A Lightweight XML Constraint Check and Update Framework

Hong Su, Bintou Kane, Victor Chen, Cuong Diep, De Ming Guan, Jennifer Look, and Elke A. Rundensteiner

Department of Computer Science
Worcester Polytechnic Institute
100 Institute Road, Worcester, MA 01609-2280
{suhong,bkane,vchen,cdiep,deguan,jlook,rundenst}@cs.wpi.edu

Abstract. Support for updating XML documents has recently attracted interest. When an XML document is to conform to a given schema, the problem of *structural consistency* arises during updating, i.e., how to incrementally guarantee that the modified XML document continues to conform to the given XML Schema. To achieve this following the traditional database approach, the XML Schema would first have to be analyzed to construct a structured repository and the XML documents would have to be loaded into this repository before any update could be checked for possible schema constraint violation. Due to the very nature of XML being lightweight and freely shared over the Web, we instead propose a novel approach towards incremental constraint checking that follows the loosely-coupled web paradigm. Namely, we propose to rewrite an XML update query into a safe XML update query by extending the original query with appropriate constraint checking subqueries. This enhanced XML update query can then safely be executed using any existing XQuery engine that supports updates. In order to verify the feasibility of our approach, we have implemented a prototype, *SAXE*, that implements the above techniques by extending the Kweelt XML query engine by University of Pennsylvania with both XML update support as well as incremental constraint support.

Keywords: XML Update, XQuery, XML Schema, Structural Consistency.

1 Introduction

1.1 Motivation

Change is a fundamental aspect of persistent information and data-centric systems. Information over a period of time often needs to be modified to reflect perhaps a change in the real world, a change in the user's requirements, mistakes in the initial design or to allow for incremental maintenance.

However, change support for XML in current XML data management systems is only in its infancy. First of all, practically all change support is tightly tied to the underlying storage system of the XML data. For example, both in IBM DB2 XML Extender [IBM00b] and Oracle 9i XSU [Ora02] who support

A. Olivé et al. (Eds.): ER 2002 Ws, LNCS 2784, pp. 39–50, 2003.

decomposition of XML data into relational storage or object-relational storage respectively, the user would then need to work with the relational data representing the original XML document as with any other relational data. In particular, any update on the XML data has to be specified using SQL and then will be executed on the underlying relational data. This requires users to be aware of not only the underlying storage system but also the particular mapping chosen between the XML model and the storage model. In other words, there is a lack of abstraction for specifying native updates to XML data independent of the different underlying storage models.

As the first step for native XML update support, a native language for updating XML must be proposed. In this paper, since no World Wide Web Consortium proposal on XML updating has emerged to date, we utilize an extension of XQuery [W3C01b] to support powerful XML updates [TIHW01]. We will further discuss choice of XML update language in Section 2.

An indispensable next step towards supporting updates is to provide a mechanism for maintaining the *structural consistency* of the XML documents with all associated XML schemata (if any) during the course of the update. Structural consistency is a desired property in database systems since they require that the data must always conform to its schema. An update is considered to be *safe* only if it will not result in any data violating the associated schema. For example, in a relational database, if an attribute is defined as NOT NULL in the schema, an insertion of a tuple with a NULL value for this attribute will be regarded as an unsafe operation and thus would be refused by the system. Though it is not required that XML documents must always have associated schemata due to their "self-describing" nature, many application domains tend to use some schema specification in either DTD [W3C98] or XML Schema [W3C01a] format to enforce the structure of the XML documents. Whenever XML schemata are associated with the XML data, then structural consistency should also be taken care of during update processing. No work has been done to date to address this issue for native XML.

1.2 Illustrating Example

For example, Figures 1 and 2 show an XML schema *juicers.xsd* and an XML document *juicers.xml* conforming to the schema respectively. Suppose the user specifies to remove the cost of the juicer with name "Champion Juicer" (the first juicer in *juicers.xml*). This operation will render the Champion juicer to no longer have a *cost* subelement. Such an updated XML document is inconsistent with the schema *juicers.xsd* since a *juicer* element is required to have at least one *cost* subelement, indicated as <`xsd: element ref = cost minOccurs = 1 maxOccurs = unbounded/`> in *juicer.xsd*. This update would however have been allowed for the second juicer (i.e., Omega Juicer). Some mechanisms must be developed to prevent such violation of structural consistency.

```
<xsd: schema xmlns: xsd = http://www.w3.org/2001/XMLSchema>
  <xsd: element name = "juciers">
   <xsd: complexType>
    <xsd: sequence>
     <xsd: element ref = "juicer" minOccurs = "0" maxOccus = "unbounded"/>
    </xsd: sequence>
   </xsd: element>
  <xsd: element name = "juicer">
    <xsd: complexType>
     <xsd: sequence>
       <xsd: element ref = "name"/>
       <xsd: element ref = "image" minOccurs = "unbounded">
      <xsd: element ref = "cost" minOccurs = "0" maxOccurs = "unbounded" />
     </xsd: sequence>
      <xsd: attribute ref = "quality" use = "optional"/>
    </xsd: complexType>
  </xsd: element>
  <xsd: element name = "name" type = "xsd: string"/>
  <xsd: element name = "cost" type  = "xsd: string"/>
  <xsd: element name = "image" type = "xsd: string"/>
  <xsd: attribute name = "quality" type = "xsd:string"/>
</xsd: schema>
```

Fig. 1. Sample XML Schema: *juicers.xsd*

```
<juicers>
 <juicer>
  <name> Champion Juicer </name>
  <image> images\champion.gif </image>
  <cost> 239.00 </cost>
 </juicer>
 <juicer>
  <name> Omega Juicer </name>
  <image> images\omega.jpg </image>
  <cost> 234.00 </cost>
  <cost> 359.50 </cost>
 </juicer>
</juicers>
```

Fig. 2. Sample XML Document: *juicers.xml*

1.3 Desiderata of Preserving Structural Consistency

In our current work, we assume the schema is the first-class citizen. In this sense, an update to an XML document is only allowed when the update is *safe*, i.e., the updated data would still conform to the given XML schemata. In this section, we discuss the desiderata of the mechanism for checking the safety of XML data updates.

Native XML Support. There have been some techniques proposed for translating constraints in XML to constraints in other data models, say the relational model [KKRSR00] or the object model [BGH00]. Once the mapping is set up, XML constraint checking would be achieved by the constraint enforcement mechanism supported in the other underlying models. However we prefer a native XML support for several reasons. Primarily, we want to avoid the overhead of a load into a database management system (DBMS) as well as the dependency of XML updates on some specific alternate representation.

Loosely-Coupled Constraint Checking Support. Following the traditional database approach shown in Figure 3, the XML Schema would first be analyzed to construct a fixed structure and XML documents could then be loaded into a repository in order to allow updates on the document to be checked. It is preferable to have the validity checking a lightweight standing-alone module rather than being tightly coupled to an XML DBMS. Ideally the constraint checking tool should be a middleware service so that it is general and portable over all XML data management systems.

Incremental Constraint Checking. A naive approach to ensuring the safety of data updates is to do a validation from scratch (shown in Figure 4), namely, to first execute the updates, then run a validating parser[1] on the updated XML document, and lastly decide whether to roll back to the original XML document based on the validation result. Such an approach is inefficient since it involves redundant checking on those unchanged XML fragments. It is preferable to have an incremental checking mechanism where only the modified XML fragments rather than the complete XML document are checked. Moreover when the validating parser is run on an XML document modified by a batch of updates and any inconsistency is detected, the parser is unable to tell which updates have caused the inconsistency. Hence, this would make a roll back of only the unsafe updates (but not the changes made by safe updates) impossible.

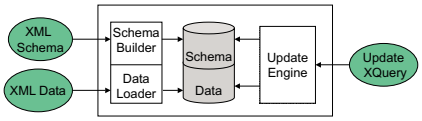

Fig. 3. Tightly-Coupled Approach

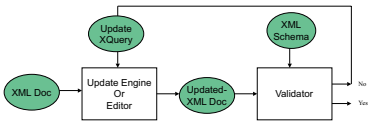

Fig. 4. From Scratch Validation Approach

1.4 Our Approach and Contributions

In this paper we introduce a native, incremental and lightweight framework for checking the validity of data updates specified in an XQuery update language *Update-XQuery* [TIHW01]. The key concept we exploit is the capacity of the XQuery query language to not only query XML data but also XML Schema. This allows us to rewrite Update-XQuery statements by extending them with appropriate XML constraint checking sub-queries.

In summary, we make the following contributions in this work:

1. We identify the issue of preserving structural consistency during the update of XML documents in a loosely-coupled native XML context.
2. We propose a general constraint checking framework that provides native, incremental and lightweight XML constraint checking support.
3. We describe the prototype system *SAXE* we have implemented. We verified the feasibility of this proposed approach by comparing its performance against that of current state-of-the-art solutions.

2 Related Work

Several XML update languages have been proposed [IBM00b] [Obj99] [SKC+00] [TIHW01]. The expressive power of the language concerns two capabilities: i.e.,

[1] Most XML document parsers [IBM00a] support validating the XML document against the given DTD or XML Schema.

the power to specify (1) what nodes to update (i.e., querying over the data to select target nodes) and (2) what actions to take on the selected nodes.

[IBM00b] provide their own language for native XML update support in DB2 XML Extender. The expressive power of the update language is limited. XML Extender allows to specify target nodes in the XML document using XPath expressions [W3C99]. XPath is a subset of XQuery, e.g., in particular, XPath does not support variable bindings. Moreover Extender only allows in-place content update on the selected nodes without other basic support such as inserting new nodes or removing existing nodes. Excelon [Obj99] offers an update language. The disadvantage of this language is that it uses its own proprietary query specification which detracts from its compatibility with the standard XML query language. An XML working group XML:DB [XML02] proposes XUpdate [XUp02] which also has the expressive power limitation in that it uses XPath as the query specification. [TIHW01] stands out among the XML update languages in terms of its expressive power and compatibility with XQuery. It is a natural extension of XQuery that supports the application of a set of update operations including insertion of new data, removal or in-place modification of existing data on bound variables.

None of the above work deals with the problem of incremental validation after the updates. To the best of our knowledge, our earlier work on XEM [SKC+00] is one of the first efforts addressing this problem. XEM proposes a set of update primitives each of which is associated with semantics ensuring the safety of the operation. In XEM, a data update primitive on the other hand is only executed when it passes the validity check. The main limitations of XEM are: (1) the data update primitives in XEM can be only performed on one single element selected by an XPath expressions; (2) XEM is a tightly-couple approach, namely, we implemented an engine on top of PSE (a lightweight object database), mapped the DTD to a fixed schema and loaded the data into object instances. Such a paradigm requires schema evolution support from PSE and specialized constraint enforcement has to be hardcoded into the PSE system.

3 XML Query and Update Language

3.1 XML Query Language: XQuery

XQuery [W3C01b] is an XML query language proposed by W3C. An XQuery statement is composed of several expressions. An important expression in XQuery is the FLWR expression constructed from `FOR`, `LET`, `WHERE` and `RETURN` clauses.

1. `FOR` and `LET` clauses bind values or expressions to one or more variables.
2. `WHERE` clause (optional) filters the bindings generated by `FOR` and `LET` clauses by any specified predicates.
3. `RETURN` clause constructs an output XML document.

We give an example XQuery over the XML document in Figure 2:

```
For $p in document(‘‘juicers.xml’’)/juicer, $c in $p/cost[1]
Return $c.
```

The variable $\$p$ is bound to iterate over each element node satisfying the expression `document(‘‘juicers.xml’’)/juicer`. For each identified binding of $\$p$, $\$c$ is bound to the first *cost* child node of $\$p$ and returned.

3.2 XML Update Language: Update-XQuery

[TIHW01] proposes a set of update operations and embeds them into the XQuery language syntax. Each update operation is performed on a target object indicated as *target*. Table 1 gives the set of update operations and their semantics.

Table 1. Taxonomy of Update Operations

Update Operation	Description
Delete *child*	Remove *child* from children list of *target*
Rename *child* To *n*	Rename *child* to name *n*
Insert new_attr(*n*, *v*)	Insert new attribute with name *n* and value *v* to *target*
Insert *c* Before/After *child*	Insert XML fragment with content of *c* directly before/after *child*
Replace *child* With new_attr(*n*, *v*)	Replace *child* with attribute with name *n* and value *v*
Replace *child* With `c`	Replace *child* with XML fragment with content *c*

[TIHW01] extends XQuery's original FLWR expressions to accommodate the update operations by introducing `FOR`... `LET`... `WHERE`... `UPDATE`..., i.e., FLWU expressions. We will refer to this extension of XQuery now as *Update-XQuery*. The BNF of FLWU expression syntax is shown in Figure 3.2 while the BNF for the `UPDATE` clause (`subOp` in Figure 3.2) in particular is shown in Figure 6.

```
FOR $binding1 IN XPath-expr, ...
LET $binding := XPath-expr, ...
WHERE predicate1, ...
UPDATE $binding {subOp{, subOp}*}
```

Fig. 5. Syntax of Update-XQuery

```
DELETE $child |
RENAME $child TO name |
INSERT (new_attr(name, value) |
        content [BEFORE | AFTER $child] |
        $copyTarget [BEFORE | AFTER $child]) |
REPLACE $child WITH (new_attr(name, value) |
                     content |
                     $copyTarget) |
FOR $binding IN XPath-expr, ...
WHERE predicate1, ...
UPDATE $binding {subOp {, subOp}*}
```

Fig. 6. BNF of `subOp`

The semantics of `FOR`, `LET` and `WHERE` clauses are exactly the same as that in a FLWR expression as briefly described in Section 3.1. The `UPDATE` clause specifies the target node to be updated and a sequence of update operations or FLWU expressions to be applied on the target node.

Figure 2 shows a sample Update-XQuery on the XML document in Figure 2. The variable $\$p$ is bound to iterate over each element node satisfying the expression `document(''juicers.xml'')/juicer` (line 1). For each identified binding of $\$p$, $\$c$ is bound to the first *cost* child nodes of $\$p$ (line 2) and $\$p$ is updated by deleting its child node, i.e., the binding of $\$c$ (line 4).

4 XML Framework For Safe Updates

Our Overall Approach. In order to allow only consistent updates to be processed on XML documents, we aim to develop a loosely-coupled update strategy that supports incremental schema constraint checking that accesses only updated parts of the XML document. The key idea is to first generate a safe Update-XQuery statement from a given input Update-XQuery statement. This generated safe Update-XQuery statement, still conforming to the standard Update-XQuery BNF, can then be safely executed on any XQuery update engine. This way we succeed in separating the concern of constraint check verification from that of developing the XML query and update engine.

For this safe query generation, we design appropriate *constraint checking subqueries*. The constraint checking subqueries take input parameters from the update operation and determine whether the update operation is valid or not. For this, we exploit the capacity of the XQuery query language to not only be able to query XML data but also XML Schema. This allows us to rewrite Update-XQuery statements by extending them with appropriate XML constraint check sub-queries for each update operation as in Table 1. The execution of an update operation is conditional on passing the constraint checking.

Illustrating Example. For example, Figure 2 shows the rewritten Update-XQuery from the Update-XQuery in Figure 2. There is one update operation in the query, i.e., `DELETE $c` in line 4. We can see that lines 3, 5 and 6[2] in Figure 2 have been inserted into this update operation so that this update is only executed when `delElePassed(...)` (line 5) returns true. `delElePassed(...)` is a constraint check function which determines the validity of the update `DELETE $c`. The subquery `schemaChkDelEle(...)` in line 3 is a function that provides information that is needed by `delElePassed(...)` to make the determination. We will further discuss the details of these two functions in Section 5.3.

```
1 FOR $p in document("juicers.xml")/juicer,
2       $c in $p/cost[1]
3 UPDATE $p {
4      DELETE $c
5 }
```

Fig. 7. Sample Update-XQuery

```
1 FOR $p in document("juicers.xml")/juicer,
2       $c in $p/cost[1]
3 LET $constraint =
  schemaChkDelEle("juicers.xsd","juicer","cost")
4 UPDATE $p {
5 WHERE delElePassed($c,$p/cost,$constraint)
6           UPDATE $p {
7               DELETE $c
8           }
9 }
```

Fig. 8. Sample Safe Update-XQuery

***SAXE* Architecture.** Figure 9 shows the architecture of *SAXE*[3], the framework for generating a safe Update-XQuery statement given an input Update-

[2] Line 6 is added only to meet the syntax requirement.
[3] SAfe Xml Evolution.

XQuery. The *safe Update-XQuery generator SAXE* is composed of the five components described below:

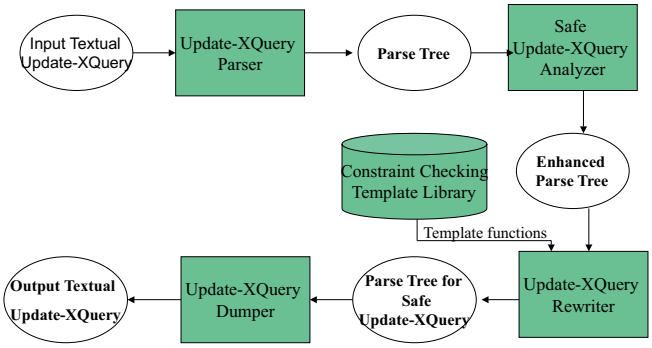

Fig. 9. An Incremental Yet Loosely-Coupled Update Processing Framework Supporting XML Updates with Schema Constraint Validation

1. *Update-XQuery Parser.* The parser takes an Update-XQuery statement and constructs a parse tree representation [ASU86] from it.
2. *Update-XQuery Analyzer.* Given a parse tree, the analyzer identifies more detailed information about types of update operations in the parse tree and derives an enhanced parse tree (refer to Section 5.2).
3. *Constraint Checking Template Library.* We generalize the constraint checking procedures by defining named parameterized XQuery functions called *constraint checking templates*. Each constraint checking template is in charge of checking constraints for one update type.
4. *Update-XQuery Rewriter.* The rewriter handles the actual generation of a safe Update-XQuery. It determines how to rewrite the original Update-XQuery statement by plugging in the appropriate constraint checking functions from the template library and correspondingly modifying the enhanced parse tree.
5. *Update-XQuery Dumper.* The dumper constructs a textual format of the modified Update-XQuery statement from the enhanced parse tree, which now is in the standard Update-XQuery syntax.

5 Components of Constraint Checking Framework

We now describe the main components of the framework shown in Figure 9.

5.1 Update-XQuery Parser

Given an Update-XQuery statement, the Update-XQuery parser constructs a parse tree which is composed of objects of classes that were designed to store the parsed query. For example, a class *Update* is defined to store update clauses. Subclasses of class *Update* are defined for four types of update operations, i.e., *Delete*, *Rename*, *Insert* and *Replace*, respectively.

5.2 Safe Update-XQuery Analyzer

Given an internal representation of an Update-XQuery, the analyzer will determine a more specific sub-type of an update operation. For example, the analyzer would examine the content of an object of class *Delete* to classify the update as either deleting an element or deleting an attribute. The detailed information of update types would then be embedded into the original parse tree. We call the new parse tree an *enhanced parse tree*.

5.3 Constraint Checking Template Library

The library stores templates that account for every type of update possible using our Update-XQuery language (See BNF in Figure 3.2). A constraint check is composed of three steps which are:

1. Query the XML schema to identify any constraints that may be violated by the specified update.
2. Query the XML document to gather information pertaining to the target elements or attributes.
3. Compare the information retrieved from the two previous steps and thus identify whether the constraints would be violated by the update.

We illustrate how this constraint check is done for a delete-element operation. The constraint checking functions `schemaChkDelEle` and `delElePassed` shown in Figures 5.3 and 5.3 jointly achieve the three steps mentioned above.

```
Function integer schemaChkDelEle($xsdName, $parentEleName, $childEleName) Return Integer
1 {
2  For $pDef In document($xsdName)/xsd:element[@name = $parentEleName],
3        $cRef In $pDef//xsd:element[@ref = $childEleName]
4   Let $cRefMinOccurs:= $cRef/minOccurs
5   Return $childRefMinOccurs }
```

Fig. 10. Constraint Checking Function *schemaChkDelEle*

```
Function delElePassed($childBinding, $childBindingPath, $childMinOccurs) Return Boolean
1 {
2      LET $childInstCount := count($childBindingPath)
3      Return
4       If ($childMinOccurs <= $childInstCount - 1
5        Then TRUE
6        Else FALSE
7 }
```

Fig. 11. Constraint Checking Function*delElePassed*

The Constraint Checking Function. `schemaChkDelEle` queries over the schema (i.e., step 1) for the information related to the constraints that may be violated when deleting an element. Deleting an element e of element type t can only violate the constraint of a required minimum occurrence of the elements of type t in the content model of e's parent. `schemaChkDelEle` is to retrieve the minimum occurrence of elements of type $\$childEleName$ in the parent type *parentEleName*. In particular, line 2 queries the XML schema file, specified by the file name $\$xsdName$, to find the element definition $\$pDef$ for type $\$parentEleName$. The element definition of *parentEleName*'s subelement referring to type *childEleName* is stored in $\$childRef$ in line 3. Line 4 then retrieves the minimum occurrence of element type *childEleName* in *parentEleName*.

Constraint Checking Function. `delElePassed` checks whether the data update is safe based on the schema constraint information collected by `schemaChkDelEle`. `delElePassed` is composed of two parts:

1. **Query over Data** (i.e., step 2). Line 2 begins querying over the XML document to find the actual count of instances of type *childEleName* that are subelements of the target object. These instances can be retrieved by the XPath expression $\$childBinding$. Function *count* on the retrieved instances returns the count of these instances. Thus there would be only *childInstCount* - 1 instances of type *childEleName* if the update is allowed to occur.
2. **Integration of Query Result over Schema and Data** (i.e., step 3). Line 4 compares the information from the XML schema and data. It compares the minimum occurrence requirement (i.e., *childRefMin*) and the actual occurrence if the update were indeed to proceed. In this example, this would be *childInstCount* - 1. If actual occurrence after the update had occurred were larger than the minimum occurrence requirement, this check is passed and the update operation is regarded as valid.

5.4 Safe Update-XQuery Rewriter

The Safe XQuery Rewriter traverses the enhanced parse tree. For each update operation, based on the update type, the Rewriter determines which template function should be used for checking the constraints of the update. Since the template is parameterized, the Rewriter also instantiates the parameters. Values for these parameters can be identified through the analysis of different parts of the parsed XQuery.

This can be seen in Figure 2. The `delElePassed` template function takes in three parameters to execute its query. For this particular example,"*juicer.xsd*" (the file name of the XML Schema), "*juicer*" (the type name of the parent element of the to-be-deleted element) and "*cost*" (the type name of the to-be-deleted element) are the three instantiated parameters respectively.

Once all parameters have been assigned values, the Rewriter needs to insert the instantiated template function into the original query. The Rewriter modifies the parse tree by inserting the constraint checking function for example via a where clause after the associated update clause (line 5 in Figure 2). After all modifications have been done to the original update XQuery, the safe Update-XQuery generation is complete. Finally, a resulting safe Update-XQuery statement is produced by the query rewriter module.

6 The *SAXE* System

We have designed and implemented the above proposed query rewriting techniques as a prototype system, called *SAXE*. Our system is based on Kweelt [SD02], a query engine for the Quilt XML query language [CRF02], a precursor of the XQuery standard, developed by the University of Pennsylvania. Kweelt is composed of two parts, i.e., the language parser and language evaluator. The parser takes a Quilt statement and constructs a parse tree. The evaluator then executes the query against the data. First, we extended the Java Compiler Compiler file (JavaCC) which is a Java parser generator in Kweelt so that the `Update` clauses are accepted by the language parser. Second, we have extended the evaluator so that an Update-XQuery statement can be executed.

We compare *SAXE* against the from-scratch validation solution, which means we first perform the given update using the extended Kweelt update engine and then we run the modified XML document through an XML-Schema Validator [Tom02] to check for conformance with the given XML schemata. We added the time for regular XQuery execution to the time needed to run the update document through the validator. Generating a typical safe Update-XQuery takes about 400 - 500 milliseconds. For some cases, the safe Update-XQuery execution takes slightly longer than the from-scratch validation solution. However, this does not mean that a safe Update-XQuery is less efficient than using validators after updates are executed. The argument is that the safe Update-XQuery is a one step process where updates are only performed once the updates are deemed safe so that all the attempts for invalid updates will be prevented. On the other hand for the execution of non-safe Update-XQuery, as mentioned in Section 1.3, the system may need to iterate several times between attempting updates and then verifying if the updates leave the XML document in a consistent state.

7 Conclusion

In this paper, we propose a lightweight approach to ensure the structural consistency of XML documents after updates. More precisely, we propose that an Update-XQuery statement can be rewritten into a safe Update-XQuery statement by embedding constraint checking operations into the query. This approach is lightweight in the sense that it can be implemented as a middleware independent of any underlying system for XML data management. To ensure the structural consistency, any Update-XQuery statement is first fed to our safe Update-XQuery generator, *SAXE*, while the returned safe Update-XQuery statement can then be executed by any system supporting Update-XQuery.

Currently, our safety checking semantics is at the atomic level, i.e., each atomic update on a single object is allowed if this update leads to a valid XML document. As a next step, we want to explore the concept of transactional update, i.e., a batch of updates are only allowed to be executed if the overall effect of executing them leads to a valid document.

References

ASU86. V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools* . Addison-Wesley, 1986.

BGH00. L. Bird, A. Goodchild, and T. A. Halpin. Object role modelling and xml-schema. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 309–322, 2000.

CRF02. Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt, 2002.

IBM00a. IBM. XML Parser for Java, 2000. `http://www.alphaworks.ibm.com/tech/xml4j`.

IBM00b. IBM Software: Database and Data Management. DB2 XML Extender. `http://www-4.ibm.com`, 2000.

KKRSR00. G. Kappel, E. Kapsammer, S. Rausch-Schott, and W. Retschitzegger. X-ray - Towards integrating XML and relational database systems. In *International Conference on Conceptual Modeling / the Entity Relationship Approach*, pages 339–353, 2000.

Obj99. Object Design. Excelon Data Integration Server. `http://www.odi.com/excelon`, 1999.

Ora02. Oracle. Oracle9i application developer's guilde - xml release 1 (9.0.1): Database support for xml. `http://download-east.oracle.com/otndoc/oracle9i/901_doc/appdev.901/a88894/adx05xml.htm`, 2002.

SD02. A. Sahuguet and L. Dupont. Querying xml in the new millennium, 2002.

SKC+00. H. Su, D. Kramer, K. Claypool, L. Chen, and E.A. Rundensteiner. XEM: Managing the Evolution of XML Documents. In *International Workshop on Research Issues in Data Engineering*, pages 103 – 110, 2000.

TIHW01. I. Tatarinov, Z. Ives, A.Y. Halevy, and D.S. Weld. Updating XML. In *SIGMOD*, pages 413 – 424, 2001.

Tom02. Henry Tompson. xsv: schema validator, 2002.

W3C98. W3C. *Guide to the W3C XML Specification ("XMLspec") DTD, Version 2.1.* `http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm`, 1998.

W3C99. W3C. XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, 1999.

W3C01a. W3C. *XML Schema* . `http://www.w3.org/XML/Schema`, 2001.

W3C01b. W3C. *XQuery 1.0: An XML Query Language.* `http://www.w3.org/TR/xquery/`, 2001.

XML02. XML:DB. `http://www.xmldb.org/`, 2002.

XUp02. XUpdate. XML:DB. `http://www.xmldb.org/xupdate/xupdate-wd.html`, 2002.