then showing how our approach can be applied for an example. Related work is shown in **section 5**. Finally, in **section 6** we draw our conclusions.

2. TECHNICAL BACKGROUND

In this section we present a common notation used in the remainder of this paper. At first, we will shortly introduce the XSD (XML Schema Definition [4]), before details concerning ELaX (Evolution Language for XML-Schema [16]) and the logging of ELaX are given.

The XML Schema *abstract data model* consists of different components (simple and complex type definitions, element and attribute declarations, etc.). Additionally, the *element information item* serves as an XML representation of these components and defines which content and attributes can be used in an XML Schema. The possibility of specifying declarations and definitions in a local or global scope leads to four different modeling styles [13]. One of them is the *Garden of Eden* style, in which all above mentioned components are globally defined. This results in a high re-usability of declarations and defined data types and influences the flexibility of an XML Schema in general.

The transformation language ELaX[1] was developed to handle modifications on an XML Schema and to express such modifications formally. The *abstract data model*, *element information item* and *Garden of Eden* style were important through the development process and influence

In: G. Specht, H. Gamper, F. Klan (eds.): Proceedings of the 26^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 21.10.2014 - 24.10.2014, Bozen, Italy, published at http://ceur-ws.org.

[1]The whole transformation language ELaX is available at: `www.ls-dbis.de/elax`

the EBNF (Extended Backus-Naur Form) like notation of ELaX.

An ELaX statement always starts with "add", "delete" or "update" followed by one of the alternative components (simple type, element declaration, etc.), an identifier of the current component and completed with optional tuples of attributes and values (examples follow on, e.g. see figure 1). The identifier is a unique *EID* (emxid)[2], a QNAME (qualified name) or a subset of XPath expressions. In the remaining parts we will use the *EID* as the identifier, but a transformation would be easily possible.

ELaX statements are logged for further analyses and also as a prerequisite for the rule-base optimizer (see section 3). Figure 1 illustrates the relational schema of the log. The

file-ID	time	EID	op-Type	msg-Type	content
1	1	1	add	0	add element name 'name' type 'xs:decimal' id 'EID1' ;
1	2	1	upd	0	update element name 'name' change type 'xs:string' ;
1	3	2	add	0	add element name 'count' type 'xs:decimal' id 'EID2' ;
...	...	...	...	...	...

Figure 1: Schema with relation for logging ELaX

chosen values are simple ones (especially the length). The attributes *file-ID* and *time* are the composite key for the logging relation, the *EID* represents the unique identifier for a component of the XSD. The *op-Type* is a short form for add, delete (del) or update (upd) operations, the *msg-Type* is for the different message types (ELaX (0), etc.). Lastly, the *content* contains the logged ELaX statements. The *file-ID* and *msg-Type* are management information, which are not covered in this paper.

3. RULE-BASED OPTIMIZER

The algorithm ROfEL (Rule-based Optimizer for ELaX) was developed to reduce the number of logged ELaX operations. This is possible by combining given operations and/or removing unnecessary or even redundant operations. Furthermore, the algorithm could identify invalid operations in a given log and correct these to a certain degree.

ROfEL is a rule-based algorithm. Provided that a log of ELaX operations is given (see section 2), the following rules are essential to reduce the number of operations. In compliance with ELaX these operations are delete (**del**), **add** or update (**upd**). If a certain distinction is not necessary a general operation (**op**) or variable (_) are used, **empty** denotes a not given operation. Additionally, the rules are classified by their purpose to handle redundant (**R**), unnecessary (**U**) or invalid (**I**) operations. ROfEL stops (**S**) if no other rules are applicable, for example no other operation with the same EID is given.

$$\text{S: } empty \rightarrow \textbf{\textit{op(EID)}} \Rightarrow op(EID) \tag{1}$$

// ↓ most recent operation: delete (**del**) ↓

$$\text{R: } del(EID) \rightarrow \textbf{\textit{del(EID)}} \Rightarrow del(EID) \tag{2}$$

$$\text{U: } add(EID, content) \rightarrow \textbf{\textit{del(EID)}} \Rightarrow empty \tag{3}$$

$$\text{U: } upd(EID, content) \rightarrow \textbf{\textit{del(EID)}} \Rightarrow del(EID) \tag{4}$$

with $time(del(EID)) := \textbf{TIME}(del(EID),\ upd(EID,\ content))$

[2]Our conceptual model is EMX (Entity Model for XML Schema [15]), in which every component of a model has its own, global identifier: *EID*

// ↓ most recent operation: **add** ↓

$$\text{U: } op(EID) \rightarrow del(EID) \rightarrow \textbf{\textit{add(EID, content)}} \Rightarrow op(EID) \rightarrow \textbf{\textit{add(EID, content)}} \tag{5}$$

$$\text{I: } add(EID, \_) \rightarrow \textbf{\textit{add(EID, content)}} \Rightarrow add(EID, content) \tag{6}$$

$$\text{I: } upd(EID, \_) \rightarrow \textbf{\textit{add(EID, content)}} \Rightarrow upd(EID, content) \tag{7}$$

// ↓ most recent operation: update (**upd**) ↓

$$\text{I: } op(EID) \rightarrow del(EID) \rightarrow \textbf{\textit{upd(EID, content)}} \Rightarrow op(EID) \rightarrow \textbf{\textit{upd(EID, content)}} \tag{8}$$

$$\text{U: } add(EID, content) \rightarrow \textbf{\textit{upd(EID, content)}} \Rightarrow add(EID, content) \tag{9}$$

$$\text{U: } add(EID, content) \rightarrow \textbf{\textit{upd(EID, content')}} \Rightarrow add(EID, \textbf{MERGE}(content', content)) \tag{10}$$

$$\text{R: } upd(EID, content) \rightarrow \textbf{\textit{upd(EID, content)}} \Rightarrow upd(EID, content) \tag{11}$$

$$\text{U: } upd(EID, content) \rightarrow \textbf{\textit{upd(EID, content')}} \Rightarrow upd(EID, \textbf{MERGE}(content', content)) \tag{12}$$

The rules have to be sequentially analyzed from left to right ($\rightarrow$), whereas the left operation comes temporally before the right one (i.e., time(left) < time(right). To warrant that the operations are working on the same component, the EID of both operations is equal. If two operations exist and a rule applies to them, then the result can be found on the right side of $\Rightarrow$. The time of the result is the time of the prior (left) operation, except further investigations are explicit necessary or the time is unknown (e.g. empty).

Another point of view illustrates, that the introduced rules are complete concerning the given operations add, delete and update. Figure 2 represents an operation matrix, in which every possible combination is covered with at least one rule. On the x-axis the prior operation and on the y-axis the

operation		prior		
		add	delete	update
recent	add	(6)	(5)	(7)
	delete	(3)	(2)	(4)
	update	(9) , (10)	(8)	(11) , (12)

Figure 2: Operation matrix of rules

recent operation are given, whereas the three-valued rules (5) and (8) are minimized to the both most recent operations (e.g. without *op(EID)*). The break-even point contains the applying rule or rules (considering the possibility of merging the content, see below).

Rule (4) is one example for further investigations. If a component is deleted (*del(EID)*) but updated (*upd(EID)*) before, then it is not possible to replace the prior operation with the result (*del(EID)*) without analyzing other operations between them. The problem is: if another operation (*op(EID')*) references the deleted component (e.g. a simple type) but because of ROfEL *upd(EID)* (it is the prior operation) is replaced with *del(EID)*, then *op(EID')* would be invalid. Therefore, the function **TIME()** is used to determine the correct time of the result. The function is given in pseudocode in figure 3. **TIME()** has two input parame-

```
TIME(op, op'):
// time(op) = t; time(op') = t'; time(opx) = tx;
// op.EID == op'.EID; op.EID != opx.EID; t > t';
   begin
     if ((t > tx > t') AND
         (op.EID in opx.content))
       then return t;
     return t';
   end.
```

Figure 3: TIME() function of optimizer

```
MERGE(content, content'):
// content = (A1 = 'a1', A2 = 'a2',
//            A3 = '', A4 = 'a4');
// content' = (A1 = 'a1', A2 = '',
//            A3 = 'a3', A5 = 'a5');
   begin
     result := {};
     count := 1;
     while (count <= content.size())
       result.add(content.get(count));
       if (content.get(count) in content')
         then
           content'.remove(content.get(count));
       count := count + 1;
     count := 1;
     while (count <= content'.size())
       result.add(content'.get(count));
       count := count + 1;
// result = (A1 = 'a1', A2 = 'a2',
//           A3 = '', A4 = 'a4', A5 = 'a5');
     return result;
   end.
```

Figure 4: MERGE() function of optimizer

ters and returns a time value, dependent on the existence of an operation, which references the EID in its content. If no such operation exists, the time of the result in rule (4) would be the time of the left (*op*), otherwise of the right operation (*op'*). The lines with // are comments and contain further information, some hints or even explanations of variables.

The rules (6), (7) and (8) adapt invalid operations. For example if a component is updated but deleted before (see rule (8)), then ROfEL has to decide, which operation is valid. In this and similar cases the most recent operation is preferred, because it is more difficult (or even impossible) to check the intention of the prior operation. Consequently, in rule (8) *del(EID)* is removed and rule *op(EID)* $\rightarrow$ *upd(EID, content)* applies (*op(EID)* could be *empty*; see rule (1)).

The rules (10) and (12) removes unnecessary operations by merging the content of the involved operations. The function **MERGE()** implements this, the pseudocode is presented in figure 4. **MERGE()** has two input parameter, the content of the most recent (left) and prior (right) operation. The content is given as a sequence of attribute-value pairs (see ELaX description in section 2). The result of the function is the combination of the input, whereas the content of the most recent operation is preferred analogical to the above mentioned behaviour for **I** rules. All attribute-value pairs of the most recent operation are completely inserted into the result. Simultaneously, these attributes are removed from the content of the prior operation. At the end of the function, all remaining attributes of the prior (right) operation are inserted, before the result is returned.

All mentioned rules, as well as the functions **TIME()** and **MERGE()** are essential parts of the main function **ROFEL()**; the pseudocode is presented in figure 5. **ROFEL()**

```
ROFEL(log):
// log = ((t1,op1), (t2,op2), ...); t1 < t2 < ...;
 begin
   for (i := log.size(); i >= 2; i := i - 1)
     for (k := i - 1; k >= 1 ; k := k - 1)
       if(!(log.get(i).EID == log.get(k).EID AND
           log.get(i).time != log.get(k).time))
         then continue;
// R: del(EID) -> del(EID) => del(EID) (2)
       if (log.get(i).op-Type == 1 AND
           log.get(k).op-Type == 1)
         then
           log.remove(i);
           return ROFEL(log);
// U: upd(EID, content) -> del(EID)
//     => del(EID) (4)
       if (log.get(i).op-Type == 1 AND
           log.get(k).op-Type == 2)
         then
           temp := TIME(log.get(i), log.get(k));
           if (temp == log.get(i).time)
             then
               log.remove(k);
               return ROFEL(log);
           log.get(k) := log.get(i);
           log.remove(i);
           return ROFEL(log); [...]
// U: upd(EID,con) -> upd(EID,con')
//     => upd(EID, MERGE(con',con)) (12)
       if (log.get(i).op-Type == 2 AND
           log.get(k).op-Type == 2)
         then
           temp := MERGE(log.get(i).content,
                         log.get(k).content);
           log.get(k).content := temp;
           log.remove(i);
           return ROFEL(log);
   return log;
 end.
```

Figure 5: Main function ROFEL() of optimizer

has one input parameter, the log of ELaX operations. This log is a sequence sorted according to time, it is analyzed reversely. In general, one operation is pinned (*log.get(i)*) and compared with the next, prior operation (*log.get(k)*). If *log.get(k)* modifies the same component as *log.get(i)* (i.e., EID is equal) and the time is different, then an applying rule is searched, otherwise the next operation (*log.get(k - 1)*) is analyzed. The algorithm terminates, if the outer loop completes successfully (i.e., no further optimization is possible).

Three rules are presented in figure 5; the missing ones are skipped ([**...**]). The first rule is (2), the occurrence of redundant delete operations. According to the above men-

tioned time choosing guidelines, the most recent operation (*log.get(i)*) is removed. After this the optimizer starts again with the modified log recursively (*return ROFEL(log)*).

The second rule is (4), which removes an unnecessary update operation, because the whole referenced component will be deleted later. This rule uses the **TIME()** function of figure 3 to decide, which time should be assigned to the result. If another operation between *log.get(i)* and *log.get(k)* exists and this operation contains or references *log.get(i).EID*, then the most recent time (*log.get(i).time*) is assigned, otherwise the prior time (*log.get(k).time*).

The last rule is (12), different updates on the same component are given. The **MERGE()** function of figure 4 combines the content of both operations, before the content of the prior operation is changed and the most recent operation is removed.

After introducing detailed information about the concept of the ROfEL algorithm, we want to use it to optimize an example in the next section.

4. EXAMPLE

In the last section we specified the rule-based algorithm ROfEL (Rule-based Optimizer for ELaX), now we want to explain the use with an example: we want to store some information about a conference. We assume the XML Schema of figure 6 is given, a corresponding XML document is also presented. The XML Schema is in the *Garden of Eden* style

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="conf" type="confType" id="EID9"/>
  <xs:complexType name="confType" id="EID4">
    <xs:sequence id="EID5">
      <xs:element ref="name" id="EID6"/>
      <xs:element ref="count" id="EID7"/>
      <xs:element ref="start" id="EID8"/>
    </xs:sequence>
  </xs:complexType>
  <xs:element name="name" type="xs:string" id="EID1"/>
  <xs:element name="count" type="xs:decimal" id="EID2"/>
  <xs:element name="start" type="xs:date" id="EID3"/>
</xs:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<conf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="gvd-bsp.xsd">
  <name>gvd</name>
  <count>26</count>
  <start>2014-10-21</start>
</conf>
```

Figure 6: XML Schema with XML document

and contains four element declarations (*conf*, *name*, *count*, *start*) and one complex type definition (*confType*) with a group model (*sequence*). The group model has three element references, which reference one of the simple type element declarations mentioned above. The identification of all components is simplified by using an EID, it is visualized as a unique ID attribute (*id = "..."*).

The log of modification steps to create this XML Schema is presented in figure 7. The relational schema is reduced in comparison to figure 1. The *time*, the component *EID*, the *op-Type* and the *content* of the modification steps are given. The log contains different modification steps, which are not

time	ROfEL	EID	op-Type	content
1	10	1	add	add element name 'name' type 'xs:decimal' id 'EID1' ;
2		1	upd	update element name 'name' change type 'xs:string' ;
3		2	add	add element name 'count' type 'xs:decimal' id 'EID2' ;
4		3	add	add element name 'start' type 'xs:date' id 'EID3' ;
5		42	add	add element name 'stop' type 'xs:date' id 'EID42' ;
6	3	4	add	add complextype name 'confType' id 'EID4' ;
7		5	add	add group mode sequence id 'EID5' in 'EID4' ;
8		42	upd	update element name 'stop' change type 'xs:string' ;
9		6	add	add elementref 'name' id 'EID6' in 'EID5' ;
10	4	7	add	add elementref 'count' id 'EID7' in 'EID5' ;
11		8	add	add elementref 'start' id 'EID8' in 'EID5' ;
12		42	del	delete element name 'stop' ;
13	2	9	add	add element name 'conf' type 'confType' id 'EID9' ;
14		42	del	delete element name 'stop' ;

Figure 7: XML Schema modification log of figure 6

given in the XML Schema (EID > 9). Additionally, some entries are connected within the new introduced column *ROfEL*. The *red lines and numbers* represent the involved log entries and applying ROfEL rule.

The sorted log is analyzed reversely, the operation with time stamp 14 is pinned and compared with time entry 13. Because the modified component is not the same (EID not equal), the next operation with time 12 is taken. Both operations delete the same component (*op-Type == 1*). According to rule (2), the redundant entry 14 is removed and ROFEL restarts with the adapted log.

Rule (4) applies next, a component is updated but deleted later. This rule calls the **TIME()** function to determine, if the time of the result (i.e., *del(EID)*) should be 12 or 8. Because no operation between 12 and 8 references EID 42, the time of the result of (4) is 8. The *content* of time 8 is replaced with *delete element name 'stop';*, the *op-Type* is set to 1 and the time entry 12 is deleted.

Afterwards, ROFEL restarts again and rule (3) could be used to compare the new operation of entry 8 (original entry 12) with the operation of time 5. A component is inserted but deleted later, so all modifications on this component are unnecessary in general. Consequently, both entries are deleted and the component with EID 42 is not given in the XML Schema of figure 6.

The last applying rule is (10). An element declaration is inserted (time 1) and updated (time 2). Consequently, the **MERGE()** function is used to combine the content of both operations. According to the ELaX specification, the content of the update operation contains the attribute *type* with the value *xs:string*, whereas the add operation contains the attribute *type* with the value *xs:decimal* and *id* with *EID1*. All attribute-value pairs of the update operation are completely inserted into the output of the function (*type = "xs:string"*). Simultaneously, the attribute *type* is removed from the content of the add operation (*type = "xs:decimal"*). The remaining attributes are inserted in the output (*id = "EID1"*). Afterwards, the content of entry 1 is replaced by *add element 'name' type "xs:string" id "EID1";* and the second entry is deleted (time 2).

The modification log of figure 7 is optimized with rules (2), (4), (3) and (10). It is presented in figure 8. All in all, five of 14 entries are removed, whereas one is replaced by a combination of two others.

time	EID	op-Type	content
1	1	add	add element name 'name' type 'xs:string' id 'EID1' ;
3	2	add	add element name 'count' type 'xs:decimal' id 'EID2' ;
4	3	add	add element name 'start' type 'xs:date' id 'EID3' ;
6	4	add	add complextype name 'confType' id 'EID4' ;
7	5	add	add group mode sequence id 'EID5' in 'EID4' ;
9	6	add	add elementref 'name' id 'EID6' in 'EID5' ;
10	7	add	add elementref 'count' id 'EID7' in 'EID5' ;
11	8	add	add elementref 'start' id 'EID8' in 'EID5' ;
13	9	add	add element name 'conf' type 'confType' id 'EID9' ;

Figure 8: XML Schema modification log of figure 7 after using rules (2), (4), (3) and (10) of ROfEL

This simple example illustrates how ROfEL can reduce the number of logged operations introduced in section 3. More complex examples are easy to construct and can be solved by using the same rules and the same algorithm.

5. RELATED WORK

Comparable to the object lifecycle, we create new types or elements, use (e.g. modify, move or rename) and delete them. The common optimization rules to reduce the number of operations are originally introduced in [10] and are available in other application in the same way. In [11], rules for reducing a list of user actions (e.g. move, replace, delete, ...) are introduced. In [9], pre and postconditions of operations are used for deciding which optimizations can be executed. Additional applications can easily be found in further scientific disquisitions.

Regarding other transformation languages, the most commonly used are XQuery [3] and XSLT (Extensible Stylesheet Language Transformations [1]), there are also approaches to reduce the number of unnecessary or redundant operations. Moreover, different transformations to improve efficiency are mentioned.

In [12] different "high-level transformations to prune and merge the stream data flow graph" [12] are applied. "Such techniques not only simplify the later analyses, but most importantly, they can rewrite some queries" [12], an essential prerequisite for the efficient evaluation of XQuery over streaming data.

In [5] packages are introduced because of efficiency benefits. A package is a collection of stylesheet modules "to avoid compiling libraries repeatedly when they are used in multiple stylesheets, and to avoid holding multiple copies of the same library in memory simultaneously" [5]. Furthermore, XSLT works with templates and matching rules for identifying structures in general. If different templates could be applied, automatic or user given priorities manage which template is chosen. To avoid unexpected behaviour and improve the efficiency of analyses, it is a good practise to remove unnecessary or redundant templates.

Another XML Schema modification language is XSchema-Update [6], which is used in the co-evolution prototype EXup [7]. Especially the auto adaptation guidelines are similar to the ROfEL purpose of reducing the number of modification steps. "Automatic adaptation will insert or remove the minimum allowed number of elements for instance" [6], i.e., "a minimal set of updates will be applied to the documents" [6].

In [8] an approach is presented, which deals with four operations (insert, delete, update, move) on a tree representation of XML. It is similar to our algorithm, but we use ELaX as basis and EIDs instead of update-intensive labelling mechanisms. Moreover the distinction between property and node, the "deletion always wins" view, as well as the limitation that "reduced sequence might still be reducible" [8] are drawbacks. The optimized reduction algorithm eliminates the last drawback, but needs another complex structure, an operation hyper-graph.

6. CONCLUSION

The rule-based algorithm ROfEL (Rule-based Optimizer for ELaX) was developed to reduce the number of logged ELaX (Evolution Language for XML-Schema [16]) operations. In general ELaX statements are add, delete and update operations on the components of XML Schema, specified by a user.

ROfEL allows the identification and deletion of unnecessary and redundant modifications by applying different heuristic rules. Additionally, invalid operations are also corrected or removed. In general if the preconditions and conditions for an adaptation of two ELaX log entries are satisfied (e.g. EID equivalent, op-Type correct, etc.), one rule is applied and the modified, reduced log is returned.

We are confident, that even if ROfEL is domain specific and the underlying log is specialized for our needs, the above specified rules are applicable in other scenarios or applications, in which the common modification operations add, delete and update are used (minor adaptations preconditioned).

Future work. The integration of a cost-based component in ROfEL could be very interesting. It is possible, that under consideration of further analyses the combination of different operations (e.g. rule (10)) is inefficient in general. In this and similar cases a cost function with different thresholds could be defined to guarantee, that only efficient adaptations of the log are applied. A convenient cost model would be necessary, but this requires further research.

Feasibility of the approach. At the University of Rostock we implemented the prototype CodeX (Conceptual design and evolution for XML Schema) for dealing with the co-evolution [14] of XML Schema and XML documents; ROfEL and corresponding concepts are fully integrated. As we plan to report in combination with the first release of CodeX, the significantly reduced number of logged operations proves that the whole algorithm is definitely feasible.

7. REFERENCES

[1] XSL Transformations (XSLT) Version 2.0. `http://www.w3.org/TR/2007/REC-xslt20-20070123/`, January 2007. Online; accessed 25-June-2014.

[2] Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/2008/REC-xml-20081126/`, November 2008. Online; accessed 25-June-2014.

[3] XQuery 1.0: An XML Query Language (Second Edition). `http://www.w3.org/TR/2010/REC-xquery-20101214/`, December 2010. Online; accessed 25-June-2014.

[4] 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 25-June-2014.

[5] XSL Transformations (XSLT) Version 3.0. http://www.w3.org/TR/2013/WD-xslt-30-20131212/, December 2013. Online; accessed 25-June-2014.

[6] F. Cavalieri. Querying and Evolution of XML Schemas and Related Documents. Master's thesis, University of Genova, 2009.

[7] F. 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] F. Cavalieri, G. Guerrini, M. Mesiti, and B. Oliboni. On the Reduction of Sequences of XML Document and Schema Update Operations. In *ICDE Workshops*, pages 77–86, 2011.

[9] H. U. Hoppe. Task-oriented Parsing - a Diagnostic Method to Be Used Adaptive Systems. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '88, pages 241–247, New York, NY, USA, 1988. ACM.

[10] M. Klettke. *Modellierung, Bewertung und Evolution von XML-Dokumentkollektionen*. Habilitation, Fakultät für Informatik und Elektrotechnik, Universität Rostock, 2007.

[11] R. Kramer. iContract - the Java(tm) Design by Contract(tm) tool. In *In TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 295. IEEE Computer Society, 1998.

[12] X. Li and G. Agrawal. Efficient Evaluation of XQuery over Streaming Data. In *In Proc. VLDB'05*, pages 265–276, 2005.

[13] E. Maler. Schema Design Rules for UBL...and Maybe for You. In *XML 2002 Proceedings by deepX*, 2002.

[14] T. Nösinger, M. Klettke, and A. 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] T. Nösinger, M. Klettke, and A. Heuer. A Conceptual Model for the XML Schema Evolution - Overview: Storing, Base-Model-Mapping and Visualization. In *Grundlagen von Datenbanken*, 2013.

[16] T. Nösinger, M. Klettke, and A. Heuer. XML Schema Transformations - The ELaX Approach. In *DEXA (1)*, pages 293–302, 2013.