Type Safe Programming of XML-based Applications

Martin Kempa
sd&m AG
software design & management
Carl-Wery-Str. 42
D-81739 München, Germany

Volker Linnemann
Universität zu Lübeck
Institut für Informationssysteme
Ratzeburger Allee 160, Geb. 64
D-23538 Lübeck, Germany
E-mail: linnemann@ifis.uni-luebeck.de

Abstract: There is an emerging amount of software for generating and manipulating XML documents. This paper addresses the problem of guaranteeing the validity of dynamically generated XML structures statically at compile time of an XML-based application. In the XOBE (**X**ML **OBJE**CTS) project we extend the object-oriented programming language Java by new language constructs. XML Schema is used for describing sets of valid XML documents. An XML schema provides a vehicle to define new classes, i.e. each element declaration in a schema defines a new class of objects (XML objects). Each object within a class represents an XML structure which is valid according to the underlying XML schema. XML objects are created by a new language construct called XML object constructor. XML object constructors are expressed in XML syntax. Previously generated XML objects can be inserted according to the declared XML schema.
The main focus of the paper is the type system of XOBE. Among others, this type system provides the basis for checking the validity of assignments of XML objects to variables. The type system will be described and we present formally a type checking algorithm based on this type system.

1 Introduction

XML [W3C] plays an important role for 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. The approaches that are currently in use are not sufficient because they cannot guarantee that only valid XML documents are being dealt with. In the XML context a valid XML document is a document which is correct according to an underlying XML Schema [W3C] or an XML Document Type Definition DTD [ABS00] which we call schema in the remainder of this paper. This means that a document is an element of the language defined by a schema. Since most current languages and tools do not allow to guarantee the validity of dynamically generated XML documents at compile time, extensive runtime checking is necessary in order to achieve valid documents.

The **X**ML **OBJE**CTS project (XOBE) [LK02] at the University of Lübeck addresses this mismatch by defining XML objects representing XML fragments and by treating them as first-class data values. We extend Java for this purpose. XOBE overcomes the different representations of XML fragments as strings and as nested object structures in the same

source code. Instead, a running XOBE program works only with XML objects. A text form of XML objects with explicit tagging is constituted for communication with the outside world only. XML objects are created by a new language construct called XML object constructor. XML object constructors are expressed in XML syntax. Previously generated XML objects can be inserted according to the declared schema. The schema is used for typing the XML objects.

The main aspect of this paper is the type system of XOBE. This type system is the basis for checking the validity of assignments of XML objects to variables. The type system will be described and we present a type checking algorithm based on this type system.

The paper is organized as follows. Our approach for programming XML based applications by using XML objects in Java is introduced in Section 2. The corresponding type system is described informally and formally in Section 3. Section 4 presents implementation details of the XOBE system. Section 5 provides some related work. Concluding remarks and an outlook for future work conclude the paper.

2 XML Objects in XOBE

In this section we briefly introduce the syntax and semantics of XML objects in an informal manner. A more detailed introduction can be found in [KL02, KL03, Kem03]. XOBE extends the object-oriented programming language Java by language constructs to process XML fragments. Due to space limitations, we introduce only the construction of XML objects. Traversing XML objects by XPATH [W3C] is not needed for describing the type system of XOBE. Details can be found in [KL03, Kem03].

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 given schema is used to type different XML objects.

We use the schema given in listing 1 describing a bookstore as the basis for our example. According to this schema, a bookstore contains several books. Each book contains several authors and one title.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="bookstore"><xsd:complexType><xsd:sequence>
      <xsd:element name="book"  minOccurs="0" maxOccurs="unbounded">
        <xsd:complexType><xsd:sequence>
          <xsd:element name="author" minOccurs="0" maxoccurs="unbounded"
                                                  type="xsd:string"/>
          <xsd:element name="title" type="xsd:string"/>
        </xsd:sequence></xsd:complexType>
      </xsd:element>
  </xsd:sequence></xsd:complexType></xsd:element>
</xsd:schema>
```

Listing 1: XML schema for `bookstore`

Listing 2 shows a method which reads books with their authors and titles. It subsequently constructs an XML object for these books. The class declaring `collectBooks` uses our bookstore schema by importing the corresponding file by a clause like `import bookstore.xsd`.

```
1  bookstore collectBooks(){
2    XML<book*> books = <>;
3    int countBooks = keyboard.readint();
4    for (int i=1; i<=countBooks; i++){
5      XML<author*> authors = <>;
6      int countAuthors = keyboard.readint();
7      for (int j=1; j<=countAuthors; j++) {
8        String author = keyboard.readString();
9        authors = authors+
10           <author> {author} </author>;
11     } // for j
12     String title = keyboard.readString();
13     book b =
14       <book>
15         {authors}
16         <title>{title}</title>
17       </book>;
18     books = books + b;
19   } // for i
20   return
21       <bookstore>
22           {books}
23       </bookstore>;
24 } //collectBooks
```

Listing 2: Method `collectBooks`

Line 2 declares a variable `books` for XML objects. A corresponding value is a list of books. `books` is initialized by the empty list. Line 5 declares a variable `authors` for lists of authors. Lines 9 and 10 append a newly created author element to the list of authors. The author element is constructed in line 10 by a so called **XML object constructor**. An XML object constructor is an XML fragment where the content of a variable can be inserted in places where it is allowed according to the underlying schema. In line 10, the string content of variable author is inserted. This conforms with the underlying schema because the content of an author element is a string. In line 13, the declaration `book b` is an abbreviation for `XML<book> b`. An abbreviation of this kind is allowed in cases where the type of a variable corresponds directly to an element name, e.g. book. This does not apply to variable `books` (line (2)). Lines 13-17 use an XML object constructor for constructing a book out of the previously generated author list and the title.

This finishes our short introduction to XOBE. Due to space limitations, we did not cover the important aspect of using XPath [W3C] within XOBE for type safe decomposition of XML values. For details and more examples we refer to [Kem03, KL03, Kra02, LK02].

3 The XOBE Type System

This section is the major part of this paper presenting the XOBE type system. Type analysis is done in two steps. First, the types of the XML object expressions are determined using *type inference*. Second, the subtype relationship of the inferred types is checked by a *subtyping algorithm*. Details are given in [Kem03, KL03].

Formalizing XML

For a precise definition of our subtyping algorithm we need a formalization of XML. We use *regular hedge expressions* [BKMW01] and *regular hedge grammars* describing sets

of trees.

Definition 3.1 Let B and E be finite sets of basic types (simple types) and element names respectively with $B \cap E = \emptyset$. A *set of all hedges* T^* over a set of terminal symbols $T = B \cup E$ is defined inductively as follows:

$\epsilon \in T^*$ is the empty hedge, $\quad e[v] \in T^*$ is a hedge with $e \in E$ and $v \in T^*$,
$b \in T^*$ is a hedge with $b \in B$, $\quad vw \in T^*$ is a hedge with $v \in T^*$ and $w \in T^*$. □

Definition 3.2 The set of *regular hedge expressions Reg* over a set of terminal symbols $T = B \cup E$ and a set N of nonterminal symbols (names of groups and complex types, $N \cap T = \emptyset$) is defined recursively by:

$$\begin{array}{rlrl} \emptyset & \in \textit{Reg} \text{ (empty set)}, & \epsilon & \in \textit{Reg} \text{ (empty hedge)}, \\ b & \in \textit{Reg} \text{ (simple type)}, & n & \in \textit{Reg} \text{ (complex type)}, \\ e[r] & \in \textit{Reg} \text{ (element)}, & (r|s) & \in \textit{Reg} \text{ (regular union)}, \\ (r;s) & \in \textit{Reg} \text{ (regular concatenation)}, & (r)* & \in \textit{Reg} \text{ (Kleene star)} \end{array}$$

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

For simplifying parentheses, we assume operator precedence in the decreasing order $*$; $|$.

We define $r+ = r;r*$, $r? = r|\epsilon$ and for $I = \{i_1, ..., i_n\}$: $\quad \Big|_{i \in I} r_i = r_{i_1} \mid ... \mid r_{i_n}$.

Definition 3.3 The *hedge language* $L(r)$ over a regular hedge expression $r \in$ *Reg* for a given set of production rules P of the form $n \to r$ with $n \in N$, $r \in Reg$ and $n \to r \in P$, $m \to r' \in P \Longrightarrow n \neq m$, is defined by:

$$\begin{array}{rclrcl} L(\emptyset) & = & \{\} & L(\epsilon) & = & \{\epsilon\} \\ L(b) & = & \{b\} & L(n) & = & L(r) \text{ with } n \to r \in P \\ L(e[r]) & = & \{e[u] | u \in L(r)\} & L(r|s) & = & L(r) \cup L(s) \\ L(r;s) & = & \{uv | u \in L(r), v \in L(s)\} & L(r*) & = & \{\epsilon\} \cup L(r;r*) \end{array}$$

for all $b \in B$, $n \in N$, $e \in E$ and $r, s \in$ *Reg*. ϵ denotes the empty hedge. □

The predicate $isnullable? : Reg \to \{\textbf{true},\textbf{false}\}$ decides $\epsilon \in L(r)$ for $r \in Reg$.

Definition 3.4 A *regular hedge grammar* is defined by $G = (T, N, s, P)$ with T, N, P as defined in definitions 3.2, 3.3. $s \in$ *Reg* is a start expression. Each rule $n \to r \in P$ has to fulfill the following two conditions guaranteeing regularity of the grammar:

1. If the nonterminal symbol n is defined recursively, the recursive application has to be in the last position of the regular expression r.
2. If a nonterminal symbol n is defined recursively, the expression s in front of the recursive application has to fulfill $\neg$*isNullable?*(s). □

Definition 3.5 A *regular inequality* of two regular hedge expressions r and s is defined by: $\quad r \leq s \Leftrightarrow L(r) \subseteq L(s)$. □

Example 3.1 The following regular hedge grammar corresponds to the schema in listing 1. Element names and simple types are **`boldfaced`**, nonterminal symbols are *italic*. The start expression s is *bookstore*.

bookstore → **`bookstore`**[*book*$*$] $\quad$ *book* → **`book`**[*author*$*$; *title*]
author → **`author`**[**`string`**] $\quad$ *title* → **`title`**[**`string`**] .

For example, the following regular inequalities hold:

`book`[*author*$*$; *title*] ≤ **`book`**[*author*$*$; *title*]
`book`[*author* ; *author*$*$; *author* ; *title*] ≤ **`book`**[*author*$*$; *title*] .

Type Inference

The type corresponding to an XML object constructor is given by a regular hedge expression. An inserted variable is replaced by the symbol of the hedge expression which corresponds to the declared type of the variable. Type inference for XML object constructors is quite simple because all variables have to be declared in XOBE.

Example 3.2 The type of the left hand side of the assignment in lines 13-17 in listing 2 is *book*. The type of the right hand side is **book**[*author*$*$; **title**[**string**]].

The subtyping algorithm which is described in the next subsection has to check the regular inequality **book**[*author*$*$; **title**[**string**]] $\leq$ *book*.

Subtyping Algorithm

After inferring the types of the XML objects in a XOBE program, the type system checks the correctness of the concerned statements using the subtyping algorithm. For this we adopt Antimirov's algorithm [Ant94] for checking inequalities of regular expressions and extend it to the hedge grammar case. The idea of the algorithm is that for every invalid regular inequality there exists at least one reduced inequality which is *trivially inconsistent*, a notion which is defined by the function *inc* as follows.

Definition 3.6 A regular inequality $r \leq s$ is called *trivially inconsistent* if
$inc(r \leq s) = (isNullable?(r) \wedge \neg isNullable?(s))$ holds. □

Definition 3.7 For $r \in$ *Reg* the set of leading terminal symbols is defined by
$term(r) = \{t \mid t \in T \ \wedge \ (tw \ \in L(r) \vee \ t[v]w \in L(r)$ with hedges $v, w)\}$. □

The reduction of regular inequalities is expressed by partial derivatives *der* of regular hedge expressions. They formalize the set of hedges which can follow after an already recognized terminal symbol. A partial derivative consists of pairs of expressions. The first component of a pair stands for the element content of the reduced terminal symbol, i.e. the child dimension. The second component represents the expression part after the reduced terminal symbol, i.e. the sibling dimension. Due to space limitations, we present only some examples in this paper. For details see [Kem03].

In the following example the regular expression **author**[**string**]; *author*$*$ is reduced by the given terminal symbol **author**. According to Def. 3.2, ";" denotes regular concatenation. "," separates elements of pairs and elements of sets:

$$der_{\texttt{author}}(\textbf{author}[\textbf{string}]; author*) = \{(\textbf{string}, author*)\} \quad .$$

The result is a set of regular hedge expression pairs because we can receive multiple pairs as the following example shows:

$$der_{\texttt{book}}(\textbf{book}[title]|\textbf{book}[author; title]) = \{(title, \epsilon), (author; title, \epsilon)\} \quad .$$

Please notice that in the case of Kleene star operations the length of the resulting expressions can increase:

$$der_{\texttt{author}}((\textbf{author}[\textbf{string}]; title)*) = \{(\textbf{string}, title; (\textbf{author}[\textbf{string}]; title)*)\}.$$

Based on the partial derivatives of regular expressions we can define partial derivatives of regular inequalities. This definition is quite complex in the hedge grammar case because of the two dimensions. For the definition we adopt a set-theoretic observation from [HVP00]. Examples will follow in example 3.3.

Definition 3.8 A *partial derivative* of a regular inequality $r \leq s$ with $r, s \in \mathit{Reg}$ with respect to a terminal symbol $x \in T$ is defined by

$$\mathit{part}_x(r \leq s) = \{\ (c_r \leq \Big|_{i \in I} c_s^i)\ \vee\ (r_r \leq \Big|_{i \in \overline{I}} r_s^i) \mid (c_r, r_r) \in \mathit{der}_x(r)\ \wedge$$
$$\mathit{der}_x(s) = \{(c_s^1, r_s^1), \ldots, (c_s^n, r_s^n)\}\ \wedge$$
$$I\ \in \mathcal{P}(\{1, \ldots, n\})\ \wedge\ \overline{I} = \{1, \ldots, n\} \backslash I\ \}$$

with $c_r, r_r, c_s^i, r_s^i \in \mathit{Reg}$ and $\mathcal{P}(I) = \{J \mid J \subseteq I\}$. □

The subtyping algorithm is defined by two *subtyping judgements* $\Gamma \vdash r \leq s \Rightarrow \Gamma'$ and $\Gamma \vdash^* r \leq s \Rightarrow \Gamma'$ with a set Γ of regular inequalities of type $t \leq u$. Both judgements have to be interpreted as: "The algorithm proves $r \leq s$ and all inequalities $t \leq u$ in Γ are not trivially inconsistent. All results are returned in the set Γ' with all regular inequalities of the partial derivatives of $r \leq s$". Because the production rules of the hedge grammar can be defined recursively it can happen that an already calculated inequality appears during the algorithm later on. To ensure termination in that case we save all already seen inequalities in the set Γ. For this reason we have to introduce the two subtyping judgements. With judgement $\vdash^*$ we indicate that an inequality has been added to Γ.

Definition 3.9 The *subtyping algorithm* is defined by the following rules:

$$\frac{r \leq s \in \Gamma}{\Gamma \vdash r \leq s \Rightarrow \Gamma} \qquad \text{(HYP)}$$

$$\frac{\begin{array}{c} r \leq s \notin \Gamma, \\ \Gamma \cup \{r \leq s\} \vdash^* r \leq s \Rightarrow \Gamma' \end{array}}{\Gamma \vdash r \leq s \Rightarrow \Gamma'} \qquad \text{(ASSUM)}$$

$$\frac{\begin{array}{c} \neg \mathit{inc}(r \leq s), \\ \text{For all } x \in \mathit{term}(r)\ \mathit{and}\ \text{for all } i \in \{1, \ldots, k\} \text{ with } \mathit{part}_x(r \leq s) = \\ \{(c_{r,1} \leq c_{s,1} \vee r_{r,1} \leq r_{s,1}), ..., (c_{r,k} \leq c_{s,k} \vee r_{r,k} \leq r_{s,k})\} \text{ is} \\ \Gamma_{i-1} \vdash c_{r,i} \leq c_{s,i} \Rightarrow \Gamma_i \vee \Gamma_{i-1} \vdash r_{r,i} \leq r_{s,i} \Rightarrow \Gamma_i \end{array}}{\Gamma_0 \vdash^* r \leq s \Rightarrow \Gamma_k} \qquad \text{(REC)}$$

□

With rule HYP we test if the inequality in question is already in the set of all calculated inequalities Γ terminating that recursion branch. Rule ASSUM switches between the two judgements $\vdash$ and $\vdash^*$ adding the inequality to Γ. The rule REC is applicable if the inequality is not trivially inconsistent. With operation *part* all partial derivatives are calculated and checked recursively.

Example 3.3 Consider the regular inequality $\mathit{author}*; \mathit{title} \leq \mathit{author}; \mathit{author}*; \mathit{title} | \mathit{title}$. We start with $\Gamma_0 = \emptyset$. The computations and the derivation tree of the execution are given in figure 1. The inequality is accepted finally.

Due to the regularity of the production rules the subtyping algorithm is guaranteed to terminate. For a detailed correctness proof we refer to [Kem03].

Complexity and Extensions

The complexity of the subtyping algorithm is EXPTIME complete as shown in [Sei90]. This means that in the worst case the number of checked inequalities depends exponentially on the length of the given inequality. Nevertheless, in contrast to the classic procedure using a tree automaton the algorithm works more efficiently in some cases. In the

$$term(\mathit{author}*;\mathit{title}) = \{\texttt{author}, \texttt{title}\}$$
$$der_{\texttt{author}}(\mathit{author}*;\mathit{title}) = \{(\texttt{string}, \mathit{author}*;\mathit{title})\}$$
$$der_{\texttt{author}}(\mathit{author};\mathit{author}*;\mathit{title}|title) = \{(\texttt{string}, \mathit{author}*;\mathit{title})\}$$
$$der_{\texttt{title}}(\mathit{author}*;\mathit{title}) = \{(\texttt{string}, \epsilon)\}$$
$$der_{\texttt{title}}(\mathit{author};\mathit{author}*;\mathit{title}|title) = \{(\texttt{string}, \epsilon)\}$$
$$part_{\texttt{author}}(\mathit{author}*;\mathit{title} \leq \mathit{author};\mathit{author}*;\mathit{title}|\mathit{title}) =$$
$$\{\quad (\texttt{string} \leq \texttt{string} \vee \mathit{author}*;\mathit{title} \leq \emptyset), \qquad (1)$$
$$(\texttt{string} \leq \emptyset \vee \mathit{author}*;\mathit{title} \leq \mathit{author}*;\mathit{title})\ \} \qquad (2)$$
$$part_{\texttt{title}}(\mathit{author}*;\mathit{title} \leq \mathit{author};\mathit{author}*;\mathit{title}|\mathit{title}) =$$
$$\{\quad (\texttt{string} \leq \texttt{string} \vee \epsilon \leq \emptyset), \qquad (3)$$
$$(\texttt{string} \leq \emptyset \vee \epsilon \leq \epsilon)\ \} \qquad (4)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_7 \vdash^* \epsilon \leq \epsilon \Rightarrow \Gamma_3}\,\textsc{Rec}}{\Gamma_6 \vdash \epsilon \leq \epsilon \Rightarrow \Gamma_3} \vee \dfrac{Error}{\Gamma_6 \vdash \epsilon \leq \emptyset},}{\Gamma_3 \vdash \epsilon \leq \emptyset} \vee \dfrac{}{\Gamma_3 \vdash \epsilon \leq \epsilon \Rightarrow \Gamma_3}\,\textsc{Hyp}}{\Gamma_6 \vdash^* \texttt{string} \leq \texttt{string} \Rightarrow \Gamma_3} \vee \dfrac{\begin{array}{c}Error\\ \vdots\end{array}}{\Gamma_1 \vdash \mathit{author}*;\mathit{title} \leq \emptyset}, \qquad (1)}{\Gamma_3 \vdash \texttt{string} \leq \emptyset} \vee \dfrac{\begin{array}{c}\textsc{Hyp}\\ \vdots\end{array}}{\Gamma_3 \vdash \mathit{author}*;\mathit{title} \leq \mathit{author}*;\mathit{title} \Rightarrow \Gamma_5}, \qquad (2)}{\Gamma_5 \vdash \texttt{string} \leq \texttt{string} \Rightarrow \Gamma_5} \vee \dfrac{Error}{\Gamma_5 \vdash \epsilon \leq \emptyset}, \qquad (3)}{\Gamma_5 \vdash \texttt{string} \leq \emptyset} \vee \dfrac{\textsc{Hyp}}{\Gamma_5 \vdash \epsilon \leq \epsilon \Rightarrow \Gamma_5} \qquad (4)}{\Gamma_1 \vdash^* \mathit{author}*;\mathit{title} \leq \mathit{author};\mathit{author}*;\mathit{title}|\mathit{title} \Rightarrow \Gamma_5}}{\Gamma_0 \vdash \mathit{author}*;\mathit{title} \leq \mathit{author};\mathit{author}*;\mathit{title}|\mathit{title} \Rightarrow \Gamma_5}$$

Figure 1: Proving inequality *author*$*$; *title* $\leq$ *author*; *author*$*$; *title*|*title*

classic procedure both automata representing the regular hedge expression of both sides of the inequality in question have to be made deterministic. In our algorithm the right hand side of the inequality has to be made deterministic lazily, i.e. only as much as needed.

In XML Schema and DTDs restrictions to general hedge grammars are assumed. First all element types with the same element name in one content model have to have the same content model. As shown in [Kem03] this simplifies the subtyping algorithm to a PSPACE complete complexity, which is the same as comparing regular string expressions [Ant94]. The second restriction is that the content models have to be one-unambiguous. This leads to a linear subtyping algorithm.

The subtyping algorithm described so far deals with the *structural typing* in XML. However, in XML Schema additional concepts like substitution groups, type extensions and type restrictions sometimes called *named typing* exist. This requires an extension of our regular hedge grammars. Therefore we introduce a reflexive and transitive *substitution group relation SubGr* holding the relations of element names as defined as substitution groups in the schema. Additionally the *named type relation Inh* is defined were the non-terminal types of the hedge grammar are in relation corresponding to the specified type extensions and type restrictions. Further the strategy of the subtyping algorithm has to be more sophisticated as well, because during the calulations we have to take care of the two

relations. The idea is to reduce the regular inequality by appropriate nonterminal symbols also instead of reducing only by terminal symbols. This allows to take the relations *SubGr* and *Inh* into account. This extended algorithm is described in detail in [Kem03].

4 Implementation

XOBE is realized as a Java preprocessor [Kra02]. The general architecture is shown in Figure 2.

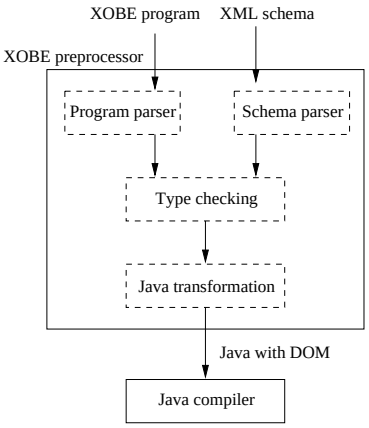

Figure 2: Architecture of the XOBE system

In our implementation we use the Java compiler compiler JavaCC [Web02] to generate the XOBE parser. Additionally we use the XML parser Xerces [Apa01] to recognize the used schemas. The internal representation of the processed XOBE program is done with the Java tree builder JTB [TWP00].

XML objects are internally stored by using the standard representation of the Document Object Model (DOM) [W3C], recommended by the W3Consortium. Please note that even though DOM is an untyped XML implementation not guaranteeing validity at compile time, the transformed XML objects in the XOBE program are valid. This holds because our type checking algorithm guarantees this property.

The XOBE system including the type checking system was, among others, successfully used in a web based real estate broker system [Kra02] and in a web based academic exercise administration system [Spi04].

5 Related Work

The most elementary way to deal with XML fragments is to use ordinary strings without any structure. For example, this technique is used in Java Servlets. Java Server Pages provide an improvement over pure string operations especially when the XML fragments are almost static, i.e. there are only a few places where dynamically generated values are inserted. There is no validation at compile time.

High level binding approaches like JAXB [Sun01] or CASTOR [Exo01] assume that all processed documents follow a given schema. Validity is only supported to a limited extent depending on the selected language mapping.

Recently, full compile time validation gained some interest. Xduce [HP03] is a special functional language developed as an XML processing language. Compile time validity is ensured by type inference and by a subtyping algorithm based on regular tree automata. Xtatic [GP03] applies the concepts of Xduce to the object-oriented programming language

C#. In contrast to XOBE, Xtatic does not support XPATH. JWig [CMS03] which is based on BigWig [BMS02] is quite close to XOBE. JWig extends Java by language constructs for XML support. Validity is guaranteed at compile time. However, in contrast to XOBE, there is no XML type system in JWig, i.e. there is only one XML type. XML fragments are generated by using XML constructors with previously generated values inserted. Validity is achieved by two data flow analyses constructing a graph which summarizes all possible XML fragments. JWig's data flow analysis is rather time consuming. Other full compile time validation approaches include Xen [MSB03], XL [FGK02], XJ [HRS+03] and WASH [Thi].

6 Concluding Remarks, Outlook for Future Work

This paper presented a short overview over the XOBE project and concentrated on the type system of XOBE. XOBE is an extension of the programming language Java, addressing the programming purposes of web applications and web services. The language extension combines Java with XML by introducing XML objects which represent XML fragments. XML objects are created using XML object constructors. In XML object constructors, previously generated XML objects can be inserted in places which are allowed according to the declared schema. The validity of all XML objects within a program is checked by the XOBE type system at compile time.

We are currently working on the integration of the XML query language XQuery [W3C] into XOBE. Allowing XML objects to be persistent and allowing to modify XML objects results then in an XML-based database programming language [SKL04].

Moreover, in addition to our real-estate brokering system [Kra02] and our academic exercise administration system [Spi04] we plan to use XOBE in other application areas.

References

[ABS00] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web, From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.

[Ant94] Valentin Antimirov. Rewriting Regular Inequalities. In Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *Lecture Notes in Computer Science (LNCS)*, pages 116–125. Springer-Verlag, 1994.

[Apa01] The Apache XML Project. Xerces Java Parser. `http://xml.apache.org/xerces-j/index.html`, 15. November 2001. Version 1.4.4.

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets: Version 1. Technical Report HKUST-TCSC-2001-05, Hong Kong University of Science & Technology, Theoretical Computer Science Center, April 3 2001.

[BMS02] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The BIGWIG project. In *ACM Transactions on Internet Technology*, volume 2(2), pages 79–114. ACM, 2002.

[CMS03] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for High-Level Web Service Construction. In *ACM Transactions on Programming Languages and Systems*, volume 25(6), pages 814–875. ACM, 2003.

[Exo01] ExoLab Group. Castor. ExoLab Group, `http://castor.exolab.org/`, 2001.

[FGK02] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. 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*, pages 65–76. ACM, 2002. ISBN 1-880672-20-0.

[GP03] Vladimir Gapayev and Benjamin C. Pierce. Regular Object Types. In *ECOOP 2003*, volume 2743 of *Lecture Notes in Computer Science (LNCS)*, pages 151–175. Springer-Verlag, 2003.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A Statically Typed XML Processing Language. In *ACM Trans. on Internet Technology*, volume 3(2), pages 117–148, 2003.

[HRS+03] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael Burke, Vivek Sarkar, and Rajesh Bordawekar. XJ: Integration of XML Processing into Java. *IBM Research Report RC23007 (W0311-138)*, November 18, 2003.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular Expression Types for XML. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada*, volume 35(9) of *SIGPLAN Notices*, pages 11–22. ACM, September 18-21 2000. ISBN 1-58113-202-6.

[Kem03] Martin Kempa. *Programmierung von XML-basierten Anwendungen unter Berücksichtigung der Sprachbeschreibung*. PhD thesis, Institut für Informationssysteme, Universität zu Lübeck, 2003. Aka Verlag, Berlin, (in German).

[KL02] Martin Kempa and Volker Linnemann. VDOM and P-XML – Towards A Valid Programming Of XML-based Applications. *Information and Software Technology, Elsevier Science B. V.*, pages 229–236, 2002. Special Issue on Objects, XML and Databases.

[KL03] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *Proceedings of Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung*, volume P-26 of *Lecture Notes in Informatics*, pages 227–246. Gesellschaft für Informatik, Feb. 2003.

[Kra02] Jens-Christian Kramer. Erzeugung garantiert gültiger Server-Seiten für Dokumente der Extensible Markup Language XML. Master's thesis, Institut für Informationssysteme, Universität zu Lübeck, 2002. (in German).

[LK02] Volker Linnemann and Martin Kempa. Sprachen und Werkzeuge zur Generierung von HTML- und XML-Dokumenten. *Informatik Spektrum, Springer-Verlag Heidelberg*, 25(5):349–358, 2002. (in German).

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

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal of Computing*, 19(3):424–437, June 1990.

[SKL04] Henrike Schuhart, Martin Kempa, and Volker Linnemann. $\text{XOBE}_{\text{DBPL}}$: A statically typed XML Database Programming Language Based on JAVA and XQUERY, 2004.

[Spi04] Torben Spiegler. Übungsdatenverwaltungssystem mit XOBE. Master's thesis, Institut für Informationssysteme, Universität zu Lübeck, 2004. (in German).

[Sun01] 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`, 2001.

[Thi] Peter Thiemann. An Embedded Domain-Specific Language for Type-Safe Server-Side Web-Scripting. Univ. of Freiburg 2003, ACM Trans. on Internet Technology, to appear.

[TWP00] Kevin Tao, Wanjun Wang, and Dr. Jens Palsberg. Java Tree Builder JTB. `http://www.cs.purdue.edu/jtb/`, 15. May 2000. Version 1.2.2.

[W3C] W3Consortium. `http://www.w3.org`.

[Web02] WebGain. Java Compiler Compiler (JavaCC) – The Java Parser Generator. `http://www.webgain.com/products/java_cc/`, 2002. Version 2.1.