Valid Updates for Persistent XML Objects

Henrike Schuhart and Volker Linnemann

Institut für Informationssysteme
Universität zu Lübeck
Ratzeburger Allee 160, D-23538 Lübeck, Germany
{schuhart|linnemann}@ifis.uni-luebeck.de

Abstract: XML has emerged as the industry standard for representing and exchanging data and is already predominant in several applications today. Business, analytic and structered data will be exchanged as XML between applications and web services. XQuery is a language designed and developed for querying, filtering and generating XML structured data and is currently being standardized by the World Wide Web Consortium(W3C). XQuery seems to become the query language in context of (native) XML databases. Moreover in the context of document management XQuery seems suitable for querying large collections of documents with more irregular and deeply nested data structures. Despite these promising features XQuery or more precisely its FLWOR expression lacks of any update capability.

In this paper we present important results concerning the development of **XOBE**$_{\text{DBPL}}$ (**X**ML **OB**j**E**cts **D**ata**B**ase **P**rogramming **L**anguage). **XOBE**$_{\text{DBPL}}$ is the successor of the **XOBE** project. **XOBE** integrates XML and XPath into the Java programming language. In **XOBE**$_{\text{DBPL}}$ XML objects can become persistent. Moreover, a new feature in **XOBE**$_{\text{DBPL}}$ is the integration of xFLWOR(e**x**tended FLWOR) expressions for updating and querying XML objects. XML updates and queries in **XOBE**$_{\text{DBPL}}$ are statically typechecked. Finally we perform experiments with the **XOBE**$_{\text{DBPL}}$ prototype showing that the performance of low level API-based interfaces can be improved, as well as the performance of related approaches.

1 Introduction

XML plays an important role for operational information systems and internet data. Due to this fact, there is an emerging amount of software for generating and manipulating XML documents. Therefore, programming language concepts and tools for this purpose are needed. Most of the approaches that are currently in use are not sufficient, since they cannot guarantee that only valid XML documents are processed and especially result from update operations. A valid XML document in this context means a document which is correct according to a given language description or schema. In this paper a schema denotes either an XML Schema or an XML Document Type Definition(DTD). If a document is valid, it is an element of the language defined by the schema. Tools and languages which do not guarantee the validity of generated XML documents at compile time have to execute extensive runtime tests.

Our ultimate goal is to develop a database programming language for XML applications by extending the general-purpose language JAVA. In the **X**ML **OB**j**E**cts(**XOBE**) project[Ke03, KL03] XML is integrated into JAVA by defining XML objects representing XML fragments and by treating them as first-class data values. Accessing XML objects in **XOBE** is realized by integrating XPath.

In this paper we wish to introduce the successor project **XOBE**$_{\text{DBPL}}$ offering

1. update operations for XML objects
2. static type analysis for these update operations guaranteeing valid updated XML objects.

Update operations in **XOBE**$_{\text{DBPL}}$ are syntactically realized by e**x**tended FLWOR[W304] (xFLWOR) expressions. xFLWOR expressions support inplace updates in **XOBE**$_{\text{DBPL}}$.

XOBE$_{\text{DBPL}}$ provides a type safe general basis for any application involving XML and persistent XML, including web services and servlets based on XML databases.

Contributions of this paper. This paper introduces **XOBE**$_{\text{DBPL}}$ by an example taken from the XMark[SWK^{+}02] project dealing with an auction scenario. Moreover this paper formally presents xFLWOR(e**x**tended FLWOR) expressions and type inference rules designed for type safe querying and updating (persistent) XML structured data. Finally we present experimental results showing that **XOBE**$_{\text{DBPL}}$ attains the performance of traditional low level API-based solutions as well as related approaches.

A running prototype of the **XOBE**$_{\text{DBPL}}$ system is available including a connection to a native XML database.

This paper is organized as follows. In Section 2 we give an overview of related work regarding XML in programming languages, persistent XML as well as aspects of updating XML. In Section 3 we give a brief review on the integration of XML objects in **XOBE**. The central section of our paper is Section 4, where we present new features in **XOBE**$_{\text{DBPL}}$, more precisely persistent XML objects, xFLOWR expressions and type inference rules used in **XOBE**$_{\text{DBPL}}$ which are suitable for type safe updates on (persistent) XML structured data. Section 5 gives a survey of the architecture of our **XOBE**$_{\text{DBPL}}$ prototype implementation and provides some details of the **XOBE**$_{\text{DBPL}}$ preprocessor. In Section 6 we finally present some experimental results concerning the performance of valid updates in **XOBE**$_{\text{DBPL}}$, low level API-based solutions and related approaches. Concluding remarks and an outlook for future work finish the paper.

2 Related Work

XML in Programming Languages. The most elementary way to deal with XML fragments is to use ordinary strings without any structure, i.e. Java Servlets [Wi99]. Java Server Pages [PLC99] provide an improvement by allowing to switch between XML parts and Java by using special markings. All these approaches share the disadvantage that not

even well-formedness is checked at compile time.
Low-level-binding approaches like the Document Object Model DOM [W398] or JAVA DOM (JDOM) [JD] provide classes for nodes in an XML document thus allowing to access and manipulate arbitrary XML fragments by object oriented programming. Low-level bindings ensure well-formedness of dynamically generated documents at compile time, but defer validation until runtime. The ECMAScript language extension for XML[Ec04] integrates the construction and manipulation based on a tree-like navigation of XML objects into ECMAScript[Ec99], validation against an XML language description is not supported.
High-level bindings [Bo02] like Sun's JAXB, Microsoft's .Net Framework, Exolab's Castor, Delphi's Data Binding Wizard and Oracle's XML Class Generator [Su01, Mi01, Ex01, Bo01, Or01] assume that all processed documents follow a given schema. This description is used to map the document structure onto language types or classes reproducing directly the semantics intended by the schema. Apache's XML Beans[Ap03] offer a DOM-like tree navigation API to underlying XML documents or additionally generate a set of Java classes and interfaces corresponding to XML schemas, if XML schemas are compiled. A third API supports XQuery. Validity at compile time is only supported up to a limited extent depending on the selected language mapping.
Recently, the aspect of guaranteeing the validity of XML structures at compile time gained some interest. The XDuce language [HP03] is a special functional language developed as an XML processing language. XML elements are created by specific constructors. The content can be accessed through pattern matching. XDuce supports type inference and performs a subtyping analysis to ensure validity of XML expressions at compile time. The subtyping algorithm is implemented on the basis of regular tree automata. The Xtatic project [GP03] is the successor of XDuce. The main purpose of Xtatic is to couple the concepts of XDuce with the object oriented programming language C#. Xtatic has similar goals as **XOBE**$_{\text{DBPL}}$, however, Xtatic is still in an early stage and, in contrast to **XOBE**$_{\text{DBPL}}$, Xtatic does not support XPATH. BigWig [BMS02] is a special programming language for developing interactive web services. JWig [CMS03] is the successor of BigWig integrating the XML-specific parts of BigWig into JAVA. JWig is quite close to **XOBE**$_{\text{DBPL}}$. The main difference in JWig is that there is only one XML type. Typed XML document templates with gaps are introduced. In order to generate XML documents dynamically, gaps can be substituted at runtime by other templates or strings. For these templates JWig validates all possibly dynamically computed documents according to a given abstract DTD by data flow analysis. This data flow analysis is rather time consuming. In the Xact project [KMS04] JWig's validation algorithm is extended to the problem of static analysis of XML transformations in JAVA. As in **XOBE**$_{\text{DBPL}}$, XPath is used for expressing XML transformations. XJ [HRS^{+}03], a new project by IBM research, integrates XML into Java concentrating on traversing XML structures by using XPath. The distinguishing characteristic of XJ is its support for inplace updates. Xen [MSB03] is an integration of XML into popular object-oriented programming languages such as C# or JAVA currently under development at Microsoft Company. Xen uses XML constructors similar as **XOBE**$_{\text{DBPL}}$'s XML object constructors. XL [FGK02] is a special programming language for the implementation of web services. It provides high-level and declarative constructs adopting XQuery [W304]. Additionally imperative language statements are

introduced making XL a combination of an imperative and a declarative programming language.
Updates in XML. The Document Object Model DOM [W398] offers low-level updates, which ensure well-formedness but no validation. There are several approaches dealing with updates for XML in the context of XQuery. Extending XQuery to support updates is proposed in [TIHW01, Le01, W302]. The syntactic proposal made in [TIHW01] is adopted in this paper. Basically [TIHW01] deals with the mapping of these update expressions to SQL. Any validation of updates is omitted. In [BA03] updates are checked before execution but this happens at runtime. The language introduced in [SHS04] is based on proposals in [W302] and provides XML update support tightly coupled with XQuery itself. Validity checking is done during the update execution phase. In [SKC$^+$03] updates as defined in [TIHW01] are rewritten at compile time but validity checking is done at runtime. In [KLL03, PV03, YXGZ03] validity of updates is checked upon the resulting XML documents or respectively the resulting XML data. A declarative XML update language is proposed in [LLW03]. XML updates in XML syntax are proposed in the XUpdate project[XM04]. Contrary to all approaches mentioned here **XOBE**$_{\text{DBPL}}$ integrates high-level XML updates into JAVA and checks their validity at compile time.
Persistent XML. For storing XML structures persistently several approaches exist. One approach is to map XML structures to relational tuples or blobs and store them in an object relational data base. For example, Oracle [Or03, MB03] and DB2 [IB, CX00] belong to this group of systems. Although Oracle extends SQL to provide support for XPATH [W399], application programs work with the provided structures by using conventional tools like Java Database Connectivity JDBC. JDBC does not provide any facilities to guarantee that the application program works only with valid XML structures. This means that either there is no validation or validation has to be performed on demand at runtime for the whole document. In contrast to **XOBE**$_{\text{DBPL}}$ this means that validity cannot be guaranteed at each step of the XML generation and manipulation. The same holds in the case when the application program does not work with tables but uses exported XML structures. In this case, conventional tools like DOM [W398] or SAX [xml] are used. DOM and SAX do not allow to check the validity of XML structures at compile time, i.e. runtime validation is necessary.
Another group of approaches is known as the group of native XML data base systems. Prominent examples are Tamino [Sc01],Xindice[Ap03], Infonyte DB [In03] and Natix [FHK$^+$02]. These systems do not use relations or objects, instead, they store XML structures directly. Most systems use application programming interfaces that base on DOM or SAX, i.e. there is no validity checking at compile time. Moreover, DOM and SAX are rather low level interfaces requiring a lot of programming. An exception is Xindice[Ap03] which supports XUpdate[XM04]. To the best of our knowledge, there is no system with an application programming interface that allows checking the validity of XML structures statically at compile time of the application program.

3 Review on XML Objects

In this section we briefly summarize the syntax and semantics of XML Objects in **XOBE** in an informal manner, because **XOBE**$_{\text{DBPL}}$ is based on **XOBE**. A more detailed introduction can be found in [Ke03] and [KL03]. **XOBE** extends the object oriented programming language JAVA by language constructs to process XML fragments in particular XML documents. XPATH[W304] is used for traversing XML objects.

In **XOBE** we represent XML fragments, i.e. trees corresponding to a given schema by XML objects. Therefore, XML objects are first-class data values that may be used like any other data value in JAVA. The declared schema is used to type different XML objects.

Throughout this paper we use the **auction** schema of the XMark[SWK$^+$02] project as the basis for our examples. The XMark database models an Internet auction site. The main elements are: item, open auction, closed auction, person and category. A schema description of elements and types used in this paper can be found in the appendix.

Listing 1 introduces the most important features of **XOBE**. In line 7 an XML object of type *person* is created with a so-called XML constructor. The type declaration `person` of variable `p` in line 2 is an abbreviated version. In general the type declaration of an XML variable starts with the keyword `xml` followed by square brackets. Within the brackets either a single schema type identifier followed by an optional star or a choice of them is used to type the XML variable. XML objects can be accessed by an XPath expression. In line 3 a previously declared XML object **auctionSite** of type *site* is searched whether the new person already exists. Only if the result is negative, the new XML object person is created and returned.

```
1    person createPerson(String p_name, String p_email, String p_id){
2      person p;
3      xml<person*> per = $auctionSite//person[@id={p_id}]$;
4      if(per.getLength()>0) return null; //person already exists

6      //new person is created
7      p = <person @id={p_id}>
8                <name>{p_name}</name>
9                <email>{p_email}</email>
10         </person>;

12         return p;
13   }
```

Listing 1: **XOBE**$_{\text{DBPL}}$ method `createPerson`

4 Updates in XOBE$_{\text{DBPL}}$

In this section we concentrate on **XOBE**$_{\text{DBPL}}$ and its new features: xFLWOR(**ex**tended FLWOR) expressions for type safe querying and updating persistent XML objects. As introduced in Section 3 type safe extraction of XML values can be done by using XPath. In order to execute more complex queries, especially in the context of persistent XML objects stored in a database, **XOBE**$_{\text{DBPL}}$ integrates e**x**tended FLWOR(xFLOWR) expressions. In contrast to FLWOR expressions xFLWOR expressions overcome read-only limitations and support update operations on XML data.

Persistent XML Objects. In this passage we want to explain briefly how we make XML objects persistent. As explained later in section 5 about implementation details, XML objects are internally represented as DOM [W398] objects at runtime of any **XOBE**$_{\text{DBPL}}$ program. Since there already exists a lot of solutions for making DOM objects persistent, we use a native XML database to realize persistency. Further aspects like transactions and a more sophisticated persistence mechanism are beyond the scope of this paper.

4.1 Update expressions

Now, that we have explained how XML objects can become persistent and how these persistent XML objects are accessed in a **XOBE**$_{\text{DBPL}}$ program we introduce the xFLWOR expressions. xFLWOR expressions in **XOBE**$_{\text{DBPL}}$ are supposed to query and update persistent XML objects. xFLWOR expressions in **XOBE**$_{\text{DBPL}}$ adopt syntactical proposals of [TIHW01] to extend XQuery's [W304] well-known FLWOR expression construct. In this approach any RETURN clause can optionally be replaced by a so-called UPDATE clause.

Definition 4.1 (xFLWOR expression) An xFLWOR expression is defined by the following grammar:

```
<xFLWOR> →   (<LetClause> | <ForClause>)+
             <WhereClause>?
             <OrderByClause>?
             (<ReturnClause> | <UpdateClause>)
```

□

All nonterminals except `<UpdateClause>` are defined exactly as in XQuery's FLWOR expression grammar[W304]. An `<UpdateClause>` is based on the definition:

Definition 4.2 (UPDATE clause)

An UPDATE clause is defined by the following grammar:

```
<UpdateClause>       →   UPDATE <Variable> <subOp> (,<subOp>)*
<subOp>              →   <InsertOperation>
                         | <DeleteOperation>
                         | <RenameOperation>
                         | <ReplaceOperation>
<InsertOperation>    →   INSERT <Content>
                         ((INTO|BEFORE|AFTER) <LocationPath>)?
<DeleteOperation>    →   DELETE <LocationPath>
<RenameOperation>    →   RENAME <LocationPath> TO <Name>
<ReplaceOperation>   →   REPLACE <LocationPath> WITH <Content>
```

□

The grammar states that an UPDATE clause is initiated by the keyword **UPDATE** followed by a variable name upon which the update is performed(in the following also called **update target**). The update operation is described by a sequence of fundamental suboperations, at least one suboperation must be given. The variable being the target of the update operation must be formerly defined and has to be either an XML object or a list of XML objects.

Each suboperation is performed successively upon the XML object(s).

The `<Content>` in **XOBE**$_{\mathrm{DBPL}}$ is a previously defined XML variable and `<LocationPath>`s are XPath expressions selecting **descendant** objects of the update target. It is important to notice that the XPath expressions are limited to those selecting descendants or childs, because any other context cannot be guarenteed to exist for XML objects in a **XOBE**$_{\mathrm{DBPL}}$ program at runtime.

```
1  synchronized int bid(String p_id,int incr, String a_id){

3    //calculate new current
4          xml<current*> cur = $auctionSite//open_auction[@id={a_id}]/current$;
5          current n_current = <current>{cur.itemAsInt(0) + incr}</current>;
6          //create new bidder
7          bidder bid = <bidder>
8                           <date>{getDate()}</date>
9                           <time>{getTime()}</time>
10                          <personref person={p_id}/>
11                          <increase>{incr}</increase>
12                        </bidder>;
13         //update auction
14           $LET i := auctionSite//open_auction[@id={a_id}]
15            UPDATE i INSERT {bid},
16            REPLACE i/current WITH {n_current}$;
17   return n_current;
18 }
```

Listing 2: **XOBE**$_{\mathrm{DBPL}}$ method `bid`

The **XOBE**$_{\mathrm{DBPL}}$ method in listing 2 registers a new bid for an auction. The bidder and the auction are selected by their ids. Additionally the increase is passed as parameter as well. In line 4 and 5 the new current bid is calculated, in a second step in line 7-12 the new bidder is created as an XML object. Finally in line 14-17 the update operation upon the auction site is executed. The update operation itself consists of an insert and a replace. The first one is needed to insert the new bidder and the second to replace the old with the new current. Please notice, that this method is declared as `synchronized`. `synchronized` is used in connection with Java threads and guarantees that methods are not executed in parallel.

4.2 Validity of Updates

To explain how **XOBE**$_{\mathrm{DBPL}}$ statically checks the validity of updates we need to explain its type system first.

Type System. **XOBE**$_{\mathrm{DBPL}}$'s type system is an extension of the **XOBE** type system which is described in detail in [Ke03] on top of the standard JAVA type system. Checking type correctness of a **XOBE**$_{\mathrm{DBPL}}$ program consists of three parts.

Formalization translates the declared schema description into a more formal representation.

Type inference is used to determine XML types and differs for XML constructors, XPath expressions and xFLWOR expressions in a **XOBE**$_{\text{DBPL}}$ program.

Subtype algorithm checks, if the inferred XML types are valid. The description of the subtype algorithm is not part of this paper. Details can be found in [Ke03] and [KL03].

In **XOBE**$_{\text{DBPL}}$ we formalize and represent types as regular hedge expressions representing regular hedge languages [BKMW01]. Consequently a schema is formalized and represented internally by a regular hedge grammar.

Limitations of formalization. Any resulting regular hedge grammar in **XOBE**$_{\text{DBPL}}$ is limited to cover structural constraints on XML types. In particular this means that the value-based constraints implied by ID/IDREFs in DTDs as well as ID/IDREFs and key/keyrefs in XML Schemas are not preserved by the formalization process. Since **XOBE**$_{\text{DBPL}}$ intents to check static validity, this is not a limitation. In general such value-based constraints cannot be checked at compile time at all.

Regular hedge expressions and regular hedge grammars are used in **XOBE**$_{\text{DBPL}}$ as in **XOBE** [KL03, Ke03]. For readability, the corresponding definitions are repeated here.

Definition 4.3 (regular hedge grammar) A regular hedge grammar is defined by $G = (T, N, s, P)$ with a set $T = B \cup E$ of terminal symbols, consisting of simple type names B and a set E of element names (Tags), a set N of nonterminal symbols (names of groups and complex types), a start expression s and a set P of rules or productions of the form $n \rightarrow r$ with $n \in N$ and r is a regular hedge expression over $T \cup N$.[1] □

We define the regular hedge expression, referred in short as regular expression, similar to the notation used in [W304].

Definition 4.4 (regular hedge expression) Given a set of terminal symbols $T = B \cup E$ and a set N of nonterminal symbols, the set Reg of regular hedge expressions is defined recursively as follows:

$\emptyset$	$\in Reg$	the empty set,	$e[r]$	$\in Reg$	the elements,
ϵ	$\in Reg$	the empty hedge,	$r\|s$	$\in Reg$	the regular union operation,
b	$\in Reg$	the simple types,	r, s	$\in Reg$	the concatenation operation, and
n	$\in Reg$	the complex types,	$r*$	$\in Reg$	Kleene star operation.

for all $b \in B, n \in N, e \in E, r, s \in Reg$.

□

Attributes are treated as element types with simple content having a name prepended by '@'. Disorder constraints of attributes can be simulated by generating a choice type of all possible sequences. However, our prototype is implemented more efficiently, which is beyond the scope of this paper.

The formalisation step applied to the **auction** schema in appendix A yields the following regular hedge grammar:

[1]We restrict r to be recursive in tail position only. This ensures regularity.

Example 4.1
As explained above only structural constraints of schemas are formalized. Element names and simple types are **boldfaced**, nonterminal symbols are *italic*. An **@** marks an attribute. The start expression s is *auction_xsd*.

$$
\begin{aligned}
\textit{auction\_xsd} &\rightarrow \texttt{\textbf{document}}[\textit{site}]\\
\textit{site} &\rightarrow \texttt{\textbf{site}}[\textit{regions},\textit{categories},\textit{catgraph},\textit{people},\textit{open\_auctions},\textit{closed\_auctions}]\\
\textit{people} &\rightarrow \texttt{\textbf{people}}[(\textit{person})*]\\
\textit{open\_auctions} &\rightarrow \texttt{\textbf{open\_auctions}}[(\textit{open\_auction})*]\\
\textit{open\_auction} &\rightarrow \texttt{\textbf{open\_auction}}[\texttt{\textbf{@id}}[\text{string}],\textit{initial},\textit{bidder},\textit{current},\textit{itemref},\\
&\qquad\qquad\qquad \textit{seller},\textit{quantity}]\\
\textit{initial} &\rightarrow \texttt{\textbf{initial}}[\text{string}]\\
\textit{bidder} &\rightarrow \texttt{\textbf{bidder}}[\textit{date},\textit{time},\textit{personref},\textit{increase}]\\
\textit{date} &\rightarrow \texttt{\textbf{date}}[\text{string}]\\
\textit{time} &\rightarrow \texttt{\textbf{time}}[\text{string}]\\
\textit{personref} &\rightarrow \texttt{\textbf{personref}}[\texttt{\textbf{@person}}[\text{string}]]\\
\textit{increase} &\rightarrow \texttt{\textbf{increase}}[\text{string}]\\
\ldots &\rightarrow \ldots
\end{aligned}
$$

The regular expression type of the start expression s is implicitly defined by the schema. The *auction_xsd* type represents the condition that each element which is defined as a direct child of the schema root element, can be used as a valid root element within a corresponding schema instance(document). The **auction** schema defines a single document root element *site*, therefore *auction_xsd* is derived to **document**[*site*]. If more root elements are defined, the content type becomes a choice, i.e. **document**[*root_type_1*|...|*root_type_n*].

In a next step XML types in a **XOBE**$_{\text{DBPL}}$ program are inferred.

In **XOBE**$_{\text{DBPL}}$ all variables have to be declared, therefore type inference of variables is simple. In listing 2 variable `bid` is declared of type *bidder*, variables `p_id` and `incr` of type *string* and the result types of methods `getDate()` as well as `getTime()` are *string*. Based on variable and result types, types of whole XML constructors on the right hand side of an assignment can be inferred quite intuitively. In the example above:
bidder[**date**[string],**time**[string],**personref**[**@person**[string]],**increase**[string]].

After inferring the types of the left and right hand sides, the **XOBE**$_{\text{DBPL}}$ type system checks if the type of the right hand side is a subtype of the type of the left hand side. In this example **XOBE**$_{\text{DBPL}}$ has to check if
bidder[**date**[string],**time**[string],**personref**[**@person**[string]],**increase**[string]]
is a subtype of *bidder* according to the **auction** schema in listing A in the appendix.

4.2.1 Type inference for Updates

As introduced by an example in the last section type inference for XML object constructors in **XOBE**$_{\text{DBPL}}$ can be understood quite intuitively. Type inference for XPath expressions in **XOBE**$_{\text{DBPL}}$ starts with the context variable and proceeds by inferring recursively the types of selected nodes by each step. The result type is inferred after the last XPath expression step is handled.

Example 4.2
The type of the XPath expression in listing 2 line 4:

auctionSite//open_auction[@id={a_id}]

is inferred as follows. The context variable is declared of type *site*. This XPath expression consists of one step selecting all descendant elements with tagname **`open_auction`**. The result type is inferred as `[`*`open_auction`*`]`$*$, which is of course not the best type. Since **XOBE**$_{\text{DBPL}}$'s type system lacks of value based constraints, the predicate cannot be taken into account.

Details of XPath type inference rules can be found in [Ke03]. Type inference rules for xFLWOR expressions containing a RETURN clause are a mixture of XPath type inference rules to infer FOR and LET clause variables and XML constructor type inference rules used to infer the resulting type of the Return clause. In this section we will concentrate on type inference of xFLWOR expressions containing an UPDATE clause. An xFLWOR expression with an UPDATE clause consists of one to many LET and FOR clauses defining *local variables*, corresponding types can be inferred with known rules. Consequently a new set of rules is merely needed for the UPDATE clause itself. Finally the type of the updated variable is defined to be the result type of the whole expression.

In the following we will concentrate on inferring types of UPDATE clauses **`UPDATE`** `i`..., with the type of variable `i` already given.

Let's look at the following update clause example
`UPDATE` `i` **`DELETE`** `i/person`, with variable `i` declared of type *people*. Here child elements named **`person`** are targets of a delete operation. **XOBE**$_{\text{DBPL}}$ infers the updated type as **`people`**[ϵ |*person*$*$], either all person elements are deleted(ϵ) or none is deleted(*person*$*$). Obviously all person elements will be deleted, but in general the influence of predicates must be taken into account. The type above can be simplified to **`people`**[*person*$*$], which is valid since it is the *people* type as defined in the schema.

Influence of predicates. Each *LocationPath* can be rewritten so that a single predicate is part of the last or respectively of the first step. Constraints of this predicate can refer to any node of any step of the whole path. Among these constraints are value-based constraints, which in general cannot be checked by any static type inference system. Therefore our type inference rules for update operations implicitly assume predicate constraints for each step. As type inference works only structure based, the selection of predicates applies to structure as well. Consequently predicate constraints can either evaluate to `true` or to `false` for nodes with the same type, which is reflected by the choice types in our type inference rules.

Each basic update operation (delete, insert, rename and replace) has got its own set of auxiliary type inference rules. Due to their analogous construction we concentrate on introducing the sets for delete operations. As mentioned above XPath expressions selecting target nodes of an update operation are only allowed to contain `descendant` as well as `child` axis. This can easily be checked before applying any of the following type inference rules.

Definition 4.5 (function *delete*)

The function `delete` : *Reg* × *Path* → *Reg* provides the type $r \in Reg$ in case the nodes selected by path $p \in Path$ are deleted and is recursively defined as:

delete(∅,p)	=	∅
delete(ϵ,p)	=	∅
delete(b,p)	=	∅
delete(n,p)	=	delete(r,p) with n → r ∈ P
delete(r|s,p)	=	delete(r,p) | delete(s,p)
delete((r,s),p)	=	delete(r,p) , delete(s,p)
delete(r∗,p)	=	(delete(r,p))∗
delete(e[r],//test)	=	e[r'] ,r' = deleteDescendants(r,test)
delete(e[r],/test)	=	e[r'] ,r' = deleteChildren(r,test)
delete(e[r],//test/*p*)	=	e[r'] ,r' = applyDescendants(r,test,*p*)
delete(e[r],/test/*p*)	=	e[r'] ,r' = applyChildren(r,test,*p*)

with `b` ∈ B, n ∈ N, e ∈ E, r,s ∈ *Reg* and `p` ∈ *Path*. `//test` indicates a descendant step with a `test` ∈ E ∪ {∗} , respectively `/test` indicates a child step. In this context ∗ stands for a wildcard. */p* represents any non empty path.

The auxiliary functions *deleteDescendants, deleteChildren, applyDescendants* and *applyChildren* are defined in the following. □

Definition 4.6 (function *applyDescendants*)

The function `applyDescendants` : *Reg* × *E* × *Path* → *Reg* navigates through the input type $r \in Reg$ and calls the function *delete* with the unchanged parameter $path \in Path$ for descendants conforming to the $nodetest \in E \cup \{*\}$. The function is recursively defined as:

applyDescendants(∅,test,*p*)	=	∅
applyDescendants(ϵ,test,*p*)	=	ϵ
applyDescendants(b,test,*p*)	=	b
applyDescendants(n,test,*p*)	=	applyDescendants(r,test,*p*) with n → r ∈ P
applyDescendants(e[r],test,*p*)	=	$\begin{cases} e[r'] \mid \text{delete}(e[r'],p) & \text{,if } e = \text{test} \vee \text{test} =' *' \\ e[r'] & \text{,else} \end{cases}$
applyDescendants((r,s),test,*p*)	=	applyDescendants(r,test,*p*) , applyDescendants(s,test,*p*)
applyDescendants(r|s,test,*p*)	=	applyDescendants(r,test,*p*) | applyDescendants(s,test,*p*)
applyDescendants(r∗,test,*p*)	=	(applyDescendants(r,test,*p*))∗

with r' = applyDescendants(r,test,*p*) and `b` ∈ B, n ∈ N, e ∈ E, r,s ∈ *Reg*, p ∈ *Path* and `test` ∈ E ∪ {∗} . □

Definition 4.7 (function *applyChildren*)

The function `applyChildren` : *Reg* × *E* × *Path* → *Reg* navigates through the input type $r \in Reg$ and calls the function *delete* with the unchanged parameter $path \in Path$ for children conforming to the $test \in E \cup \{*\}$. It is defined analogously to *applyDescendants* except:

$$\text{applyChildren(e[r],test,}p\text{)} = \begin{cases} e[r] \mid \text{delete}(e[r],p) & \text{,if } e = \text{test} \vee \text{test} =' *' \\ e[r] & \text{,else} \end{cases}$$

with $\texttt{b} \in$ B, n $\in$ N, e $\in$ E, r,s $\in$ *Reg*,p $\in$ *Path* and $\texttt{test} \in$ E $\cup \{*\}$. $\square$

Definition 4.8 (function *deleteDescendants*)

The function `deleteDescendants : Reg × E → Reg` provides the type $r \in$ *Reg* in case the descendants conforming to the *test* $\in E \cup \{*\}$ are deleted. The function is recursively defined as:

$$\begin{array}{lcl}
\text{deleteDescendants}(\emptyset,\text{test}) & = & \emptyset \\
\text{deleteDescendants}(\epsilon,\text{test}) & = & \epsilon \\
\text{deleteDescendants(b,test)} & = & \text{b} \\
\text{deleteDescendants(n,test)} & = & \text{deleteDescendants(r,test) with n} \rightarrow \text{r} \in \text{P} \\
\text{deleteDescendants(e[r],test)} & = & \left\{ \begin{array}{ll} e[r'] \mid \epsilon & \text{,if } e = \text{test} \vee \text{test} =' *' \\ e[r'] & \text{,else} \end{array} \right. \\
\text{deleteDescendants((r,s),test)} & = & \text{deleteDescendants(r,test)} \\
 & & \text{, deleteDescendants(s,test)} \\
\text{deleteDescendants(r|s,test)} & = & \text{deleteDescendants(r,test)} \\
 & & \mid \text{deleteDescendants(s,test)} \\
\text{deleteDescendants(r}*\text{,test)} & = & \text{(deleteDescendants(r,test))}*
\end{array}$$

with r' = deleteDescendants(r,test) and $\texttt{e} \in$ E, r $\in$ *Reg* and $\texttt{test} \in$ E $\cup \{*\}$. $\square$

Definition 4.9 (function *deleteChildren*)

The function `deleteChildren : Reg × E → Reg` provides the type $r \in$ *Reg* in case the children conforming to the *test* $\in E \cup \{*\}$ are deleted. It is defined analogously to *deleteDescendants* except:

$$\text{deleteChildren(e[r],test)} \quad = \quad \left\{ \begin{array}{ll} e[r] \mid \epsilon & \text{,if } e = \text{test} \vee \text{test} =' *' \\ e[r] & \text{,else} \end{array} \right.$$

with $\texttt{e} \in$ E, r $\in$ *Reg* and $\texttt{test} \in$ E $\cup \{*\}$. $\square$

Now the following concluding type inference rule can be formulated. In case of other update suboperations, i.e. `replace, rename` and `insert, insert after` or `insert before` rules are defined analogously.

Definition 4.10

With the help of the type inference functions defined in 4.5 - 4.9, the following concluding rule can be formulated:

$$\frac{i \quad : r \in \textit{Reg}}{\text{DELETE } i/\text{path} \quad : \text{delete(r,path)} \in \textit{Reg}} \quad \text{(DELETE)} \qquad \square$$

In case a delete operation on variable `i` is detected at compile time, the `DELETE` rule is applied to infer the resulting type of `i`. If the resulting type is still valid according to the schema, the delete operation is accepted. The following example demonstrates the influence of predicates in case of update operations.

Example 4.3
Let's infer the type of the following update operation:
`UPDATE i DELETE i//person[@id='p01']` and assume that variable `i`'s type is former inferred as *people*, due to readability let *person* be of type **`person`**`[`*`name`*`, `*`emailaddress`*`]`.

delete(*people*,'//person')	=	**people**[r]
r	=	deleteDescendants(*person*∗,'person')
	=	(deleteDescendants(*person*,'person'))∗
	=	(**person**[s]| ϵ)∗
s	=	deleteDescendants((*name*,*emailaddress*),'person')
	=	deleteDescendants(*name*,'person'), deleteDescendants(*emailaddress*,'person')
	=	**name**[u],**emailaddress**[t]
u	=	deleteDescendants(string,'person')
	=	string
t	=	deleteDescendants(string,'person')
	=	string

→ **people**[(**person**[**name**[string],**emailaddress**[string]]| ϵ)∗] →**people**[*person*∗]

The inferred type is used as input for the type checking analysis as described before. In this case the delete operation yields a valid type, because deleting person element(s) of the optional node sequence inside a people element is allowed. The inferred type can further be simplified to the original type *people*. The update operation of the example deletes one single person element by its id. Although static type inference systems in general cannot check value-based constraints, such constraints can implicitely be taken into account(the choice type in line 4).

5 Implementation

The architecture of the **XOBE**$_{\text{DBPL}}$ preprocessor and a transformed **XOBE**$_{\text{DBPL}}$ program at runtime is shown in figure 1. At runtime **XOBE**$_{\text{DBPL}}$'s xFLWOR engine accesses the XML database to perform updates and queries.

In our implementation we use the JAVA compiler compiler JavaCC [We02] to generate the **XOBE**$_{\text{DBPL}}$ parser. Additionally we use the XML parser Xerces [Ap01] to recognize the declared schemas. The internal representation of processed **XOBE**$_{\text{DBPL}}$ programs is done via the JAVA tree builder JTB [TWP00].

XML objects are internally represented and stored using the Document Object Model (DOM) [W398]. Please note that even though DOM is untyped, the transformed XML objects of a **XOBE**$_{\text{DBPL}}$ program are valid, because type checking is done by the preprocessor before.

The **XOBE**$_{\text{DBPL}}$ prototype includes an implemented xFLWOR(see section 4.1) analyzer and engine. The xFLWOR analyzer is part of the **XOBE**$_{\text{DBPL}}$ preprocessor and checks extended FLWOR expressions, an update capable extension of XQuery's FLWOR expression [W304], at compile time. The xFLWOR engine is part of any transformed **XOBE**$_{\text{DBPL}}$ program's runtime environment and executes valid xFLWOR expressions.

At the moment **XOBE**$_{\text{DBPL}}$ uses as database backend the native XML database Infonyte [In03], but this could easily be replaced by any other native or object oriented database system.

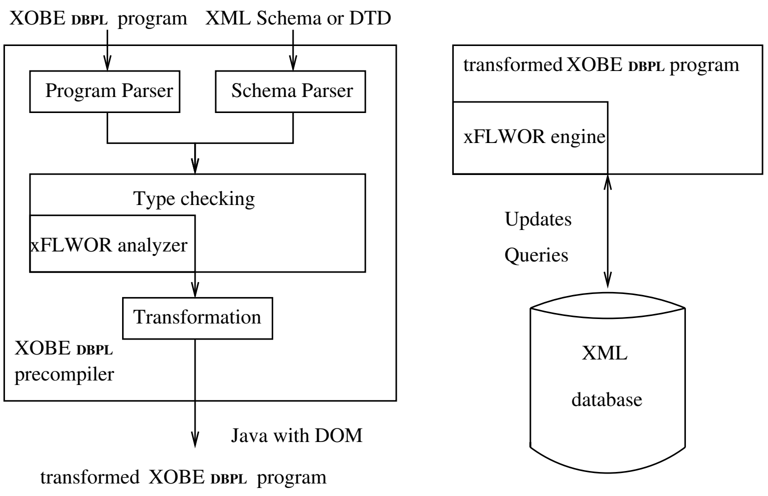

Figure 1: On the left the architecture of the **XOBE**$_{\text{DBPL}}$ preprocessor. On the right a transformed **XOBE**$_{\text{DBPL}}$ program at runtime.

6 Experimental Results

We have tested our **XOBE**$_{\text{DBPL}}$ prototype on four XML documents generated by `xmlgen` of the XMark [SWK$^+$02] Benchmark project. We used rather small scaling factors for `xmlgen` resulting in XML documents with sizes reaching from kilobytes to 3 megabytes. Moreover we defined four representative updates and measured times to validate and execute them. Validation and execution times of the **XOBE**$_{\text{DBPL}}$ prototype implementation were compared with three other approaches all referenced in section 2 on related work. Validation of updates in context of the approaches using DOM and Infonyte[In03] as well as Xindice[Ap04] is tested at runtime by parsing the whole DOM against the schema. Contrary **XOBE**$_{\text{DBPL}}$ and Xact are based on static validation, which is independent of the XML document's size. The four updates we have chosen are defined as follows.

Update 1. deletes an existing person of the auction site by its id.

Update 2. deletes all closed auctions elements of the auction site.

Update 3. inserts a new person into the people element of the auction site.

Update 4. is an update operation consisting of a delete operation as defined in the first update and an insert as defined in update 3.

The four tables 2,3,4,5 present the results. The four rows represent the four different sized XML documents generated by the XMark generator. Each record consists of two numbers representing static validation time as well as the execution time of the valid update measured in seconds respectively. Latter times do not include former times. A `'-'` for static validation time indicates that static validiation is not supported. The leftmost column contains the scaling factors of the XMark project. A scaling factor of `1.0` produces an XML `auction` schema instance in the size of 100 MB, a scaling factor of 0.1 consequently leads to a 10 MB sized document and so forth. In particular the scaling factor `0.0` generates a minimum XML document according to the `auction` schema. As we limit the size of the largest XML document to 3 Megabyte we could achieve that all operations take place in main memory. This was required because Xact does not contain a database connection, but is the only other available prototype supporting static validation and XML manipulation. Consequently measured times are not influenced by times accessing hard disk. Moreover times to load and/or store the documents are not included.

The experimental results show that **XOBE**$_{\text{DBPL}}$ is very well suited to replace existing approaches based on DOM and rather low-level APIs. Especially times needed to statically validate the four updates are very small and much better than Xact's. Even for the chosen, rather small XML documents, approaches like `Infonyte+DOM` and `Xindice` suffer from the time to check the validity of updates at runtime. And contrary to static validation times, these times will grow as the size of the documents grow.

We run our tests using Sun's Java 1.4.1 virtual machine on a 2.0 GHz Pentium 4 with 768 MB RAM. Each test was run repeatedly to get average times.

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.23	13.5/13.64	-/0.36	-/0.37
0.01	0.17/0.52	13.5/13.64	-/0.69	-/2.09
0.02	0.17/0.83	13.5/13.64	-/0.87	-/2.67
0.03	0.17/1.06	13.5/13.67	-/1.07	-/3.96

Figure 2: Update 1. Deletion of an existing person by its id[2].

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.19	11.7/11.87	-/0.36	-/0.45
0.01	0.17/0.19	11.7/12.17	-/0.99	-/1.49
0.02	0.17/0.19	11.7/12.47	-/1.48	-/2.57
0.03	0.17/0.19	11.7/12.7	-/3.96	-/4.76

Figure 3: Update 2. Deletion of all closed auction elements[2].

7 Concluding Remarks

This paper presented updates in **XOBE**$_{\text{DBPL}}$ a novel XML, JAVA and **XOBE** based database programming language. **XOBE**$_{\text{DBPL}}$ combines JAVA with XML by introducing XML ob-

[2]Times of validation at compile time and execution of valid updates are given in seconds. The scaling factor 0.0 produces a minimal `auction` XML document.

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.19	13.34/13.54	-/0.35	-/0.47
0.01	0.17/0.23	13.34/14.24	-/0.95	-/1.32
0.02	0.17/0.24	13.34/14.94	-/1.31	-/2.37
0.03	0.17/0.25	13.34/15.14	-/1.66	-/3.96

Figure 4: Update 3. Insert of a new person into the people element[2].

XMark scaling factor	**XOBE**$_{\text{DBPL}}$	Xact	DOM+Infonyte	Xindice
0.0	0.17/0.4	22.14/22.35	-/0.52	-/0.61
0.01	0.17/0.73	22.14/22.37	-/1.47	-/3.18
0.02	0.17/1.1	22.14/44.64	-/2.74	-/5.24
0.03	0.17/1.3	22.14/47.24	-/3.62	-/7.52

Figure 5: Update 4. Update operations 1 and 3 are combined[2].

jects which represent XML fragments. To update and query (persistent) XML objects **XOBE**$_{\text{DBPL}}$ integrates xFLWOR(e**x**tended FLWOR) expressions. The validity of update operations in a **XOBE**$_{\text{DBPL}}$ program according to the declared schema is checked by the **XOBE**$_{\text{DBPL}}$'s type system at compile time. The new set of type inference rules for update operations in **XOBE**$_{\text{DBPL}}$ was presented and demonstrated using the XMark[SWK^{+}02] project. The architecture of our running prototype including a connection to a native XML database system was described. Moreover experimental results comparing **XOBE**$_{\text{DBPL}}$ with some related approaches [KMS04, In03, Ap04] were presented.

In the future we plan to further develop **XOBE**$_{\text{DBPL}}$ to a full database programming language, including optimization techniques and inherent database connectivity as well as type independent persistency and transactions for multi user access. In particular, XML indexing techniques will be looked at in order to speed up XPath expressions, which are an integral part of any xFLWOR expression. Especially, we plan to integrate a new index concept for XML called KeyX[HKL04] into **XOBE**$_{\text{DBPL}}$. In this context, it will be important to evaluate other native XML database systems and object relational systems in exchange to Infonyte DB[In03] in order to find out which system is best suited for the needs of an XML database programming language.

Acknowledgments The authors would like to thank Martin Kempa for his pioneering work on **XOBE** and for critical comments on this paper.

References

[Ap01] Apache XML Project, T. Xerces Java Parser. `http://xml.apache.org/xerces-j/index.html`. 15. November 2001. Version 1.4.4.

[Ap03] Apache XML Beans Project, T. Apache XML Beans. `http://xml.apache.org/xmlbeans/index.html`. 19. June 2003. Version 2.0.

[Ap04] Apache Xindice Project, T. Xindice. `http://xml.apache.org/xindice/index.html`. Januar 2004. Version 2.0.

[BA03] Bouchou, B. und Alves, M. H. F.: Updates and Incremental Validation of XML Documents. In: *Proceedings of the 9th International Conference on Data Base Programming Languages(DBPL)*. Potsdam, Germany. 6-8. September 2003.

[BKMW01] Brüggemann-Klein, A., Murata, M., und Wood, D.: Regular tree and regular hedge languages over unranked alphabets: Version 1. Technical Report HKUST-TCSC-2001-05. Hong Kong University of Science & Technology. April 3 2001. Theoretical Computer Science Center.

[BMS02] Brabrand, C., Møller, A., und Schwartzbach, M. I.: The bigwig project. In: *ACM Transactions on Internet Technology*. volume 2(2). S. 79–114. ACM. 2002.

[Bo01] Borland: *XML Application Developer's Guide, JBuilder*. Borland Software Corporation. Scotts Valley, CA. 1997,2001. Version 5.

[Bo02] Bourret, R. XML Data Binding Resources. web document, `http://www.rpbourret.com/xml/XMLDataBinding.htm`. 28. July 2002.

[CMS03] Christensen, A. S., Møller, A., und Schwartzbach, M. I.: Extending java for high-level web service construction. In: *ACM Transactions on Programming Languages and Systems*. volume 25(6). S. 814–875. ACM. 2003.

[CX00] Cheng, J. und Xu, J.: Xml and db2. In: *Proceedings of the 16th IEEE International Conference on Data Engineering (ICDE)*. S. 569–576. IEEE. 2000.

[Ec99] Ecma International. EcmaScript Language Specification. `http://www.ecma-international.org/`. December 1999. Edition 3.0.

[Ec04] Ecma International. EcmaScript for XML Specification. `http://www.ecma-international.org/`. June 2004. Edition 1.0.

[Ex01] ExoLab Group. Castor. ExoLab Group, `http://castor.exolab.org/`. 2001.

[FGK02] Florescu, D., Grünhagen, A., und Kossmann, D.: XL: An XML Programming Language for Web Service Specification and Composition. In: *Proceedings of International World Wide Web Conference (WWW 2002), May 7-11, Honolulu, Hawaii, USA*. S. 65–76. ACM. 2002. ISBN 1-880672-20-0.

[FHK$^+$02] Fiebig, T., Helmer, S., Kanne, C.-C., Moerkotte, G., Neumann, J., Schiele, R., und TillWestmann: Anatomy of a native XML base management system. In: *The VLDB Journal*. volume 11. S. 292–314. 2002.

[GP03] Gapayev, V. und Pierce, B. C.: Regular object types. In: *ECOOP 2003, Lecture Notes in Computer Science 2743*. S. 151–175. Springer-Verlag. 2003.

[HKL04] Hammerschmidt, B. C., Kempa, M., und Linnemann, V.: A selective key-oriented XML Index for the Index Selection Problem in XDBMS. In: *DEXA Conference August 30 - September 3, 2004, Lecture Notes in Computer Science*. Springer-Verlag. 2004.

[HP03] Hosoya, H. und Pierce, B. C.: Xduce: A statically typed xml processing language. In: *ACM Transactions on Internet Technology*. volume 3(2). S. 117–148. ACM. 2003.

[HRS+03] Harren, M., Raghavachari, M., Shmueli, O., Burke, M., Sarkar, V., und Bordawekar, R.: XJ: Integration of XML Processing into Java. *IBM Research Report RC23007 (W0311-138)*. November 18, 2003.

[IB] IBM Corporation. IBM DB2 XML Extender. URL: `http://www-3.ibm.com/software/data/db2/extenders/xmlext/`.

[In03] Infonyte GmbH. Infonyte DB. URL: `http://www.infonyte.com`. 2003.

[JD] JDOM Project. JDOM FAQ. `http://www.jdom.org/docs/faq.html`.

[Ke03] Kempa, M.: *Programmierung von XML-basierten Anwendungen unter Berücksichtigung der Sprachbeschreibung*. PhD thesis. Institut für Informationssysteme, Universität zu Lübeck. 2003. Aka Verlag, Berlin, (in German).

[KL03] Kempa, M. und Linnemann, V.: Type Checking in XOBE. In: Weikum, G., Schöning, H., und Rahm, E. (Hrsg.), *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung,*. volume P-26 of *Lecture Notes in Informatics*. S. 227–246. Gesellschaft für Informatik. 26.-28. Februar 2003.

[KLL03] Kim, S.-K., Lee, M., und Lee, K.-C.: Validation of XML Document Updates Based on XML Schema in XML Databases. volume 2736 of *Lecture Notes in Computer Science (LNCS)*. S. 98–108. Heidelberg. 2003. Springer-Verlag.

[KMS04] Kirkegaard, C., Møller, A., und Schwartzbach, M. I.: Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*. 30(3):181–192. March 2004.

[Le01] Lehti, P.: Desing and Implementation of a Data Manipulation Processor for an XML Query Processor. Master's thesis. Technical University of Darmstadt, Darmstadt. August 2001.

[LLW03] Liu, M., Lu, L., und Wang, G.: A Declarative XML-R Update Language. volume 2831 of *Lecture Notes in Computer Science (LNCS)*. S. 506–519. Heidelberg. 2003. Springer-Verlag.

[MB03] Murthy, R. und Banerjee, S.: XML Schemas in Oracle XML DB. In: *Proceedings of the 29th VLDB Conference, Berlin, Germany*. S. 1009–1018. 2003.

[Mi01] Microsoft Corporation. .NET Framework Developer's Guide. web document, `http://msdn.microsoft.com/library/default.asp`. 2001.

[MSB03] Meijer, E., Schulte, W., und Biermann, G. Programming with Circles, Triangles and Rectangles. `http://www.cl.cam.ac.uk/~gmb/Papers/vanilla-xml2003.html`. 2003.

[Or01] Oracle Corporation: *Oracle9i, Application Developer's Guide – XML, Release 1 (9.0.1)*. Redwood City, CA 94065, USA. June 2001. Shelley Higgins, Part Number A88894-01.

[Or03] Oracle Corporation. Oracle XML DB. URL: `http://otn.oracle.com/tech/xml/xmldb/index.html`. 2003.

[PLC99] Pelegrí-Llopart, E. und Cable, L. JavaServer Pages Specification, Version 1.1. Java Software, Sun Microsystems, `http://java.sun.com/products/jsp/download.html`. 30. November 1999.

[PV03] Papakonstantinou, Y. und Vianu, V.: Incremental Validation of XML Documents. volume 2572 of *Lecture Notes in Computer Science (LNCS)*. S. 47–63. Heidelberg. 2003. Springer-Verlag.

[Sc01] Schöning, H.: Tamino - A DBMS designed for XML. In: *Proceedings of the 17th International Conference on Data Engineering*. S. 149–154. Heidelberg, Germany. April 2-6 2001. IEEE Computer Society.

[SHS04] Sur, G. M., Hammer, J., und Simeon, J.: UpdateX - An XQuery-Based Language for Processing Updates in XML. In: *International Workshop on Programming Language Technologies for XML(PLAN-X 2004)*. S. 40–53. January 2004.

[SKC+03] Su, H., Kane, B., Chen, V., Diep, C., Guan, D. M., Look, J., und Rundensteiner, E.: A Leightweight XML Constraint Check and Update Framework. volume 2784 of *Lecture Notes in Computer Science (LNCS)*. S. 39–50. Heidelberg. 2003. Springer-Verlag.

[Su01] Sun Microsystems, Inc. Java 2 Platform, Standard Edition, v 1.3.1, API Specification. `http://java.sun.com/j2se/1.3/docs/api/index.html`. December 2001.

[SWK+02] Schmidt, A., Waas, F., Kersten, M., Florescu, D., Manolescu, I., Carey, M., und Busse, R.: XMark: A Benchmark for XML Data Management. In: *International Conference on Very Large Data Bases(VLDB'02)*. S. 974–985. Hong Kong. August 2002.

[TIHW01] Tatarinov, I., Ives, Z. G., Halevy, A. Y., und Weld, D. S.: Updating XML. In: *ACM Sigmod Conference 2001*. S. 413–424. ACM. 2001.

[TWP00] Tao, K., Wang, W., und Palsberg, D. J. Java Tree Builder JTB. `http://www.cs.purdue.edu/jtb/`. 15. May 2000. Version 1.2.2.

[W398] W3Consortium. Document Object Model (DOM) Level 1 Specification, Version 1.0. Recommendation, `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`. 1. October 1998.

[W399] W3Consortium. XML Path Language (XPath), Version 1.0. Recommendation, `http://www.w3.org/TR/xpath`. 16. November 1999.

[W302] W3Consortium. Updates for XQuery. Working Draft, unpublished. October 2002.

[W304] W3Consortium. XQuery 1.0: An XML Query Language. Working Draft, `http://www.w3.org/TR/2002/WD-xquery-20041029/`. 29. October 2004.

[We02] WebGain. Java Compiler Compiler (JavaCC) – The Java Parser Generator. `http://www.webgain.com/products/java_cc/`. 2002. Version 2.1.

[Wi99] Williamson, A. R.: *Java Servlets by Example*. Manning Publications Co. Greenwich. 1999.

[XM04] XML Database Initiative(XML:DB). XUpdate. `http://xmldb-org.sourceforge.net/xupdate`. 2004.

[xml] About SAX. `http://sax.sourceforge.net`.

[YXGZ03] Yue, K., Xu, Z., Guo, Z., und Zhou, A.: Constraint Preserving XML Updating. volume 2642 of *Lecture Notes in Computer Science (LNCS)*. S. 47–58. Heidelberg. 2003. Springer-Verlag.

A Auction Schema

The `auction` schema elements and types of the XMark[SWK+02] project, which are used in this paper.

```
<schema>
        <element name="site">
         <complexType>
          <sequence>
           <element name="regions" type="regionsType"/>
           <element name="categories" type="categoriesType"/>
           <element name="catgraph" type="catgraphType"/>
           <element name="people" type="peopleType"/>
           <element name="open_auctions" type="open_auctionsType"/>
           <element name="closed_auctions" type="closed_auctionsType"/>
          </sequence>
         </complexType>
        </element>
        <!--people          -->
        <complexType name="peopleType">
         <sequence>
          <element name="person" minOccurs="0" maxOccurs="unbounded"/>
           <complexType>
            <sequence>
             <element name="name" type="string"/>
             <element name="emailaddress" type="string"/>
             <element name="creditcard" minOccurs="0" type="string"/>
             <element name="watches" type="watchesType" minOccurs="0"/>
            </sequence>
            <attribute name="id" use="required"/>
           </complexType>
          </element>
        </sequence>
        </complexType>
        <!--open auctions -->
        <complexType name="open_auctionsType">
         <sequence>
          <element name="open_auction" type="open_auctionType" minOccurs="0"
               maxOccurs="unbounded"/>
         </sequence>
        </complexType>
        <complexType name="open_auctionType">
         <sequence>
          <element name="initial" type="string"/>
          <element name="bidder" type="bidderType" minOccurs="0" maxOccurs="
               unbounded"/>
          <element name="current" type="string"/>
          <element name="itemref" type="itemrefType"/>
          <element name="seller" type="sellerType"/>
          <element name="quantity" type="string"/>
         </sequence>
         <attribute name="id" use="required"/>
        </complexType>
        <!--bidder       -->
        <complexType name="bidderType">
         <sequence>
          <element name="date" type="string"/>
          <element name="time" type="string"/>
          <element name="personref">
           <complexType>
            <attribute name="person" use="required"/>
           </complexType>
          </element>
          <element name="increase" type="string"/>
         </xs:sequence>
        </xs:complexType>
</schema>
```