Consistently Updating XML Documents using Incremental Constraint Check with XQueries

A Thesis

by

Bintou Kane

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2003

APPROVED:

Professor Elke Rundensteiner, Major Thesis Advisor

Professor Kathi Fisler, Thesis Reader

Professor Micha Hofri, Head of Department

Abstract

When updating a valid XML Data or Schema, an efficient yet light-weight mechanism is needed to determine if the update would invalidate the document. Towards this goal, we have developed a framework called SAXE. First, we analyzed the constraints expressed in XML schema specifications to establish constraint rules that must be observed when a schema or an XML data conforming to a given XML Schema is altered. We then classify the rules based on their relevancy for a given update case. That is, we show the minimal set of rules that must be checked to guarantee the safety for each update primitive. Next, we illustrate that this set of incremental constraint checks can be specified using generic XQuery expressions composed of three type of components. Safe updates for the XML data have the following components: (1) XML schema meta-queries to retrieve any constraint knowledge potentially relevant to the given update from the schema or XML data being altered, (2) retrieval of specific characteristics from the to-be-modified XML, and (3) an analysis of information collected about the XML schema and the affected XML document to determine validity of the update. For the safe schema alteration, the components are: (1) XML schema meta-queries to retrieve relevant information from the schema (2) analysis and usage of retrieved information to update the schema, and (3) propagation of the changes to the XML data when necessary. As a proof of concept, we have established a library of these generic XQuery constraint checks for the type-related XML constraints. The key idea of SAXE is to rewrite each XQuery update into a safe XML Query by extending it with appropriate constraint check subqueries. This enhanced XML update query can then safely be executed using any existing XQuery engine that supports updates - thus turning any update engine automatically into an incremental constraint-check engine. In order to verify the feasibility of our approach, we have implemented a

prototype system SAXE that generates safe XQuery updates. Our experimental evaluation assesses the overhead of rewriting as well as the performance of our loosely-coupled incremental constraint check approach compares with the more traditional first-change-document and then revalidate-it approach.

Acknowledgements

I would like to express my gratitude to my adviser, Prof. Elke Rundensteiner for her excellent guidance through the process of completing this thesis. Without her suggestions, ideas, feedback and support this thesis would not have been achieved.

I also thank my thesis reader Prof. Kathi Fisler for her valuable feedback and my colleagues at the DSRG Lab.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Today XML is the facto standard data exchange format for information on the Web. Nearly all-major database system providers have extended their existing database technologies to manage XML data. Each of these vendors assumes 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. Practically all change support is tightly tied to the underlying storage system of the XML data. For example, both IBM DB2 XML Extender [IBM00b] and Oracle 9i [Ora02], which support decomposition of XML data into relational [IBM00b] or object-relational storage [Ora02], still require 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 and updates must be done using SQL-like language. Little has been done to provide mechanisms for maintaining the *structural consistency* of the XML documents with all associated XML schemata during an update. *Structural consistency* is a desired property in database systems which requires 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. Though it is not required that XML documents must always have associated schema 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. An update to an XML document should thus only be allowed when the update is *safe*, i.e., the updated data would continue to conform to the given XML schemata.

To achieve this, techniques have been proposed for translating constraints in XML to constraints in other data models, say the relational model [KKRSR00] or the object model [BGH00]. Following the traditional database approach depicted in Figure 1.1, first the XML Schema would be analyzed to construct a schema in the underlying storage system, and second the XML documents could be loaded into the repository, only after that would updates on the document be permitted. Thereafter XML constraint checking would be achieved by the constraint enforcement mechanism supported in the underlying data store. However we prefer **native XML update** support to avoid the overhead of a load into a database management system (DBMS), and the dependency of XML updates on some specific alternate data representation.

Alternatively as depicted in Figure 1.2, a native XML approach to ensuring the safety of data updates is to first execute the updates on the XML document directly, 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 likely inefficient since it involves redundant checking of the complete XML document.

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

It is preferable to have an **incremental checking mechanism** where only the modified XML fragments rather than the complete XML document are checked. Also it would be preferable not to have to load it into some data repository first.

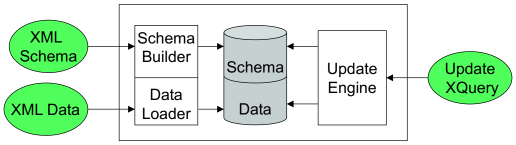


Figure 1.1: Tightly-Coupled Approach

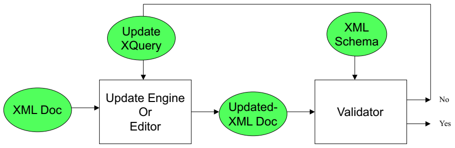


Figure 1.2: Loosely-Coupled Approach

1.2 Our Approach

It is preferable to provide constraint checking as a lightweight middleware service rather than being tightly coupled to an XML data management system. This way the service could be general and portable across different XML applications. The key concept we propose to exploit is the capacity of the XQuery query language to not only query XML data but also XML Schema. This allows us to rewrite XML update statements by extending them with appropriate XML constraint check sub-queries. This enhanced XML update query can then safely be executed using any existing XQuery engine that supports updates - thus turning any update engine automatically into an incremental constraint-check engine. Figure 1.3 depicts the main flow of our constraint checking approach.

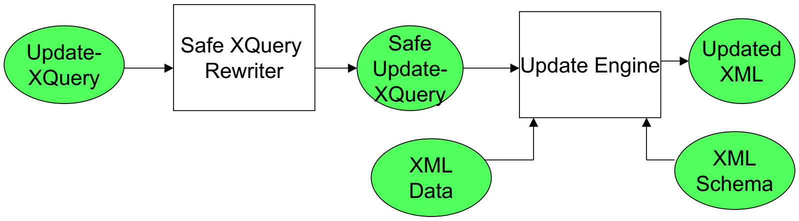


Figure 1.3: An Incremental Yet Loosely-Coupled Update Processing Framework Supporting XML Updates with Schema Constraint Validation

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

1. We analyze XML schema specifications and develop general constraint rules to perform safe update operations for the XML Schema and the XML Data.

2. We propose a general constraint checking framework that provides native, incremental and lightweight XML Schema and XML Data update support by query rewriting.

3. We have implemented a prototype system to verify the feasibility of this proposed approach.

4. We have conducted an experimental study that compares the performance of the proposed approach against that of current state-of-the-art solutions.

1.3 Thesis Outline

The rest of this thesis is structured as follows:

1. Related Work: Gives an overview of literature and tools related to this domain of study.

2. XML Update language: Gives a concept of XQuery and the update language used for SAXE.

3. Modeling XML Schema, XML Data and their Interrelationship: Models XML Schema, XML Data and their mapping.

4. Consistency Under XML Data and XML Schema Updates: Shows How the system integrity is maintained when changes originate either from the Schema or XML Data.

5. SAXE Framework: Gives the overview of the SAXE System architecture and its implementation.

6. SAXE Experiments: Gives a precise evaluation of the system.

7. Conclusion: Gives a detailed summary and possible future work on this domain.

8. Appendix A: Describes the template libraries for safe data updates when changes originate from the XML data.

9. Appendix B: Describes the safe update queries generated when change originate from the XML schema.

Chapter 2

Related Work

2.1 XML Evolution

With XML becoming more mainstream, and much more XML data becoming available around, new technologies has been developed to better process and manipulate XML. Management of XML documents is becoming an increasingly important task [TIHW01a]. We see a proliferation of databases that store, query and update XML documents.

Both the data and the structure of XML documents (schema) tend to change over time for a multitude of reasons. Changes are fundamental aspect of persistent information and data centric systems. Over time information tends to change, and modification should be made to reflect the necessary evolutions. Beyond frequent data updates which are widespread, we find that schema changes are also fairly common in modern applications for the following reasons. Schema mapping techniques [HH00] which aim at mapping between heterogeneous sources are semi-automatic and depend a lot on domain knowledge. As a result, a good mapping is thus hard to find and may evolve over time. Sources also have been found to be fairly volatile to the extent that some of them may be temporarily or even permanently unavailable [ITY99]. Another important point is that a schema change could occur for numerous reasons during the software life-cycle, including design errors, schema redesign during the early stages of database deployment, the addition of new functionalities and even new developments in the application domain.

2.2 XML and Database Systems

Even though the topic of updating XML data is still at its infancy, there have been significant developments on this topic over the last few years. Today the community sees XML not only as a way to serialize and communicate the already existing data structures, but most importantly also as a way to think about modeling application data.

One of the first database systems that supported updates is the eXcelon XML repository [eXc98]. Its support is basic and expresses simple deletion and insertion using an extension of the XPath language [W3C99]. In the spirit of making XML fully evolve into a universal data repository, more and more suitable data query and update languages are being developed. Some of the tools such as (XSLT) [W3C01d] have focused on various language formats as a mechanism for manipulating XML data. Extensible Stylesheet Language Transformations (XSLT) [W3C01d] is a language designed for transforming individual XML documents. Contrary to SAXE, XSLT does not require any schema attached to the XML data. The user can specify arbitrary data transformation rules. Lexus (XML Update Language) [Inf00] is another declarative language proposed by an open source group, Infozone, to update stored documents. However Lexus uses primitives, which only work on the document level without taking the schema into account.

There are few native XML editor tools capable of validating XML schema and data updates, one of these editors is XMLspy [XML]. Contrary to editors like XMLspy [XML], SAXE allows a set of related safe updates to be done automatically in one batch using an XQuery expression. Also SAXE is a one step process where updates are only performed once they are deemed safe so that all the attempts for invalid updates will be prevented and the safe ones will be performed. With the XMLSpy editor any invalid update will force a roll back for all intended updates. The XEM project [SKC$^+$00] from WPI deals with the problem of XML evolution and updating. XEM proposes a set of update primitives, each of which is associated with semantics ensuring the safety of the operation. The main limitations of XEM are: (1) the data update primitives in XEM can only be performed on one single element selected by an XPath expression at a time; (2) XEM is a tightly-coupled approach, namely, it is implemented as an engine on top of PSE (a lightweight object database), by first mapping the DTD to a fixed object schema and then loading the data into the schema as object instances. Such more traditional database paradigm requires schema evolution support from the underlying DBMS engine, in the XEM case, the PSE system. While XEM provides such schema evolution support for PSE database evolution, this now would either require a specialized constraint enforcement to be hard-coded into the PSE system. Clearly, such an approach is a high-overhead strategy, requiring significant support from the underlying DBMS system or major software development on the XML evolution system.

In traditional database management systems such as relational databases (RDBMS) the problem of storing and querying XML data has been widely studied. [MAG$^+$97] did investigate the mapping of semi-structured data into relational databases, while [CACS94] looks at SGML (the predecessor of XML) storage in an object-oriented database management system (OODBMS). [TIHW01a] is one of the first to address the problem of updating hierarchical XML data stored across multiple relational tables. Oracle's XML SQL Utility (XSU) [Ora02] and IBM's DB2 XML Extender [IBM00b] are commercial relational database products extended with XML support. The two mainly provide two choices for managing XML data. The choices are either to store XML data as a blob or to decompose XML data to relational instances. So problems may arise in case of any update to the external data. For the first choice the data needs to be reloaded and for the second choice one has to make changes at the schema level of the relational database. Hence the change propagation from an external XML document to its internal relational storage or schematic structure is not supported in either of the two commercial database systems. Although changes are allowed on the XML data in these systems, it's done typically via some limited interface and the schema is considered fixed and given in advance. This restriction is imposed because schema evolution in traditional DBMS systems tends to be very expensive and disruptive to execute.

2.3 Incremental Validation and Constraint Checks for XML

XML schema design did adopt the form of constraints prevalent in the database literature, however changed the semantics of keys, foreign keys, and unique constraints. [AFL02] demonstrate the costly effect of this slight change on the feasibility of consistency checking. Its shows that even without foreign keys and with very simple DTD features, checking consistency of XML-schema specification is intractable. SAXE instead focuses on a subset of core constraint quantifiers and types checking when the consistency check is feasible. Regarding incremental validation of the XML schema, [PV03] models DTDs as extended context free grammars and the schema as "abstracted specialized DTDs " allowing to decouple types from element tags. From that, it exhibits an algorithm with significant improvement over the brute-force re-validation from scratch algorithm when updates consist of element tag renamings, insertions and deletions. To our knowledge there is no available implementation for this approach even though the theory part has been proven. SAXE [KSR02] and [PV03] have different primary goals, SAXE main purpose is to provide a way of updating while keeping the consistency between an XML schema and its associated XML data, while [PV03] focus on how to discover a best algorithm when altering an element on the DTD or schema.

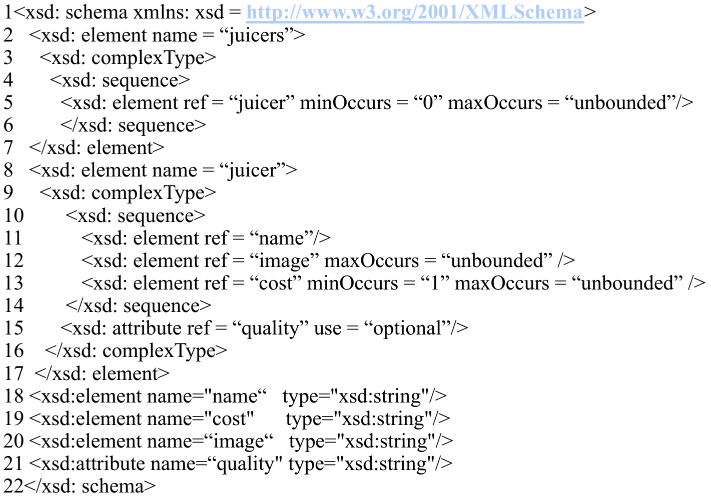
Chapter 3

The XML Update Language

3.1 XML Query Language

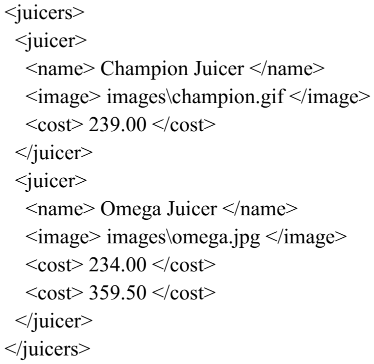
Though it is not required that XML documents must always have an associated schemata due to their ”self-describing” nature, many application domains tend to use some schema specification in either DTD [W3C98] or XML Schema [W3C01a] format to enforce the structure of the XML documents. Whenever XML schemata are associated with the XML data, then structural consistency should also be taken care of during update processing.

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 = “juicers”>
   <xsd: complexType>
     <xsd: sequence>
       <xsd: element ref = “juicer” minOccurs = “0” maxOccurs = “unbounded”/>
       </xsd: sequence>
 </xsd: element>
 <xsd: element name = “juicer”>
   <xsd: complexType>
      <xsd: sequence>
         <xsd: element ref = “name”/>
         <xsd: element ref = “image” maxOccurs = “unbounded” />
         <xsd: element ref = “cost” minOccurs = “1” 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>
```

Figure 3.1: 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>
```

Figure 3.2: Sample XML Document: *juicers.xml*

3.2 Update Language

XQuery [W3C01c] is an XML query language proposed by World Wide Web Consortium for querying XML documents. 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: They serve to bind values or expressions to one or more variables. In particular, `FOR` is used whenever the binding iterates over a list of nodes returned by the expression, while `LET` simply binds the variable to the value of the expression with no iteration.

2. `WHERE` clause (optional): It filters the bindings generated by `FOR` and `LET` clauses by any specified predicates.

3. `RETURN` clause: It constructs an output XML document.

Figure 3.3 gives an example Xquery over the XML document in Figure 3.2. The variable $p iterates over each element node satisfying the expression `document(“juicers.xml”`

```
1 FOR $p in document("juicers.xml")/juicer,
2 $child in $p/cost[1]
3 RETURN $child
```

Figure 3.3: Sample XQuery

) / juicer (line 1). For each identified binding of $\$p$, $\$child$ is bound to the first *cost* child node of $\$p$ (line 2). These cost elements $\$child$ then are returned (line 3).

[TIHW01b] extends XQuery's original FLWR expressions to accommodate the update operations by introducing UPDATE..., clauses i.e., FLWU expressions. We will refer to this language extension of XQuery as the *Update-XQuery* language. The BNF syntax is shown in Figure 3.4 while the BNF for the UPDATE clause (subOp in Figure 3.4) in particular is shown in Figure 3.5.

FOR $\$binding_1$ in *XPath-expr*, ...

LET \$*binding* := *XPath-expr*, ...

WHERE *predicate1*, ...

UPDATE \$*binding* {subOp{, subOp}*}

Figure 3.4: Syntax of Update-XQuery

The semantics of FOR, LET and WHERE clauses are exactly the same as those in a FLWR, while the UPDATE clause specifies a sequence of update operations to be applied on the target nodes identified by FLW.

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}*}

Figure 3.5: BNF of subOp

Chapter 4

Modeling XML Schema, XML Data And Their Interrelationship

4.1 A Verbose Schema

In our current work, we assume the schema is first-class citizen. In this sense, an update to an XML data document is only allowed when the update is *safe*, i.e., the updated data would still conform to the given XML schemata. Without loss of generality we choose to work with a verbose type of XML schema, sometimes referred to as "salami slice design schema" [Cor02]. A verbose schema has a design approach that disassembles instance documents into their individual components.

4.2 XML Schema Modeling

In the schema we first define each component as a separate element declaration, and then assemble them together. Note how the schema in Figure 4.1 declares each component individually (*name*, *image*, *cost* and *quality*) respectively at lines 18, 19, 20 and 21 and

```
<xsd: schema xmlns: xsd = http://www.w3.org/2001/XMLSchema>
 <xsd: element name = “juicers”>
  <xsd: complexType>
   <xsd: sequence>
    <xsd: element ref = “juicer” minOccurs = “0” maxOccurs = “unbounded”/>
    </xsd: sequence>
 </xsd: element>
 <xsd: element name = “juicer”>
  <xsd: complexType>
     <xsd: sequence>
       <xsd: element ref = “name”/>
       <xsd: element ref = “image” maxOccurs = “unbounded” />
       <xsd: element ref = “cost” minOccurs = “1” 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>
```

Figure 4.1: 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>
```

Figure 4.2: Sample XML Document: *juicers.xml*

then assembles them together in the creation of the component *juicer* at line 8. One of the advantages of the verbose schema is its layout structure, the layout enable us to know where to retrieve the needed constraint information in specific places of the XML schema. All components have a global scope and are direct children of the schema root. A verbose schema can be extended to other available schema design styles and vice versa. For example, Figures 4.1 and 4.2 show a verbose XML schema *juicers.xsd* and its XML data *juicers.xml* conforming to the schema. They are used as running samples in this thesis. The XML schema is composed of a root and a set of components, where a component can be of type attribute or element for example in Figure 4.1 at line 8 we have a global component element and at line 21 we have a global component attribute. Each element component can in turn contain attribute declarations or subelement declarations referring to other previously defined components. A component without subelement declarations will be considered as an empty component. Elements of XML data are instances of components from XML schemas.

Structure of an element component: [Cor02] gives a clear description of a global or local component in the verbose type schema. In this thesis the set of all global element components in an XML schema will be referred as eC. Each $c \in eC$ will be denoted by [`e_type, e_name, refEle_Defs, refAttr_Defs`] where `e_type` is the type of the component c, `e_name` is the name of c, `refEle_Defs` is the set of referenced elements, and `refAttr_Defs` is the set of referenced attributes inside the global element. For instance the *juicer* component defined at line 8 in the schema given at Figure 4.1 has for `refEle_Defs` the set of elements defined at lines 11, 12, and 13. The `refAttr_Defs` of the same component *juicer* is composed of one element defined at line 15. When `refEle_Defs` and `refAttr_Defs` are empty as in lines 18, 19, and 20 in Figure 4.1, we are dealing with a global empty component. Any element $r \in$ `refEle_Defs` is denoted by [`refE_type, refE_name, minOc-`

curs, maxOccurs] where refE_type is the type of the referred element component, refE_name is the name of the referred element component, minOccurs and maxOccurs are quantifiers used in the XML schema to specify respectively the minimum and maximum number of occurrences of the referred component may appear in an instance of the referring component. Note that in a schema when neither minOccurs nor maxOccurs are specified for a referred component, their default value is considered to be equal to *1*.

The refEle_Defs of the *juicer* component in Figure 4.1 is made up with the element declarations on lines 11, 12 and 13: refEle_Defs = { <xsd:element ref = *name* > , <xsd:element ref = *image* minOccurs = *unbounded*/>, <xsd:element ref = *cost* minOccurs = 1 maxOccurs = *unbounded*/>}.

The last tuple <xsd:element ref = *cost* minOccurs = 1 maxOccurs = *unbounded*/> refers to the component *cost* defined at line 19 in *juicers.xsd* schema. *cost* is an empty global component, it's type can be found at line 19 defined as a string. XML uses name name tags are type identification for non-empty global component.

Definition 1: A one-to-one function *E* is used to express the relationship between each referred element in a global component g, with g identified by [e_type, e_name, refEle_ Defs, refAttr_Defs]. The function is *E,* given *E* : refEle_Defs $\longrightarrow eC$, where refEle_Defs is the set of referred elements in the given global component g element and eC is the set of all global elements in the schema. We have $\forall\ r \in$ refEle_Defs $\exists\ e \in eC$ such that *E*(r) = e with r.refE_name = e.e_name. And r is assigned refE_type which is equivalent to the type of e (e_type).

refAttr_Defs is the set of referred attributes in a global component element of the schema. An attribute node $ad \in$ refAttr_Defs can be denoted by [refA_type, refA_name, refA_use] where refA_type is the type of the referred attribute, refA_name is the name of the referred attribute, and `refA_use` indicates whether the attribute is *required*, *optional* or even *prohibited*. If a default or fixed value is specified in the attribute declaration then the value `refA_use` must be *optional*. For example, in Figure 4.1 the only element of `refAttr_Defs` in the component *juicer* is at line 15. It has for `refA_name` and `refA_type` respectively *quality* and *string* which are the name and type of the referred attribute. Its `refA_use` value is *optional*.

Structure of an attribute component:

The set of all global attribute components of a schema is referred as aC. The structure of a global attribute component is defined by $[\texttt{a\_type, a\_name}]$ where `a_name` and `a_type` are respectively the name and type of the referenced attribute. In *juicers.xsd* in Figure 4.1 we have only one global attribute component defined at line 21 as $\texttt{a\_name} =$ *quality* and $\texttt{a\_type} =$ *string* .

Definition 2: A one-to-one function *A* is used to express the relationship between each referred attribute in a global component g , with $g = [\texttt{e\_type, e\_name, refEle\_Defs, refAttr\_Defs}]$. The function *A*, given by $A : \texttt{refAttr\_Defs} \longrightarrow aC$ where `refAttr_Defs` is the set of referred attributes in a given global component g and aC the set of all attribute definitions in the schema. $\forall\, adr \in \texttt{refAttr\_Defs}\ \exists\, a \in aC$ such that $adr.\texttt{refA\_name} = a.\texttt{a\_name}$. And *adr* is assigned an identifier `refA_type`, which is the type identified in a (`a_type`).

4.3 XML Data Modeling

An XML document is defined by its name referred to as *xmldoc* and a set N of ordered labeled nodes, where each element node is an instance of one component in its associated XML schema. Every element node has a direct parent. The root is the direct child of the

Schema document.

Structure of an element node. Any element node $n \in N$ is identified by [`type, name, subEles, attrs, value`] where `type` is the type of the element node, `name` is the node tag name, `subEles` is a set composed of the direct children nodes of n, `attrs` is a set containing the attributes of the element node, and `value` the value of the element. An element node with an empty `subEles` is called an empty element or a leaf. A non-leaf element node doesn't have an `value`; thus it could be identified as [`type, name, subEles, attrs,` ϕ]. An element node leaf will obviously have no children so it can be represented as [`type, name,` ϕ`,attrs, value`]. Most of the time the `type` of a leaf is a built-in data type such as integer or string.

In particular, `subEles` can be expressed as the union of sets of sub-elements, each sub-element grouping element nodes of the same type.

$$\texttt{subEles} = \bigcup_{i=0}^{i=k} subEles_i.$$

Each subset $subEles_i$ is characterized by two identifiers: The type and tag name of the elements it is holding. The identifiers will be referred respectively as: $typesubEles_i$ and $tagsubEles_i$.

For an illustrative example, let's take the *juicers.xml* in Figure 4.2. The element node *juicer[2]* has `type` *juicer*, its `name` is also *juicer*, and its set `subEles` is composed of its direct children is: {`<name> Omega Juicer </name>`,

`<image> image/omega.gif</image>`,

`<cost>234.00< / cost>`,

`<cost>359.50</cost>`}.

`subEles` is made of three different subsets, each subset is composed of elements of same name tag. The identifiers $tagsubEles_i$ and $typesubEles_i$ of each subset respectively are

$\{$*cost*, *string*$\}$, $\{$*image*, *string*$\}$ and $\{$*name*, *string*$\}$. Note that if we were dealing with a complexType component the identifier $typesubEles_i$ and $tagsubEles_i$ will be the same due to the fact that XML uses in some cases tag names as type identification. So in this case `subEles` is composed of three subsets enumerated from 0 to 2.

$subEles_0 =$ {`<name> Omega Juicer < /name>`}

$subEles_1 =${`<image > image/omega.gif < /image>`}

$subEles_2 =${`<cost> 234.00</cost> , <cost>359.50</cost>`}

Definition 3: Let n be denoted by [`type, name, subEles, attrs, value`], and

$$sameType : \texttt{subEles} \rightarrow \bigcup_{i=0}^{i=k} subEles_i$$

e a one to one function, where `subEles` the set of direct children of n, and an $subEles_i$ groups element of `(`subEles) having the same type. *sameType(b)* = $subEles_i$ iff b.`name` = $tagsubEles_i$ and b.`type` = $typesubEles_i$

Example: We have:

sameType (`< cost> 234.00 </ cost >`) = $subEles_2$

because `< cost> 234.00 </ cost >` $\in subEles_2$

An attribute $a \in$ attrs identified by a name, type and value [`attr_type, attr_name, attr_value`], where `attr_type`, `attr_name` and `attr_value` are the type, name and value of the attribute respectively.

4.4 Mapping between XML Schema and XML Data

We now present relations that exist between an XML document and its given XML schema. A set of constraints is described in the schema and the XML document should conform to these constraints. In an XML document each element or attribute is uniquely typed, that is each element node is an instance of a unique component node from its associated schema. A mapping function will be used to express the bi-directional relationship between an XML document and its schema. The function is *typeof*, denoted by *typeof:* $N \longrightarrow eC$, where N is the set all element nodes of the XML data and eC is the set of global element components on the XML schema. The *typeof* function ensures that if $n \in N \; \exists \; c \in eC$ such that *typeof(n)* = c.`e_type`

Element Translation Function: Let n be a non - empty element node of N and c be a global element component of eC such that typeof(n)= c.`e_type`. A translation element function *typeofEle* maps the set of direct children of n with the same name tag to the corresponding element declaration in *c*. This means each $subEles_i$ of `subEles` is mapped to the corresponding element in `refEle_Defs`.

typeofEle: `subEles` $\longrightarrow$ `refEle_Defs`, for $r \in$ `refEle_Defs` and *typeofEle*($subEles_i$) = $r \Longleftrightarrow \forall \; b \in subEles_i$ we have: b.`name` = r.`refE_name` and b.`type` = r.`refE_type`.

Illustrative Example: Let n be the *juicer[2]* element node of *juicers.xml* in Figure 4.2,

$subEles_0$ = {`<name> Omega Juicer </name>`};

$subEles_1$ = {`<image > image|omega.gif </image>`};

$subEles_2$ = {`<cost> 234.00</cost> , <cost>359.50< /cost>`}.

The component `<xsd:element ref="cost" minOccurs="1" maxOccurs ="unbounded"/>` defines *cost* in juicer[2].

typeofEle ($subEles_2$) = `<xsd:element ref="cost" minOccurs="1"`
`maxOccurs="unbounded"/>`

where r.`refE_name` is *cost* and r.`refE_type` is *string*.

Attribute Translation Function: Let $n \in N$ be an element and $c \in eC$ be an element component of a schema such that typeof(n) = c.`a_type`. An attribute translation function *typeofAttr* maps each attribute of n to the corresponding attribute declaration (element of `refAttr_Defs`) of the component node c. Let a be an element of $attr$ and ad an element of `refAttr_Defs`, we have *typeofAttr*: $attrs \longrightarrow$ `refAttr_Defs`. With *typeofAttr* (a) $= ad \iff$ a.`attr_name` = ad.`a_name` and a.`attr_type` = ad.`refA_type`.

Chapter 5

Consistency Under XML Data And XML Schema Updates

5.1 Consistency for XML Data Evolution

An XML document is well formed if it meets all specifications of the World Wide Web standard [W3C01b]. A well-formed XML document can in addition be valid if it has an associated schema and if it agree to all constraints expressed in the schema. We now introduce the notion of validity by presenting rules that should hold for an XML data document to be valid with respect to its schema. Each rule is a necessary condition to assure the validity of the XML document. We make the assumption that the XML document and its schema are well-formed before any update is attempted. The set of all element nodes of the XML data document will be referred to as N and the set of all component nodes of the schema will be referred as S.

5.1.1 Rules of Consistency between Data and Schema

Rule 1: Quantifier Constraint Rule

Let c be a component element node from the schema s and n be its instance element in the XML data document node. This rule guarantees the minimum and maximum number of times the referred component in c is allowed to appear as a direct child on the instance n. We already know $\forall\ r \in$ refEle_Defs if $b \in subEles$ and *typeofEle*(b) = r and $b \in subEles_i$ then the translation element function *typeofEle* in Section 4.1 gives us $typeofEle(subEles_i)$=r. So two conditions should hold:

(1.) $\mid subEles_i \mid \geq$ r.minOccurs.

(2.) $\mid subEles_i \mid \leq$ r.maxOccurs.

For example at line 12 in Figure 4.1 the referenced component *image* has for maxOccurs the value *unbounded*. minOccurs is not specified, thus its default value is *1*. Consequently each instance node *juicer* in the XML document *juicer.xml* in Figure 4.2 must have at least one *image* element node as direct child.

Rule 2: Attribute Constraint Rule

If a component element c has an attribute declaration with its refA_use set to *required*, then any instance n of c in the XML data document must have this attribute. If refA_use is set to be optional then the instance node may or may not have the attribute defined. $\forall\ ad \in$ refAttr_Defs one of the following should hold:

1. if ad.refA_use = “required” then $\exists\ a \in$ attrs of n such that *typeofAttr*(a) = ad where *typeofAttr* is the translation attribute function already defined in Section 4.4. This is equivalent to: ad.a_named = a.attr_name and a.attr_value = ad.refA_type.

2. if ad.refA_use = "optional" then we have either (a) or (b) case below

(a) $\forall\ a \in$ attrs, *typeofAttr* (a) $\neq$ad *or*

(*b*) $\exists\ a \in$ attrs such that ad.refA_name = a.attr_name and a.attr_value = ad.refA_type.

Rule 3: Element Node Validity Rule

This rule groups all necessary conditions that an element node in the XML data document must meet in order to be called valid.

$\forall\, n \in N\; \exists\, c \in$ S such that the following conditions must hold:

1. $n.\texttt{type} = c.\texttt{e\_type}$

2. $n.\texttt{name} = c.\texttt{e\_name}$

3. $\forall\; b \in subEles\; \exists\; r \in \texttt{refEle\_Defs}$ such that *typeofEle*(subEles$_i$)=r with $b \in subEles_i$ we have: a) $\mid subEles_i \mid \geq$ r.$\texttt{minOccurs}$ and b) $|$subEles$_i \mid \leq$ r.$\texttt{maxOccurs}$.

4. $\forall\, a \in \texttt{attrs}\; \exists\, ad \in \texttt{refAttr\_Defs}$ such that $a.\texttt{attr\_name} = ad.\texttt{refA\_name}$ and $ad.\texttt{refA\_use} \neq$ ”prohibited”

5. $\forall\, a \in \texttt{attrs}$ if *typeofAttr*(a) = $ad \implies a.\texttt{attr\_type} = ad.\texttt{refA\_type}$. This condition makes sure that the attribute of the element node is of the correct type.

Rule 4: Attribute Node Validity Rule

The rule gives the necessary conditions for an attribute node to be a valid attribute of n.

1. $\forall\; a \in \texttt{attrs}\; \exists\; ad \in \texttt{refAttr\_Defs}$ such that $a.\texttt{attr\_type} = ad.\texttt{refA\_type}$, $a.\texttt{attr\_name} = ad.\texttt{refA\_name}$ where $ad.\texttt{refA\_use} =$ *”required”* or $ad.\texttt{refA\_use}$ $=$ *”optional”*.

2. $\forall\, a \in \texttt{attrs}, \forall$ x $\in \texttt{attrs} - \{a\}\; x.\texttt{attr\_name} \neq$ a.$\texttt{refA\_name}$. This rule prevents to duplicate name tag attributes.

5.1.2 Application of Constraint Rules for the XML Data Update

Changing an XML document is only allowed if it doesn’t violate any of the rules stated above. Our assumption is the XML document and its schema are valid before any update is attempted. Table 5.1 summarizes the rules that may be violated when performing an update. The table classifies the rules that must be checked whenever an update of a certain occur. [1]

Update Operation	Rules to check	Descriptions
delElePassed $child$	Rule1 condition1	Remove child from list target element node
delAtrPassed $child$	Rule2	Delete an attribute node
insElePassed $child$ [before | after (child)]	Rule 3	Insert an element node
insAtrPassed new_attr (n,v)	Rule4	Insert an attribute for an element
renameElt $child$ to	Rule3 and Rule1	Rename the tag name of an element node
renameAtr $child$ to	Rule2 condition1 and Rule4	Rename the tag of an attribute node
replaceE $child$ with c	Rule 1 and Rule 3	Replace an element node
replaceEV $child$ with	Rule 1 and Rule 3	Replace the value of a leaf element node
replaceA $child$ with new_attr(n, v)	Rule 2 and Rule 4	Replace an attribute node
replaceAV $child$ with	Rule 2 and Rule 4	Replace the value of an attribute node

Table 5.1: Constraint Checks Classified By Data Update Types

Example With delElePassed:

We will show how the update operations maintain the system integrity by using the rules above for a delete element operation. `delElePassed` deletes an element node from the XML data document. **Validation:** Let e the target element node to be deleted and p [`type, name, subEles, attrs`] its direct parent node. After the delete we want to make sure that the node p is still valid. We assume every node is valid before an update is tried. The only rule that can affect the validity of the element p, which is an instance of some component c in the XML schema, is Rule 1 condition 1. By e being a child of p then $e \in$ `subEles`, where `subEles` is composed of union of several sets $subEles_i$ (see

[1] More information about the safe data update queries can be found in Appendix A.

Section 4.3), where the set $subEles_i$ groups the direct children of p having the same type element node. Using Definition 3 of section 4.1, $\exists$ $subEles_i$ such that $e \in subEles_i$, the translation function $typeofEle$ gives $typeofEle$ ($subEles_i$) = r , where r is an element referred to in c. From Condition 1 of Rule 1 the XML data document is valid with respect to the schema after the deletion only if $r.\texttt{minOccurs} \leq |\ subEles_i\ | \ - 1$.

We illustrate how the constraint checking is done for the delete element node in our running XML data document example using XQuery. The `schemaChkDelEle` in Figure 6.4 queries the schema 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 as direct child e's parent. `schemaChkDelEle` retrieves the minimum occurrence of elements $*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*.

```
Function schemaChkDelEle($xsdName, $parentEleName,
$childEleName)
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
6 }
```

Figure 5.1: 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 }
```

Figure 5.2: Constraint Checking Function *delElePassed*

Figure 6.2 shows the rewritten Update-XQuery from the Update-XQuery in Figure 6.1. 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 6.2 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.

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

Figure 5.3: 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 }
```

Figure 5.4: Sample Safe Update-XQuery

5.2 Consistency for XML Schema Evolution

XML schema supports a variety of atomic types (e.g., string, integer, float, double, byte), complex type constructs (e.g., sequence and choice) and inheritance mechanisms (e.g., extension and restriction). The description of XML schema did not adopt all the constraints prevalent in the database literature. While consistency checking of an XML-Schema specification is intractable in certain cases [AFL02], checks for the SAXE system focus on constraint checks concerning type checking, component validity checking, attribute and element validity, and particle validity such as the quantifier `minOccurs` and `maxOccurs` values. The reasoning about constraint validation in SAXE is based on incremental constraint checks, and an update is allowed on the schema only if it leaves both the XML schema and the XML data both valid and conforming to each other. A

[2] line 6 is added only to meet the syntax requirement.

well formed XML schema meets all the specifications of the World Wide Web consortium [W3C01a] specification and all associated XML data should conform to constraints on the schema. As for data consistency we introduce rules that will guarantee the notion of validity for the evolving XML schema. Each rule is a necessary condition of validity of the XML schema. We also make the assumption that the schema is valid before any attempt of on altering the schema is made.

5.2.1 Constraint Rules for XML Schema

We present in this section rules that guarantee an XML schema is valid after modification. The rules are based on the specifications of the World Wide Web standard [W3C01b]. The structure of the schema used is verbose [Cor02], meaning that in such schema style the direct or immediate children of the schema root are referred to as global elements and local when they are nested in another component. In Figure 4.1 the root is element `<xsd:schema>`.

Rule 5: Syntax Rule

This rule guarantees that element and datatypes used to construct schemas originated from the namespace. SAXE update operations can manipulate safely any element belonging to the targeted namespace. The annotations used to construct SAXE schemas are: `schema`, `element`, `attribute`, `complexType`, and `sequence`, this can be confirmed by looking at any element on the schema *juicers.xsd* in Figure 4.1 where the namespace variables used are: { `xsd:schema`, `xsd:element`, `xsd:attribute`, `xsd:complexType`, and `xsd:sequence` } . Each of the elements of the schema has a prefix "`xsd:`" which is associated with the namespace through the declaration. Below is the list of namespace variables that can be manipulate by SAXE:

1. `xsd:element`

2. `xsd:attribute`

3. `xsd:complexType`

4. `xsd:sequence`

Rule 6: Position Rule

This rule guarantees that any component node, be it a global node or a local node, is in a proper position within the schema. A schema satisfies the *Position Rule* if the following conditions hold:

1. Let r be referred element in a component of the schema with namespace `xsd:element`, having `ref` and a value `refE_name` as attribute, then $\exists\ c \in eC$ (the set of global components of the schema) such that $r \in c.$`refEle_Defs`. c is a non-empty global element node.

   This condition ensures that a referred element node with namespace `xsd:element` is in a proper position with respect to the schema. A referred node should be positioned as a local element of the schema, e.g., if we look at the *juicers.xsd* schema in Figure 4.1 the referred elements at lines 11, 12, 13 are local elements having as immediate parent `<sequence/>`, which in turn has for immediate parent `<complexType/>`.

2. Let r be an element of the schema, having `ref` and value `refA_name` as attribute then $\exists\ c \in eC$ such that $r \in c.$`refAttr_Defs`. c is a non-empty global element node.
   This condition ensures that a referred node with namespace `xsd:attribute` is a local element of the schema, it has as immediate parent the element `<complexType/>`.

In Figure 4.1 the non-global component *juicer* refers in line 15 to *quality*, the referred element inn line 15 has as immediate parent `<complexType/>`.

3. Let c be an element of the schema with an attribute `name` and a value `e_name`, c be an immediate child of the root `<schema/>`. This condition guarantees the position of a global element in a verbose style schema which SAXE uses. For example, lines 2, 8, 18, 19, 20 and 21 are immediate children of the root `<schema />` in *juicers.xsd* Figure 4.1. They have all an attribute `name` identifying them.

4. An element `<xsd:complextype/>` of the schema is a local component and, has for immediate parent a component name definition and for immediate child the node `<sequence/>`.

   For instance all the elements of the schema in Figure 4.1 defined with `xsd:complexType` have for immediate children `<sequence/>`. For example lines 3 and 9 have respectively lines 4 and 10 (both corresponding to `<sequence/>`) as immediate child. The immediate parent of lines 3 and 9 are respectively lines 2 and 8; both elements having an attribute value `name`.

5. An element `<xsd:sequence/>` is a local element of the schema and has as immediate parent the element `<xsd:complexType/>`.

6. An element with namespace `<xsd:element>` and having `ref` as attribute has for immediate parent the element `<xsd:sequence>`. In Figure 4.1 we have the elements at lines 11, 12, 13 which all have as immediate parent the element `<sequence/>` at line 10.

7. An element with namespace `<xsd:attribute>` and having `ref` as an attribute has for immediate parent `<xsd:complexType>`. Line 15 in Figure 4.1 has as

immediate parent `<xsd:complexType>` at line 9.

Rule 7: Referred Element Rule

This rule guarantees the correctness of a referred element on the schema. $\forall\, r \in$ `refEle_Defs` defined by [`refE_type, refE_name, minOccurs, maxOccurs`] then the following condition should hold:

1. `refE_name` $\neq$ empty string.

   This guarantees the value of *ref* attribute for the element r not being null. In Figure 4.1 we see that each referred element has *ref* and its value assigned.

2. $\exists\, e \in eC$ / r.`refE_name` = e.`e_name`.

   For instance in Figure 4.1 any referred node is an existing defined global element node of the schema. For example lines 11, 12, 13 are all defined in line 18, 19 and 20 respectively.

3. The *minOccurs* and *maxOccurs* attribute and their value are not required, when declared their value has to be set to a positive integer.

Rule 8: Referred Attribute Rule

This rule guarantees the well-formeness of a referred attribute node in the schema.

$\forall\, ar \in$ `refAttr_Defs` defined by [`refA_type, refA_name, refA_use`]

if ar is referred locally in a component element node then it is valid if the following conditions hold:

1. `refA_name` $\neq$ empty string. with `refA_name` being the value of `ref`.

   For instance in Figure 4.1 at line 15 the *ref* value is not null.

2. $\exists$ $at \in aC$ (the set attribute component of the schema) such that ar.refE_name = at.a_name

   For instance in Figure 4.1 the referred quality component has been already defined by the global component node in line 21.

3. use with its assigned value refA_use are not required, but when declared the value should not be null. The possible values of use we can manipulate accordingly in SAXE: are optional, required and prohibited.

Rule 9: Element Component Validity Rule

This rule guarantees the validity of a global element.

Let $c \in eC$, where $c =$ [e_type, e_name, refEle_Defs, refAttr_Defs]. Then c is well-formed and valid if the following conditions are satisfied:

1. $\forall$ $c \in eC$, $\neg$ $x \in eC$ such that c.e_name = x.e_name.

   This guarantees that no duplication of global component names occurs.

2. xsd:element is the namespace used for declaring c. For instance in Figure 4.1 xsd:element is used for the declaration of *juicer* and *cost* at lines 8 and 18 respectively.

3. If c is of complex type, then name is the only attribute for the element node declaration. For example line 8 of *juicers.xsd* schema in Figure 4.1 the component has for name *juicer*.

4. If c is not declared as complex type, then refEle_Defs $= \emptyset$, refAttr_Defs $= \emptyset$, and the name with value *e_name* $\neq$ empty string.

   The second possible attribute that can be assigned to c is *type* with value *e_type*. For instance lines 18, 19 and 20 in Figure 4.1 the *juicers.xsd* schema.

5. If `refEle_Defs` $\neq \emptyset$ then we have the requirement below:

   (a) An element `<xsd:complexType/>` is an immediate child of an element defined with the namespace `<xsd:element/>` having one attribute which is `name`, e.g., line 9 of *juicers.xsd* schema in Figure 4.1.

   (b) An element `<xsd:sequence/>` must be an immediate child of `<xsd:complexType/ >`. For example line 10 of *juicers.xsd* schema in Figure 4.1.

   (c) Elements belonging to the set `refEle_Defs` must have as immediate parent `<xsd:sequence/ >`. For example the referred element on the `juicer` component in lines 11, 12 and 13 in Figure 4.1 have as immediate parent `<xsd:sequence/>` at line 10.

6. If `refAttr_Defs` $\neq \emptyset$ then we have:

   (a) The element `<xsd:complexType/>` is immediate child of the element name declaration, e.g., line 9 of *juicers.xsd* schema in Figure 4.1.

   (b) $\forall\, r \in$ `refAttr_Defs`, *r* has as immediate parent `<xsd:complexType/>`, e.g., the referred element in the `juicer` component at line 15 in Figure 4.1 has for immediate parent `<xsd:complexType/>` at line 10.

Rule 10: Attribute Component Validity Rule

This rule guarantees the validity of the element attribute declaration $c \in aC$.

Let c be identified with [`a_type, a_name`], c is well formed and valid if the following conditions hold:

1. $\forall\ c \in aC$, $\nexists\ x \in aC$ such c.`a_name` = x.`a_name`. This guarantees that global component names are not duplicates.

2. The namespace used for the component attribute declaration is `xsd:attribute`. For example in Figure 4.1 its *quanlity* component at line 21.

5.2.2 Application of Constraint Rules for Schema Update

Changing an XML schema is only allowed if it leaves the schema and XML data valid and conforming to each other.

As we did when we were dealing with changes that originated from the XML data, the purpose here also is to generate a safe update query from an XQuery statement written with the intent of altering the schema. Contrary to data updates where only the XML data is altered during the process, updating the XML schema may result in altering both the XML schema and the XML data. The safe XQuery or XQueries generated should leave the XML documents in a consistent state. Depending on the type of updates, the safe queries embed the rules that should hold, and ensure an update will not violate the validity of the schema nor the XML data. Table 5.2 summarizes the rules that should hold whenever an update's intent is to alter the schema.

Application With rep_Glo_EleName:

The purpose of this example is to show how the system integrity is kept during the schema modification. `rep_Glo_EleName` is a query that is written with an intent of replacing the name of a global component element with namespace `xsd:element`. An example will be to replace the name of the component *smalljuicer* at line 19 by another name such as *newname* in Figure 7.1 located in Section 7.1. This update case is equivalent of changing the type of a non-empty component from the schema. **Validation:** Let $e \in eC$ be the target element node, with `e_name` *oldname*, and we want to replace `e_name` value with *newname*. The aim is that after alteration, the modified component e and the schema must be valid. We assume that the schema is valid before any update is tried. The changes here affect e, precisely e.e_name, so any component of the schema that depends on e need to be adjusted. Also the instances of e in the XML data need to be updated. Let $e =$ [`e_type, e_name, refEle_Defs, refAttr_Defs`]. Before the update we have

Query Update	Rules To Check	Comments
del_Glo_Ele	Rule 7 (2) ; Rule 3 (3)	Query deletes global component with xsd:element as namespace
del_Glo_Atr	Rule 8 (2) ; Rule 4 (1)	Query deletes global component with xsd:attribute as namespace
del_Ref_Ele	Rule 6 (6) ; Rule 3 (3)	Query deletes referred component with xsd:element as namespace
del_Ref_Atr	Rule 6 (7) ; Rule 4 (1)	Query deletes referred component with xsd:attribute as namespace
del_min	Rule 7 (3) ; Rule 1 (1)	Query deletes minOccurs
del_max	Rule 7 (3) ; Rule 2 (2)	Query deletes maxOccurs
del_use	Rule 8 (3) ; Rule 4 (1)	Query deletes use
ins_eltName	Rule 5 (1) ; Rule 6 (3)	Query inserts an element with `name` attribute as <xsd:element name =”value”>
ins_attrName	Rule 10 (2) Rule 5 (2); Rule 6 (3)	Query inserts an element with `name` attribute as <xsd:attribute name =”value”>
ins_compType	Rule 5 (3) , Rule 6 (4)	Query inserts <xsd:complexTye>
ins_seq	Rule 5 (4) , Rule 9 (5 b)	Query inserts <xsd:sequence>
ins_type	Rule 9 (4)	Query inserts `type` for <xsd:element name =”aname” type=”atype”>
ins_ref_Ele	Rule 7 (1,2) ; Rule 3	Query inserts referred component <xsd:element ref =”aname”/ >
ins_ref_Atr	Rule 8 (1), 2; Rule 4	Query inserts referred component <xsd:attribute ref=”aname”/ >
ins_Min	Rule 7 (3) ; Rule 1 (1)	Query inserts minOccurs
ins_Max	Rule 7 (3) ; Rule 1 (2)	Query inserts maxOccurs
ins_use	Rule 8 (3) ; Rule 4 (1)	Query inserts use
rep_ref_Ele	Rule 7 ; Rule 3 (1,2,3)	Query replaces referred component
rep_ref_Atr	Rule 8 ; Rule 4	Query replaces referred component
rep_Glo_EleName	Rule 6(3),Rule 9(1),Rule 7(2);Rule 3(3)	Query replaces the name of a global component namespace xsd:element [3]
rep_Glo_AtrName	Rule 10 (1)Rule 9 (6 b); Rule 4	Query replaces the name of a global component namespace xsd:attribute

Table 5.2: Constraint Checks Classified By Schema Update Types

e.`e_name` = *oldname*, after the update we want e.`e_name` = *newname*.

Changing e.`e_name` will not violate the *Syntax Rule* because changes for this evolution case do not affect the namespace of the component. Before starting the update process one should determine that the node targeted for this name change is in fact a global component in the schema. This is done by checking *condition 3* of the *Position Rule*. With e being a global component element of the schema, updating the name of e will not have an impact on referred attributes of the schema. So there is no need for checking *Referred Attribute Rule* conditions. After alteration e should be a valid component [`e_type, e_name, refEle_Defs, refAttr_Defs`]. Here we are modifying e.`e_name`, *condition 1* of *Element Component Validity Rule* guarantees the non duplication of the global component in the schema so this should be considered during this update process. Another important fact to verify is *condition 2* of the *Referred Element Rule*, the condition guarantees component referring to the altered node are still valid. To check the validity of the instances of the target node e in the XML data, *condition 3* of *Element Node Validity Rule* should be checked.

We illustrate how the rule checks are done using XQuery; our running example in Figure 5.5 is a query written with the intent of renaming the component *smalljuicer* at line 19 in Figure 7.1. For such a query two safe queries will be generated (see Figure 5.6 for the templates of the generated queries). One query is used to update the XML schema safely and the second one is used for the XML data update.

```
For $p in document(“juicers.xsd”)/xsd:element[@name=smalljuicer]
Let $childatr := $p/@name
 Update $p {
            replace $childatr with “newName″
 }
```

Figure 5.5: Sample XQuery For Rename Component Name

The original query is rewritten by inserting the necessary checks in order of allowing a

```
For $p in document(”juicers.xsd”)/xsd:element[@name=smalljuicer]

Let $childatr := $p/@name
          Insert Schema Checks

          Update $p {
          If Conditions Satisfied Update

          replace $childatr with “newName″

          }

  Query For Data Updates
```

Figure 5.6: Template For the generated Queries

safe update on the schema. And a new query is generated for the data updates. In the table 5.2, for `rep_Glo_EleName` one needs to check the *Rule 6(3), Rule 9(1), Rule 7(2)* and *Rule 3(3)*. We already went over them on the validation part above. They are translated into XQuery in Figure 5.7. Checks for *Rule 6(3)* which is *Position Rule Condition 3* are in line 3. The statement tries to find if the the root of the XML schema is a direct parent *smalljuicer* component to be renamed. The retrieved information is stored in *$pval*. $pval helps identify if the name to be renamed is a global component node. If it is not the case the update will not be allowed. This can be found at line 7 of the same figure. Line 5 queries information that guarantees *Rule 9(1)* is equivalent to *condition 1* of the *Element Component Validity Rule*. In case there is already in the schema a global component name with *newname* it is stored in *$child*. And if *$child* is empty then line 7 of Figure 5.2 shows the update will not proceed because we don't want duplicate global components in the schema. At line 9 in the same figure the XQuery function *replaceRefSche* makes sure that the *Rule 7 (2) condition 2* of *Referred Element Rule* holds. It will update any element of the schema referring to *smalljuicer* to now refer to *newName*. For more information concerning *replaceRefSche* see Figure 5.8. An XQuery is also generated for the XML data. The following paragraph talks about it in details.

```
For $p in document(“juicers.xsd”)/xsd:element[@name=smalljuicer]
Let $childatr := $p/@name
            Let $pval := $p/parent:: ∗/@xmlns:xsd,
            $cntpval := count($pval),
            $child := $p/parent::*/xsd:element[@name=newName”],
            $childexist := count($child),
            Where $cntpval = ”1” and $childexist =”0”
update $p {
            Where replaceRefSche(”juicers.xsd”, ”smalljuicer”)
             update $p {
                replace $childatr with “newName″
             }
 }
```

Figure 5.7: Safe XQuery Generated For Schema Updates

The XQuery statement in Figure 5.9 updates the XML data. Line 1 stores all the instances of type *smalljuicer* in *$child*. The instances are then updated to *newName* at line 4.

```
Function replaceRefSche(“juicers.xsd” , “smalljuicer”, ”newName”)
{
    For $pref in document($schemadoc)//xsd:element[@ref=$childBindRefName],
        $rRef in $pref/@ref
       update $pref{
                replace $rRef with newName
        }
}
```

Figure 5.8: Function For Schema Updates *replaceRefSche*

```
For $child in document(“juicers.xml”)//smalljuicer
   let p :=child/parent::*
  Update $p {
           rename $child to ”newName”
   }
}
```

Figure 5.9: Safe Query Generated For Data Updates

5.2.3 Semantic Restriction for Safe Schema Update Generations

In the spirit of propagating schema changes to the XML data, the SAXE framework imposes semantic restrictions for certain schema update XQuery statements. Restriction applies to queries that manipulate the constraints *minOccurs*, *maxOccurs* and *ref* . Such queries require the *XPath-expr* of $binding_1$ in Figure 3.4 to specify the name of the complexType component where the modification will happen, example the binding $*psch* in Figure 5.10 has the component name *juicer* specified in its path. Mainly we need the name of the component here in order to update the instances of *juicer* in the XML data.

```
For $psch in document(”juicers.xsd”)/xsd:element[@name=”juicer”]
                          /xsd:complexType/xsd:sequence,
 $ch in $psch/xsd:element[@ref=”cost”]
  update $psch{
       delete $ch
  }
```

Figure 5.10: Sample of Query to delete a referred element from the Schema

Chapter 6

SAXE Framework

6.1 Generation of Safe Update Queries

Our Overall Approach For Safe Query Rewriting. In order to allow only consistent updates to be processed on XML data or XML schema, we aim to develop a loosely-coupled update strategy that supports incremental schema constraint checking by accessing only minimal parts of the XML documents needed to perform the checks. The key idea is to first generate a safe Update-XQuery statement from a given input Update-XQuery statement. The generated safe Update-XQuery statement, still will be conform to the standard Update-XQuery BNF and thus can be safely executed on any xQuery update engine. In this way we succeed in separating the concern of constraint check verification from the development of the XML query and update engine.

For the safe query generation, when changes originate from the XML data we have to first analyze all update operations supported by the Update-XQuery language in order to design appropriate *constraint checking subqueries*. When changes originate from the XML schema, then besides analyzing Update-XQuery operations one should also consider the type of changes allowed on an XML schema in order to design appropriate *constraint check subqueries*. The constraint check subqueries take the input parameters from the update operation and determine whether the update operation is valid or not. For the safe query, we exploit the capability 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. The execution of an update operation is conditional on passing the constraint checking.

Illustrating Example. This example illustrates how the constraint checks are inserted into an original query with the intent to update the XML data. Figure 6.2 shows the rewritten Update-XQuery from the Update-XQuery in Figure 6.1. There is one update operation in Figure 6.1, i.e., `DELETE $c` in line 4. We can see that lines 3, 5 and 6 in Figure 6.2 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 3.

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

Figure 6.1: 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 }
```

Figure 6.2: Sample Safe Update-XQuery

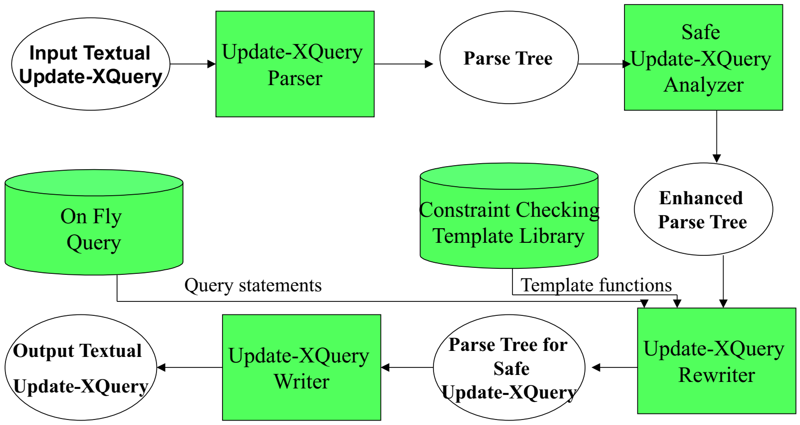

Input Textual Update-XQuery
Update-XQuery Parser
Parse Tree
Safe Update-XQuery Analyzer
Enhanced Parse Tree
On Fly Query
Constraint Checking Template Library
Query statements
Template functions
Update-XQuery Rewriter
Parse Tree for Safe Update-XQuery
Update-XQuery Writer
Output Textual Update-XQuery

Figure 6.3: An Incremental Yet Loosely-Coupled Update Processing Framework Supporting XML Updates with Schema Constraint Validation

6.2 Framework

Figure 6.3 is the final design of SAXE [KSR02] framework, it generates safe Update-XQuery statement given an input Update-XQuery. The *safe Update-XQuery generator SAXE* is composed of the components described below:

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.

3. *Constraint Checking Template Library.* We generalize the constraint checking procedures by defining named parameterized XQuery functions called *constraint check templates*. Each constraint checking template is in charge of checking constraints for a certain type of update operation.

4. *Constraint Checking Fragments.* Each schema update type is associated with a set of constraints checks translated into XQuery statements, parameterized and stored.

In case the update does not require a data update then one will have only one set of query statements, these statements are inserted into the original XQuery transforming it into a safe XQuery intended for the schema updates. When the update case requires data changes, a second set of query statements will be generated on the fly (right away), for a safe XML data update.

5. *Update-XQuery Rewriter.* The rewriter handles the actual generation of a safe Update-XQuery. Based on the parse tree generated by the Update-XQuery parser, 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.

6. *Update-XQuery Writer*. The writer constructs a textual format of the modified Update-XQuery statement from the enhanced parse tree, which now is in a standard Update-XQuery format. This can be executed be any update query system.

6.3 Components of Constraint Checking Framework

We now describe the main components of the framework shown in Figure 6.3. We do not describe the *Update-XQuery Writer* since it is straightforward.

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.

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 if any. For example, the analyzer would examine the content of an object of the 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*.

3. *Constraint Checking Template Library*

   The template library functions are used for safe query generation when changes originate from the XML data. The library stores templates that account for every type of update possible using our Update-XQuery language (See BNF in Figure 3.4). A constraint check is composed of three steps which are:

   (a) Query the XML schema to identify any constraints that may be violated by the specified update.

   (b) Query the XML document to gather information pertaining to the target elements or attributes.

   (c) 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 the delete of an element operation in order to change an XML data. The constraint check functions `schemaChkDelEle` and `delElePassed` shown in Figures 6.4 and 6.5 jointly achieve the three steps mentioned above.

   The Constraint Checking Function `schemaChkDelEle` queries 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

```
Function schemaChkDelEle($xsdName, $parentEleName, $childEleName)

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
6  }
```

Figure 6.4: 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 }
```

Figure 6.5: Constraint Checking Function*delElePassed*

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:

(a) **Query over data** (i.e., step 2). Line 2 queries 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*. The 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.

(b) **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.

4. *Constraint Checking Fragments.*

The *On Fly Queries* (Safe Queries generated for the XML schema and XML data updates when changes originate from the XML schema) accounts for all constraint checks translated into XQuery statements for each possible update targeted during within SAXE, on XML schemas. A safe update on the schema is obtained by:

(a) Query XML schema to identify pertaining information that may be violated when the update is applied.

(b) Update schema accordingly if operation allowed.

(c) Update XML data documents to reflect changes of the schema pertaining to targeted elements and attributes.

5. *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 each template is parameterized, the Rewriter would also instantiate 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 6.2. The `delElePassed` template function takes in five parameters to execute its query. For this particular example, *$c* (the element instance to be deleted), *$p/cost*, "*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 five 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 prior to the associated update clause (as shown by the example in Figure 6.2). After all modifications have been done to the original update XQuery, the safe XQuery generation is complete. Finally, a resulting safe update XQuery statement is produced.

6.4 Discussion of SAXE System Implementation

SAXE 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 textual Quilt statement and constructs a parse tree if the syntax of the statement is correct. The evaluator then executes the query against the data. First, we have 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. The Kweelt System was also extended to support the generation of safe Xqueries. The safe XQuery support was designed to be independent from the update executor in order to allow the update executor to handle both safe and non safe update XQueries.

Chapter 7

SAXE Experiments

7.1 Experimental Setup

Introduction

Experiments were conducted to evaluate SAXE system. The purpose of this section is to report on these experiments on the SAXE system, study their results and come out with a solid understanding of SAXE performance. It is important also to identify cases where the safe update operations efficiency will outperform native XML update tools available for updating XML documents. Each experiment was run ten times, and the result used is the average over ten runs. In general the main measures under consideration are the time spend in (ms) to achieve an action and the number of items modified in the XML schema or XML data documents. The size of the XML files may vary, depending the experiment being performed. In order to make the results coherent and easier to follow we decided to carry XML schema `juicers.xsd` in Figure 7.1 and the XML data `juicers.xml` in Figure 7.2 as files to modify accordingly for our tests purposes. The schema file `juicers.xsd` has root the element <*schema*> on the top of the file. It contains also three components that can be instantiated: *juicers*, *juicer* and *smalljuicer*,

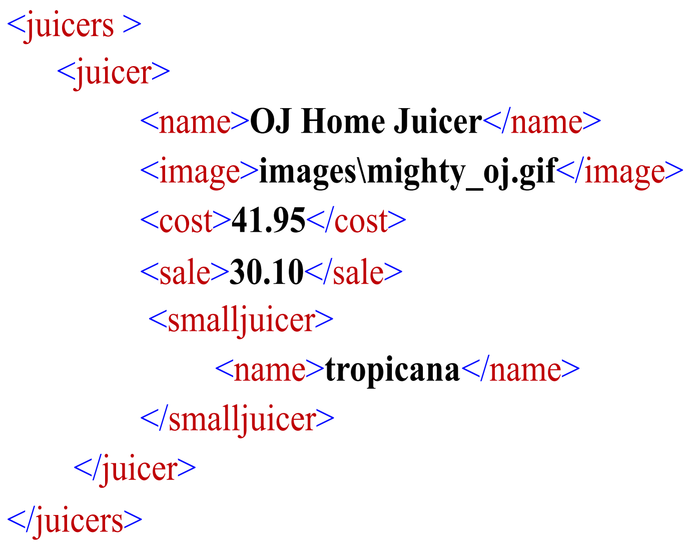
```
<xsd: schema xmlns: xsd = http://www.w3.org/2001/XMLSchema>
  <xsd: element name = “juicers”>
    <xsd: complexType>
      <xsd: sequence>
        <xsd: element ref = “juicer” minOccurs = “0” maxOccus = “unbounded”/>
        </xsd: sequence>
      </xsd: element>
   <xsd: element name = “juicer”>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

Figure 7.1: Sample XML Schema:*juicers.xsd*

it has component definitions which are the following nodes: *name*, *image*, *cost*, *sale* and *percentage*. The original data file `juicers.xml` has one instance of component composed of five sub-nodes. The schema and XML data documents are altered accordingly by adding more elements nodes, components nodes, instance nodes, etc.

Execution Platform

All experiments were performed on the same machine in order to allow proper comparisons. The platform execution is Microsoft Window XP. The processor is an ADM Duron 700 MHz. The total of memory on the machine is 128 Megabytes. An attempt to minimize the influence of other process times on the result, all applications were closed



```
<juicers >
    <juicer>
            <name>OJ Home Juicer</name>
            <image>images\mighty_oj.gif</image>
            <cost>41.95</cost>
            <sale>30.10</sale>
             <smalljuicer>
                   <name>tropicana</name>
            </smalljuicer>
      </juicer>
</juicers>
```

Figure 7.2: Sample XML Document: *juicers.xml*

during the experiments, with the exception of the DOS command window used to run the experiments.

Statistics Data Sets Used

While testing the safe update queries the following elements and constraints can be manipulated in the XML schema documents:

- The number of elements and/or attribute definitions in the XML schema documents
- The number of components from the schema that can have an instance (instantiable) in the XML document files
- The number of referred nodes in a component
- The total number of elements of the schema

- The number of <xsd:complexType > constant elements used in the XML documents

- The number of <xsd:sequence > constant name elements used in the XML schema

- The constraint `type` which is an attribute of the element or attribute definition

- The constraint `minOccurs` which is an attribute of a referred node element

- The constraint `maxOccurs` which is an attribute of a referred node element

- The constraint `ref` which is an attribute of a referred node

- The smallest XML schema file used while running the experiments is the original source *juicers.xsd* in Figure 7.1. The file is made of 3 components ”instantiable”, meaning they can have instances on a XML data document, those components are *juicers*, *juicer* (which refers to five element node) and *smallerjuicer*. The file holds also component definitions which are: *name*, *image*, *cost*, *sale* and *percentage*. The original schema file has root <*schema* > on the top is the root and the number total of nodes is 22.

- The number of element in a largest XML schema has 42 ”instantiables” component nodes, component definitions. The files is made off a total of 374 nodes

For the XML data documents the following changes can be done:

- Altering an element node such as delete, add, and modify

- Altering an attribute of an element node

- Altering the number of element nodes of the XML document

- The smallest XML data files used during the experiment is the `juicers.xml` in Figure 7.2 which has a total of 7 element nodes. The XML data file is composed of 2 instances nodes which are *juicer* and *smalljuicer* where *smalljuicer* is a child of *juicer*.
- The number of elements in a largest XML files is composed of 20,000 instances, with a total of 60,000 elements.

7.2 Experimental Results

All the experiments below were executed on a schema file obtained by altering the original `juicers.xsd` in Figure 7.1 and the original `juicers.xml` in Figure 7.2. Alteration was done done by adding, deleting and modifying elements and attributes of the XML documents. The time factor is important in evaluating the performance, so in most experiments the time variation is plotted as a function of the items of the schema or XML data when a change is using SAXE system.

7.2.1 Safe Updates Queries Generation Time

One of the SAXE system goals was to be a middleware portable in any database system. In that sense the *Query Rewriter* part is set to be independent of any other influences other that the text query input being transformed. This test shows that XML document (such as the size or the number of elements to modify) does not affect the query generation time. So the purpose of this test is to have an approximate idea of how long it would take to generated the safe query for the sample of queries used in our example. The results obtained from the evaluation have two types of metrics collected, namely the time spent parsing the original query and the time spend generating the safe queries. The timer for the generation of the safe query starts when the parser is invoked, and stops when the safe

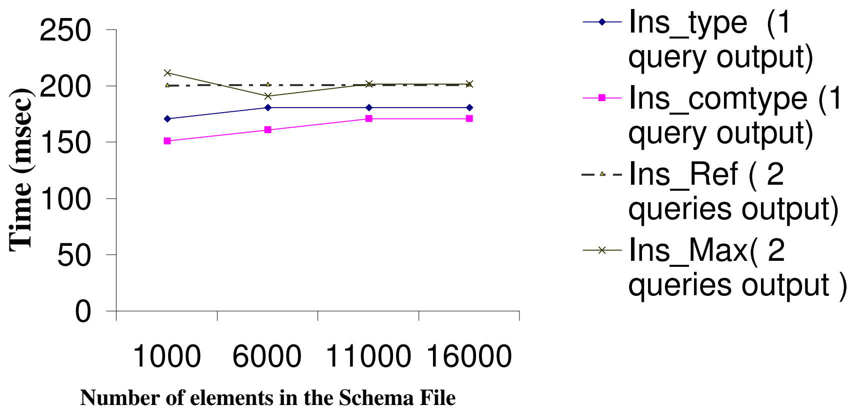


Figure 7.3: Time for Query Generation

queries has been written to the output files. Tests were performed to study the variation times for the safe update query generations when the size of the XML schema documents involved are increased by 5000 new elements after each test. Four different types of queries were used to study the safe queries generation time here. They are `Ins_type`, `ins_compType`, `ins_Max`, and `ins_Ref` [1]. Each query was run ten times and the results averaged to have the time it takes to generate a query. Safe Queries were generated based on the analysis of the original query statement rewritten to alter the XML schema. As expected the chart lines of the graph in Figure 7.3 are almost a straight line, which shows the XML documents do not influence the query generation time. The sample of queries used can be divided into two different groups, queries that give one safe query

[1] The four example of queries can be found in Appendix B.

as result (`ins_type` and `ins_compType`) and queries that give two queries as safe update queries (`ins_Ref` and `ins_Max`). We note a little more time is spent generating a safe query from `ins_type` than generating a safe query from `ins_compType` even though the two queries have almost the same size. This time difference is mainly due to the difference of checks that need to be added to each query in order to make it safe. `ins_type` requires more checks to be added to the original query to produce a new safe query. The queries for which two queries are generated as a result the time to construct with the safe queries is almost the same. This is due to the fact both query samples `ins_Ref` and `ins_Max` have almost the same size and it takes the same process to come out with necessary safe checks for rewriting the safe queries. Also for the sample queries used here, we notice a time difference between the time it takes to generate one query as safe query or two queries as safe update.

7.2.2 Analysis of Replace a Component Name (Type Change)

The purpose of this test is to study the time efficiency for the safe update when replacing the name of a component. During the test we used variable sizes of XML schemas and XML data files. Eight different tests were run on the schema shown in Figure 7.1 and four different tests were run on the XML data in Figure 7.2. The result is plotted in Figure 7.5. In both experiments the number of modified elements during each test increases proportionally with the size increase of the files. The results represent the average over ten runs. Considering the fact that XML uses tag names sometimes as type identification, replacing a component name in the schema is the same as a type change in this case. A component type changed in the schema should lead to changes in the XML data document. If a query is written with the intent of replacing the name of a complexType component node then two queries will be generated called `safe_rep_name_xsd` and `safe_rep_name_xml`. We measure the time it takes to update the schema file by exe-

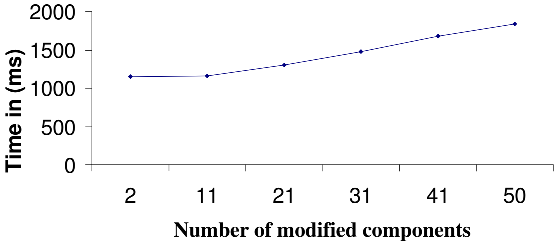


Figure 7.4: Execution Time For Rename a Component Node

cuting `safe_rep_name_xsd`. The result is presented in Figure 7.4. The time to update the XML data document was measured by executing `safe_rep_name_xml` on the XML data. The result is plotted in Figure 7.5.

In Figure 7.4 the execution time is graphed as a function of the number of component altered during the operation. At each test the number of altered nodes was increased. We end up with a linear complexity time when the number of modified components vary. Note that updating multiple of element nodes at the same time update will be more beneficial than a small number of updates. The line in Figure 7.4 shows that one update takes 1151 msec whereas 10 updates take 1162 msec and 20 updates take 1302 msec. We found again the same trend when `safe_rep_name_xml` is executed on a XML data (the number of modified elements is increased during each test on x-axis). We see this same linear time complexity for the data updates in Figure 7.5.

Figure 7.6 compares the execution time of the query `safe_rep_name_xml` to the execution of the query `safe_rep_name_xsd`. The results were obtained by running the

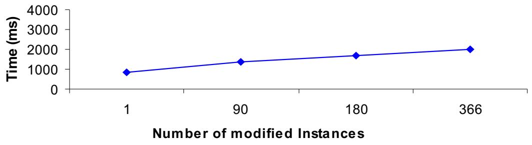


Figure 7.5: Execution Time for Rename an Instance Name in an XML Data

queries on the XML documents where the number of altered components in the schema is the same as the number of altered elements in the XML data. That is, the file size growth is kept proportional during each test. Here the time for updating the schema is more expensive than the time it takes to update the XML data. This is due to the difference of method retrievals between the two update queries. The retrieval time between the variants are reduced when we have multiple matches accessed once [FG02] on the query `safe_rep_name_xml` we were able to drop most of the information structure of the path and also access the elements to alter all at once, This way the execution time of `safe_rep_name_xml` is cheaper. We can conclude also that changing the type of a component for the sample of queries used in SAXE is linear when the number of modified elements is being increased.

7.2.3 Comparing Updates Generating Only Schema Changes

Now let's examine some pure schema updates, those do not involve any data updates. The safe queries chosen for this test are: `safe_ins_name`, `safe_ins_type`, `safe_ins_compType` and `safe_ins_sequence`. `safe_ins_name` is a safe query used for inserting *name* as its value in a component definition of the schema. As result we will have `<xsd:element name="givenValue">`. `safe_ins_type` alters the element to `<xsd:element name="givenValue" type = "givenType" >` by inserting the attribute *type*. The queries `ins_compType` and `ins_sequence` insert re-

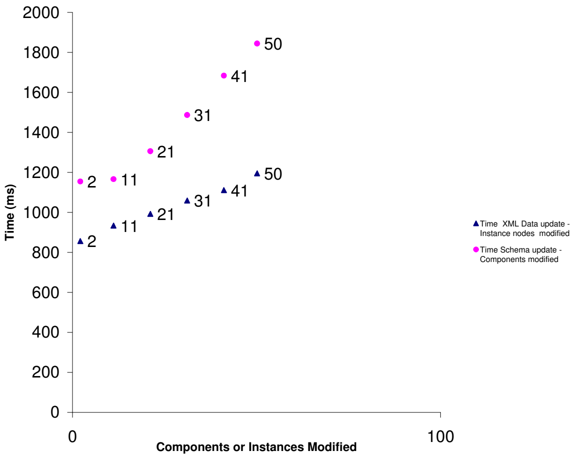


Figure 7.6: Comparing Schema and Data Time Execution for Replace Name

spectively `<xsd:complexType>` and `<xsd:sequence>` into the schema safely. They are used in the process of making a complexType component. In this experiment rather than changing the number of modified elements on the schema and the size of the schema, we change the type of safe query executed on the schema. The results represented are the averages over ten runs. The charts in Figure 7.7 investigate respectively the time to generate and run `safe_ins_name`, `safe_ins_type`, `ins_compType` and `ins_sequence`. From the graph in Figure 7.7 we can conclude the execution time of inserting an attribute node is slightly less expensive than inserting an element into the SAXE system. Also we found that all the safe insertion operations take almost the same time even though the items and the position of the insertion are not the same.

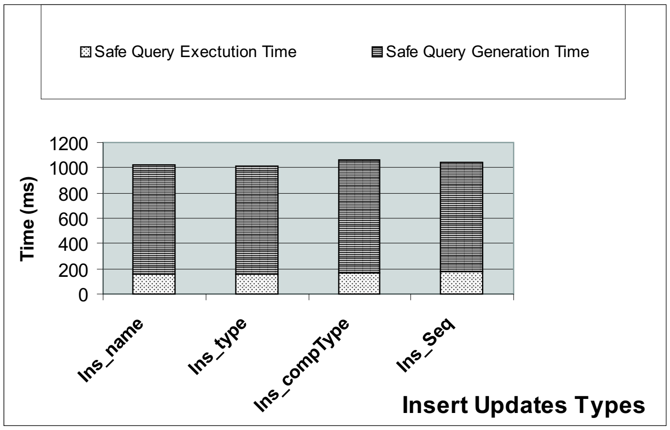


Figure 7.7: Insert Query Update Samples Affecting the Schema Alone

7.2.4 Efficiency Time Insertion for *minOccurs*, *maxOccurs* and *ref* Constraints

These experiments focus on the cost of updating some special target constraints. The constraints are *minOccurs*, *maxOccurs* and *ref*. The study below will analyze the safe queries generated when the *minOccurs*, *maxOccurs* and their values are modeled as XML attributes. For a referred node in a given component as in Figure 7.1 the component *juicer* at line 8 refers to an element *cost* at line 13. Here referring to a given component is the same as inserting *ref* attribute into a node of the schema. Such action generates two queries as result of safe updates; the two queries are called as `ins_ref_xsd` and `ins_ref_xml`. The queries for *minOccurs* insertion are `ins_min_xsd` and `ins_min_xml`. The first is executed on the schema, while the second will modify the XML data. The queries for *maxOccurs* are `ins_ref_xsd` and `ins_ref_xml` modifying the schema and the xml data respectively. The experiments below were run with a variety of sizes of XML document files. Each test was run ten times to get an accurate measure. We will analyze first the time spend executing the safe queries in the schema and second the execution for the updates on the XML data document.

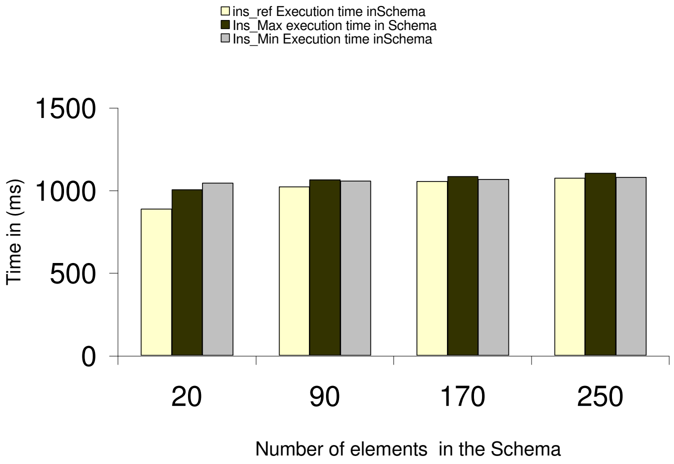


Figure 7.8: Safe Query Execution time for target constraints on The Schema

Safe Query Execution Time For The targeted Constraints on the Schema

This experiment is to study the execution of three different query schema changes. The result presented is the average of ten runs. The cost is graphed as a in function of the number of elements in the schema. The size of the schema is increased after each test. Figure 7.8 shows that it is a bit cheaper to execute `ins_ref_xsd` than `ins_Min_xsd` and `ins_Max_xsd`. This difference is mainly due to the fact that the safe query statement in the two latest cases carries more structural information in their path. In order to allow a *ref* and its value to be inserted, one necessary condition is to give the component name where the insertion will happen. For the insertion of *minOccurs* and *maxOccurs* we have two necessary conditions: the component name where the insertion will happen should be given and the name of the referred element into which the quantifiers will be inserted should be also specified in the query. We note that queries with more structural information in their path take slightly longer to execute.

The average growth rate for the execution time of those queries when the schema file is increased by one element is less than 0.26 msec. However with this approach one cannot do a bunch of updates at once in different global components, meaning the insertion of *minOccurs* and *maxOccurs* is possible only in one component at a time. This restriction on *minOccurs* and *maxOccurs* is set up in order to propagate the updates to the XML data document. Note this restriction does not hold for *ref* where a large quantity of updates can be done once.

Safe Query Execution Time for *minOccurs*, *maxOccurs* and *ref* on the Data.
Restriction was set to allow only one component update at the time when modifying *minOccurs* and *maxOccurs*, this restriction will allow the propagation of the updates to XML data. One schema update may result in a quantity of update on the XML data. This test study how efficient `ins_ref_xml`, `ins_Min_xml` and `ins_Max_xml` are when executed on a XML data. We choose to use a variable sized XML data file with the number of elements of the XML data document for the first test being equal to 60. This number is multiplied by 10 after each test. Also the number of elements to alter is increased by setting the number of altered element on a test equal to 10 power *t* where *t* is the test number. The chart in Figure 7.9 plots the time as a function of the number of modified elements on the XML data. As expected all the three plotted lines are almost the same. The purpose of `ins_ref_xml` is to insert a default child, the duty of `ins_min_xml` is to set up default children when necessary and `ins_max_xml` would be to delete the element when necessary.

The important thing to note about this test is that we have a big gain when altering a mass of element nodes. Altering 100 elements takes around 1460 msec and 6500 msec for altering 1000 elements of the data.

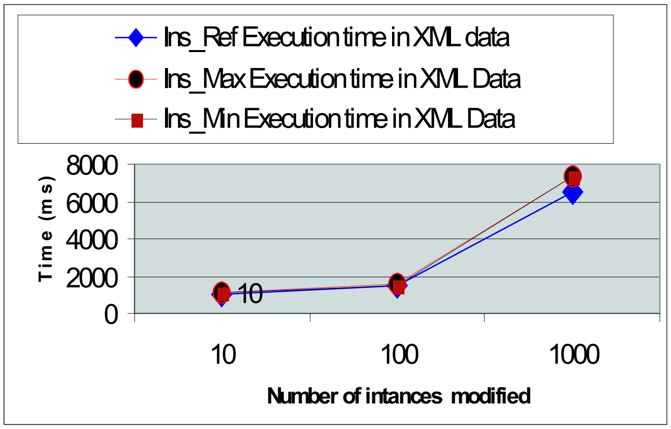


Figure 7.9: Safe Query Execution time for target constraints on The XML Data

SAXE Safe Updates Versus Third Party Validator

Now that we have studied in general the efficiency of SAXE safe updates, the next interesting step will be to compare SAXE with one of-the-shelf available tools. We used XSV a Java based XML-Schema validator [Tom02] as a comparison tool. Even though the end result of SAXE and XSV is to know if an XML document is valid or not, the problems they address are different. SAXE favors incremental constraint checking instead of re-validation from scratch. It also addresses the validation problem that would occur with manual updates. The purpose of this experiment is to compare the time it takes to safely update an XML document using SAXE or XSV. Using SAXE to achieve an update the typer should write a four line xQuery statement, one should query and execute.

As result a pair of safe queries `safe_rep_name_xsd` and `safe_rep_name_xml` will be generated. `safe_rep_name_xsd` will be executed to update the schema and `safe_rep_name_xml` will be used to update the XML data. Using XSV to achieve a component type change implies manual updates, which in some case is almost impossible for instance when the number of elements to be modified of the schema is very large. This would be done by going through the complete document and altering all the elements affected by

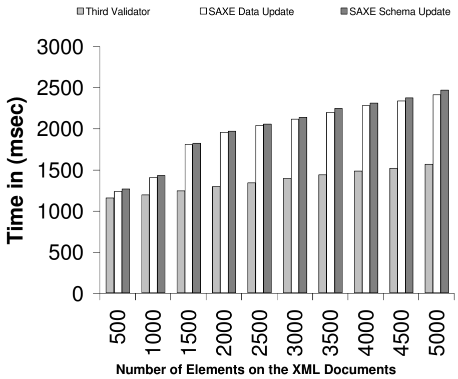


Figure 7.10: SAXE Versus Validator

the component changed and then running the XML files through the validator. The other choice is to run the safe query from the XML document directly and let the validator complain about the updates being invalid. Neither of the later two methods seems convenient.

In Figure 7.10 the plotted time for the XSV validator is the time it takes for verifying the validity of a modified XML file. Also Figure 7.10 plots the time it takes to validate an XML data or XML Schema file using SAXE. The time is set as a function of the number of elements in the XML document. The results in both graphs represent the averages of ten runs. The plot depicted in Figure 7.10 shows that the execution time using the validator is less than the time it takes to execute `safe_rep_name_xsd` and `safe_rep_name_xml`. But this does not mean that the safe update is less efficient. As we said above, the argument is that SAXE is a one step process where updates are only performed once the updates are deemed safe. So on one hand all the attempts for invalid updates will be prevented, and on the other hand this could allow a set for safe updates all at once to be committed. This is not the case of the validator where one non safe update will require to roll back and redo of all the updates. In our experiment the number of element nodes modified while updating the XML schema or XML data using SAXE is ten percent of the total number of elements on the XML document. We conclude that SAXE combined with the available editor such as XMLSpy [XML] would be a powerful and efficient update tool for XML documents. SAXE works in XML documents presented in a verbose form schema but it allows automatic and multiple safe updates once. We could use XMLSpy [XML] transform a schema in a verbose form and then take advantage of SAXE updates tools.

Chapter 8

Conclusion And Future Work

8.1 Conclusion

In this thesis, we have proposed a lightweight approach to ensure the structural consistency of XML schema and XML data after updates. More precisely, I proposed to rewrite an Update-XQuery statement into a safe Update-XQuery statement by embedding constraint check subqueries into the former 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. The key parts accomplished for this thesis are summarized below:

1. Proposed a query rewriting approach that converts an Update-XQuery into a safe query.

2. Analyzed schema specifications, and developed rules that ensure the correctness of each update originating from the XML data or the schema.

3. Produced XQuery template libraries to support safe updates when alterations originate from XML Data Updates.

4. Produced XQuery statements on the fly to support safe updates when alterations originated from the XML Schema.

5. Implemented the safe query generation when changes originate from the XML schema.

6. Performed experimental studies for the SAXE System.

8.2 Future Work

Currently, our safety checking semantics is at the atomic level, i.e., each atomic update on a single XML element is allowed if this update leads to a valid XML document. As next step, we would 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. An other field of interest will be to investigate a best incremental validation algorithm over updates using XQuery on XML documents. The efficient algorithms could be used in scenarios when update queries involve more complex manipulation of entire subtrees for instance when deleting, inserting or renaming a large quantity of elements or components safely in order to boost SAXE system performance. Also extending SAXE updates to other tractable constraint checks not considered during this study such as keys will be an valuable domain study also.

Bibliography

[AFL02] M. Arenas, W. Fan, and L. Leonid. What's hard about XML schema constraints. In *R. Cicchetti et al. (Eds.): DEXA 2002, LNCS 2453*, pages pp 269–278, 2002.

[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.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Minneapolis*, pages 313–324, June 1994.

[Cor02] MITRE Corporation. xfront.com (XML technologies), 2002.

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

[eXc98] eXcelon Corporation. Updating XML data. eXcelon 1.1 User Guide, Chapter 7, 1998.

[FG02] Norbert Fuhr and Norbert Gvert. Index compression vs. retrieval time of inverted files for xml documents. In *CIKM*, 2002.

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

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

[Inf00] Infozone Group. Infozone Working Draft for Lexus. http://www.infozone-group.org/lexusDocs/html/wd-lexus.html, 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.

[KSR02] Bintou Kane, Hong Su, and Elke A. Rundensteiner. Consistently updating XML documents using incremental constraint query checks. In *Web Information and Data Management (WIDM'02)*, pages 1–8, Nov. 2002.

[MAG$^+$97] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A Database Management System for Semistructured Data. In *SIGMOD Record 26(3)*, pages 54–66, September 1997.

[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.

[PV03] Y. Papakomstantinou and V. Vianu. Incremental validation of XML documents. In *In ICDT*, 2003.

[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 (RIDE-DM'2001)*, pages 103 – 110, 2000.

[TIHW01a] I. Tatarinov, Z. Ives, A.Y. Halevy, and D. S. Weld. Updating XML SIGMOD. In *SIGMOD*, pages 413 – 424, 2001.

[TIHW01b] I. Tatarinov, Z. Ives, A.Y. Halevy, and D. S. Weld. Updating XML SIGMOD. 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. *XML Specificaftions*. http://www.w3.org/XML, 2001.

[W3C01c] W3C. *XQuery 1.0: An XML Query Language*. http://www.w3.org/TR/xquery/, 2001.

[W3C01d] W3C XSL Working Group. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/, 2001.

[XML] XMLSpy. Spy document editor. Available at . http://www.xmlspy.com/productsdoc.html.

Appendix A

Safe Queries For XML Data Updates

Appendix A gives parameterized XQuery functions called *constraint checking templates*. Each constraint checking template is in charge of checking constraints for a certain type of update operation for the XML data updates.

```
 Function translateUse($usestatus) {
 return
 if count($usestatus) = 0
  then optional
   else $usestatus
 }
```

Function translateUse: Interpret the `use` value

```
  Function translateOccurs($contains) {
 return
  if count($contains) = 0
   then 1
  else $contains
 }
```

Function translateOccurs: Interpret minOccurs maxOccurs value

```
     Function insertEpassed($childbindN, $docxsd, $pname, $cname ){
 For $psch In document($docxsd)/xsd:element[@name =$pname],
    $xref In $psch//xsd:element[@ref=$cname]
  let $maxoccurs := $xref/@maxOccurs,
    $counted := count($childbindN),
    $val := translateOccurs($maxoccurs),
    $check := count($xref)
return
 if ( $val . >= . $counted + 1
            or $val = ”unbounded” or $check = 1 )
      Then TRUE
      else FALSE
}
```

Constraint Checking Function: Insert an element

```
     Function insertApassed($docxsd,$pname, $aname){
For $psch In document($docxsd)/xsd:element[@name = $pname],
    $schattr In $psch//xsd:attribute[@ref= $aname]
  let $use := $schattr/@use,
    $useR := translateUse($use),
 $exist := count($schattr)
return
 if ( $useR !=”prohibited” and $exist = 1)
```

```
     Then TRUE
     else FALSE
}
```

Constraint Checking Function: Insert an attribute

```
Function deletedApassed($childAtr,$docxsd,$pname,$cname){
For $psch In document($docxsd)/xsd:element[@name = $pname],
    $schattr In $psch//xsd:attribute[@ref=$cname]
let $use := $schattr/@use,
    $useR := translateUse($schattr/@use),
    $exist := count($schattr)
return
if ($useR = ”optional” and exists($childAtr)=TRUE)
     Then TRUE
     else FALSE
}
```

Constraint Checking Function: Delete an attribute node

```
    Function renameDelApassed( $childAtr, $docxsd, $pname, $cname, $nname){
   For $psch In document($docxsd)/xsd:element[@name = $pname],
    $schattr In $psch//xsd:attribute[@ref=$cname]
   let $use := $schattr/@use,
   $useR := translateUse($schattr/@use)
   return
    if $useR = ”optional” and exists($childAtr)=TRUE and $cname != $nname
     Then TRUE
```

```
        else FALSE
}
 Function renameinsApassed($atrbind, $docxsd,$pname, $nname){
 For $psch In document($docxsd)/xsd:element[@name = $pname],
    $schattr In $psch//xsd:attribute[@ref= $nname]
 let $use:= $schattr/@use,
    $useR := translateUse($use),      $exist := count($schattr)
  return
  If ( $useR !=”prohibited” and $exist = 1 )
       Then TRUE
       else FALSE
 }
```

Constraint Checking Functions: Rename an attribute

```
      Function renamedelEpassed($child, $childbindN,$docxsd,
$pname,$cname,$nname ){
 For $psch In document($docxsd)/xsd:element[@name =
$pname],
     $xref In $psch//xsd:element[@ref=$cname]
 let $minoccurs := $xref/@minOccurs,
    $val := translateOccurs($xref/@minOccurs),
    $countex := count($childbindN)
 return
  If ( $val . <= . $countex -1 and
       count($child) = 1 and ($cname != $nname))
         Then TRUE
```

```
        Else FALSE  }
Function renameinsEpassed($childbindI, $docxsd, $pname, $nname){
 For $psch In document($docxsd)/xsd:element[@name =$pname],
     $xref In $psch//xsd:element[@ref = $nname]
let $counted := count($childbindI),
     $maxoccurs := $xref/@maxOccurs,
     $val := translateOccurs($maxoccurs),
    $check := count($xref)
return
If ( ($val . >= . $counted + 1 or $val = ”unbounded”) and $check = 1 )
       Then TRUE
        else FALSE
}
```

Constraint Checking Functions: Rename a tag name

Appendix B

Safe Queries For Schema Updates

Appendix B gives sample of update queries intended for the XML Schema evolution and their generated safe updates.

Query intent to delete a global element:

Sample of Query

```
        For $psch In document(”juicers.xsd”),
          $child in $psch/xsd:element[@name =”cost”]
    update $psch {
            delete $child
     }
```

Delete a global element: *del_Glo_Ele*

Safe Query generated for del_Glo_Ele

```
1.        Function delRefComOnSchema($schemadoc, $childBindingRefName)
          {
```

```
    For $pref In document($schemadoc)//xsd:sequence,
        $child in $pref/xsd:element[@ref =$childBindingRefName]
    update $pref{
        delete $child
    }
}


FOR $psch IN document(”juicers.xsd”),
        $child IN $psch/xsd:element[/@name = ”cost”]
        let $fun := delRefComOnSchema(”juicers.xsd”,”cost”)
    update $psch{
            update $psch {
                delete $child
            }
    }
```

Safe Query Generate For Schema From `del_Glo_Ele`

```
2.          For $child in document(”juicers.xml”)//cost,
                $p in $child/parent::*
            update $p {
                    delete $child
            }
```

Safe Query Generate For Data From `del_Glo_Ele`

Query intent to delete a referred component

`Sample of Query`

```
         For $psch in document(”juicers.xsd”)/xsd:element[@name=”juicer”]
        $ch in $psch/xsd:element[@ref=”cost”]
    update $psch{
            delete $ch
}
```

Sample delete referred element: `del_Ref_Ele`

`Safe Queries Generated for del_Ref_Ele`

```
        For $psch IN document(”juicers.xsd”)/xsd:element[/@name = ”juicer”]
                        /xsd:complexType/xsd:sequence,
             $ch IN $psch/xsd:element[/@ref = ”cost”]
        update $psch {
                update $psch {
                        delete $ch
                }
        }
```

Safe Query Generated For Schema

```
        For $p in document(”juicers.xml”)//juicer,
            $child in $p/cost
                update $p {delete
$child }
```

Safe Query Generated For Data

Query intent to insert a direct element:

`Sample Query`

```
For $psch IN document(”juicers.xsd”)
   update $psch {
    insert <xsd:element/ >
   }
```

Sample insert <element/ >: `ins_Dir_Ele`

`Safe Generated Query`

```
1.      Function do_contVal_check ($result){
                return
                     if ($result=””)
                         then TRUE
                         else FALSE
        }


        Function do_Contag_check($contentName){
                 return
        if ( $contentName =”xsd:element”
                                or $contentName =”xsd:attribute”)
                        Then TRUE
                       else FALSE
        }
        For $psch IN document(”juicers.xsd”)
            update $psch {
                WHERE do_Contag_check(”xsd:element”)and do_contVal_check(””)
                update $psch {
                       insert <xsd:element/ >
                }
            }
```

Safe Query Generated For Schema From `ins_Dir_Ele`

Query intent to insert a ”name” and it’s value on < element/ >:

`Sample Query`

```
For $p in document(”juicers.xsd”)//xsd:element
    update $p{
        insert new_attribute(”name”, ”maryam juice”)
    }
```

Sample inserts ”name”: `ins_eltName`

`Safe Generated Query`

```
1.          For $p IN document(”juicers.xsd”)//xsd:element
            let $pval := $p/parent::*/@xmlns:xsd
            let $cnt := count( $p/@name),
                    $cntpval := count($pval)
            update $p {
                  WHERE $cntpval =”1” and $cnt =”0”
                     update $p {
                        INSERT new_attribute ( ”name”, ”maryam juice” )
                     }
         }
```

Safe Query Generated For Schema From ins_eltName

Query intent to insert <complexType>

Sample of Query

```
      For $psch in document(”juicers.xsd”)/xsd:element[@name=”juicer”]
            update $psch{
                  insert <xsd:complexType> < /xsd:complexType>
            }
```

insert <complexType/ > query: ins_compType

Safe Query generated

```
   1.      For $psch in document(”juicers.xsd”)/xsd:element[/@name = ”juicer”],
                 $ptype in $psch[not(@type)],
                 $pname in $ptype[@name]
           let $cre :=count( $pname//xsd:element),
                  $cra := count($pname//xsd:attribute)
           update $psch {
                 WHERE 0 . >= . $cre and 0 . >= . $cra
                 update $psch {
                    INSERT <xsd:complexType/ >
               }
          }
```

`Safe Query For Schema From ins_compType`

Query intent to insert a local element:

`Sample Query`

```
        For $psch in document(”juicers.xsd”)/xsd:element[@name=”juicer”]/
                         xsd:complexType/xsd:sequence
    update $psch{
               insert <xsd:element> < /xsd:element>
    }
```

Sample insert <element/ > as local element: `ins_Loc_Ele`

`Safe Generated Query`

```
1.        For $psch IN document(”juicers.xsd”)/xsd:element[/@name = ”juicer”]/
                               xsd:complexType/xsd:sequence
          update $psch /
               update $psch /
                    INSERT <xsd:element/ >
               }
          }
```

Safe Query Generated For Schema From `ins_Loc_Ele`

Query intent to insert ”ref”

`Sample Query`

```
For $p in document(”juicers.xsd”)/xsd:element[@name=”juicer”]/
                    xsd:complexType/xsd:sequence//xsd:element[5]
update $p {
      insert new_attribute(”ref”, ”retailer”)       }
```

Sample insert ”ref” in a local element: `ins_Ref`

`Safe Generated Query`

```
1.      For $p in document(”juicers.xsd”)/xsd:element[/@name = ”juicer”]/
                          xsd:complexType/xsd:sequence//xsd:element[position() = 5]
        let $repval := $p/@ref
        let $pval := $p/parent::*/@xmlns:xsd,
              $cntpval := count($pval)
        let $refexist := $p/parent::*/xsd:element[@ref=”retailer”],
              $cntref := count($refexist)
        For $com in document(”juicers.xsd”)/xsd:element[@name=”retailer”]
        let $cntcom := count($com)
              update $p {
                 Where 0 . >= . count($repval/text()) and 0 . >= . $cntpval and
                          $cntcom . >= . 1 and 0 . >= . $cntref
```

```
                    update $p {
                        INSERT new_attribute ( ”ref”, ”retailer” )
                    }
                }
```

Safe Query Generated For Schema From `ins_Ref`

```
2.      For $p in document(”juicers.xml”)//juicer
        let $child := $p/retailer,
            $cnt := count($child)
        update $p {
              where 0 . >= . $cnt
                update $p {
                    insert <retailer> < /retailer>
                }
        }
```

Safe Query Generate For Data From `ins_Ref`

Query intent to insert ”type”

`Sample of Query`

```
    For $p in document(”juicers.xsd”)//xsd:element
    update $p {
        insert new_attribute(”type”, ”xsd:integer”)
    }
```

insert ”minOccurs” query sample: `ins_type`

Safe Queries generated

```
1.      For $p IN document(”juicers.xsd”)//xsd:element
        let $pval := $p/parent::*/@xmlns:xsd,
            $cntdesc := count ($p/descendant::*)
        let $cnt := count( $p/@name),
            $cntpval := count($pval)
        update $p {
            WHERE $cntpval =”1” and $cnt =”1” and $cntdesc =”0”
                update $p {
                  INSERT new_attribute ( ”type”, ”xsd:integer” )
                }
        }
```

Safe Query For Schema From ins_type

Query intent to insert ”minOccurs”

Sample of Query

```
      For $p in document(”juicers.xsd”)/xsd:element[@name= ”juicer”]/
                xsd:complexType/xsd:sequence/xsd:element[@ref=”cost”]
  update $p{
            insert new_attribute(”minOccurs”, ”2”)
  }
```

insert ”minOccurs” query sample: ins_Min

Safe Queries generated

1.

```
Function check_maxmin($mvar, $val){
    return
        if ($mvar =”maxOccurs”and ($val . >= .0 or $val =”unbounded”))
            then TRUE
        else if ($mvar =”minOccurs”and $val . >= .0)
            then ”TRUE”
            elseTRUE
}
For $p in document(”juicers.xsd”)/xsd:element[/@name = ”juicer”]/xsd:complexType/
                            xsd:sequence/xsd:element[/@ref = ”cost”]
let $pval := $p/parent::*/@xmlns:xsd,
        $cntpval := count($pval)
let $mimformat := check_maxmin(”minOccurs”,”2”)
update $p {
        WHERE 0 . >= . $cntpval and $mimformat =”TRUE”
                update $p {
                    INSERT new_attribute ( ”minOccurs”, ”2” )
                }
}
```

Safe Query For Schema From ins_Min

2.

```
For $p in document(”juicers.xml”)//juicer
let $child := $p/cost,
    $cnt := count($child)
let $cnt2 := 2 - count($child)/* Number of time to run the query */
let $nodeA := $p/cost[1]
update $p {
    where $cnt2 . >= . 1
  update $p {
        insert <cost> < /cost> after $nodeA
  }
}
```

Safe Query For Data From ins_Min

Query intent to insert ”maxOccurs”

Sample of Query

```
For $p in document(”juicers.xsd”)/xsd:element[@name= ”juicer”]/
            xsd:complexType/xsd:sequence/xsd:element[@ref=”cost”]
  update $p {
      insert new_attribute(”maxOccurs”, ”3”)
  }
```

insert ”maxOccurs” query sample: ins_Max

Safe Queries generated

```
1.      Function check_maxmin($mvar, $val){
            return
              if ($mvar =”maxOccurs”and ($val . >= .0 or $val =”unbounded”))
                then TRUE
             else if ($mvar =”minOccurs”and $val . >= .0)
                then TRUE
             else
                TRUE
             }
        For $p in document(”juicers.xsd”)/xsd:element[/@name = ”juicer”]/
                              xsd:complexType/xsd:sequence/xsd:element[/@ref = ”cost”]
        let $pval := $p/parent::*/@xmlns:xsd,
             $cntpval := count($pval)
        let $mimformat := check_maxmin(”maxOccurs”,”3”)
        update $p {
              WHERE 0 . >= . $cntpval and $mimformat =TRUE
                  update $p {
                      INSERT new_attribute ( ”maxOccurs”, ”3” )
                  }
        }
```

Safe Query For Schema From ins_Max

```
2.      For $p in document(”juicers.xml”)//juicer
        let $child := $p/cost,
        $cnt := count($child)
    let $cnt2 := count($child) - 3 /* Number of time to run the query */
    let $nodeA := $p/cost[1]
    update $p {
        where $cnt2 . >= . 1
            update $p{
              delete $nodeA
            }
    }
```

Safe Query For Data From ins_Max

Query intent to insert ”use”

Sample of Query

```
    For $p in document(”juicers.xsd”)/xsd:element[@name= ”juicer”]//
                      xsd:attribute[@ref=”percentage”]
       update $p{
         insert new_attribute(”use”, ”required”)
    }
```

insert ”use” query sample: ins_Use

Safe Queries generated

```
1.        Function check_use($mvar, $val){
```

```
          return
              if ($mvar =”use” and ($val= ”optional”
                              or $val =”required” or $val = ”prohibited”))
                  then TRUE
      else TRUE
      }
      For $p in document(”juicers.xsd”)/xsd:element[/@name = ”juicer”]//
                              xsd:attribute[/@ref = ”percentage”]
      let $pval := $p/parent::*/@xmlns:xsd,
              $cntpval := count($pval)
      let $format := check_use(”use”,”required”)
          update $p {
              WHERE 0 . >= . $cntpval and $format =TRUE
      update $p {
       INSERT new_attribute ( ”use”, ”required” )
      }
}
```

Safe Query For Schema From ins_Use

```
2.          For $p in document(”juicers.xml”)//juicer
            let $child := $p/@xsi:percentage,
                    $cnt := count($child)
            update $p {
                        where 0 . >= . $cnt
                update $p {
                        insert new_attribute(”xsi:percentage”, ”defaultvalue”)
```

```
      }
    }
```

Safe Query For Data From ins_Use

Query intent to insert ”use”

Sample of Query

```
    For $p in document(”juicers.xsd”)/xsd:element[@name=”smalljuicer”]
  let $childatr := $p/@name
  update $p{
      replace $childatr with ”newName”
  }
```

replace name of component query sample: rep_Glo_EleName

Safe Queries generated

1.

```
Function replaceRefOnSchema($schemadoc, $childBindingRefName){
For $pref In document($schemadoc)//xsd:element[@ref=$childBindingRefName],
    $rRef in $pref/@ref
update $pref {
    replace $rRef with ”newName”
}
}
For $p IN document(”juicers.xsd”)/xsd:element[/@name = ”smalljuicer”]
Let $childatr := $p/@name
Let $pval := $p/parent::*/@xmlns:xsd,
    $cntpval := count($pval)
Let $child := $p/parent::*/xsd:element[@name=”newName”],
    $childexist := count($child)
Let $sref := $p/parent::*//xsd:element[@ref= ”smalljuicer”]
Let $sr := $sref/@ref
    Where $cntpval = ”1” and $childexist =”0”
      update $p {
        Where replaceRefOnSchema(”juicers.xsd”,”smalljuicer”)
          update $p {
            REPLACE $childatr WITH ”newName”
          }
      }
```

Safe Query For Schema From rep_Glo_EleName

```
2.    For $child in document(”juicers.xml”)//smalljuicer
      let $p := $child/parent::*
          update $p{
            rename $child to ”newName”
      }
```

Safe Query For Data From rep_Glo_EleName