Type Checking in XOBE

Martin Kempa, Volker Linnemann
Universität zu Lübeck
Institut für Informationssysteme
Osterweide 8
D-23562 Lübeck, Germany
email: {kempa|linnemann}@ifis.uni-luebeck.de

Abstract: XML is the upcoming standard for internet data. Java is the most important programming language for internet applications. Nevertheless, in today's languages and tools there is no smooth integration of Java and XML. The **X**ML **OBj**ects project (XOBE) at the University of Lübeck addresses this mismatch by defining XML objects by XML schemas and by making them to first-class data values. In XOBE, the distinction between XML documents and XML objects no longer exists. Instead, a running XOBE program works only with XML objects. XML documents in text form with explicit tags exist only for communicating with the outside world. This approach allows to check the validity of all XML objects within a program statically at compile time. This is accomplished by XML constructors. Previously generated XML objects are inserted in these constructors such that the validity can be checked at compile time. This paper concentrates on the type checking algorithm in XOBE which is used, among others, for checking the correctness of assignment statements involving XML objects. The type checking algorithm assures that all XML objects that can occur dynamically on the right hand side of an assignment statement are objects that can be assigned to the variable on the left hand side. This type checking is done statically without running the program. The algorithm is based upon regular hedge grammars and regular hedge expressions.

1 Introduction

In the last years the Extensible Markup Language (XML) [W3C98b] has become the standard data format of the Internet. Many different XML-based markup languages have been developed, the usages of which range from publishing documents on web sites to the exchange of data. Committees like the World Wide Web Consortium (W3C) enforce the process of developing XML-related standards, like XPointer, XPath, XSLT, XQuery and software vendors introduce XML-based tools or extend their current products to be accessible via XML. Recently the concept of web services realizing software components over the Internet instead of stand alone web applications became popular.

Today, the implementation of most web applications and web services is realized by standard programming languages like Java [AG98] or Visual Basic. Modern web applications and web services generate XML structures intensively. The content of these dynamic

structures is assembled at run time, in contrast to static web pages, which do not change at run time. For the creation of dynamically generated structures technologies like CGI [Gai95], Java Servlets [Wil99], Java Server Pages [PLC99, FK00] or JAXB [Sun01] are used.

Using these technologies guarantees the correctness of dynamically generated structures only to a very limited extend. XML structures should not only be well-formed, but should also be valid according to an underlying DTD [ABS00] or XML schema [W3C01b], which we call schema in the following, defining the used markup language. Instead, the validity must be "proven" dynamically by appropriate test runs. Moreover, some techniques differentiate between XML strings and XML objects requiring to switch between these two notions by methods called marshalling and unmarshalling. This leads to a serious mismatch between objects in an object oriented programming language and XML structures.

The **X**ML **OBjE**cts project (XOBE) [LK02] overcomes the differences between XML structures as strings and corresponding objects by an extension of the object oriented programming language Java. In other words, when using XML syntax in XOBE, it always denotes XML objects, i.e. generating and analyzing XML is done conceptually only on the basis of objects. Therefore XOBE introduces a class for every element type of a used schema, but does not generate these classes explicitly as Java classes. Instead of that they can be used like built-in data types. They are defined in such a way that the generation of XML objects is done in a syntax oriented manner allowing to check most portions of the property validity, which we call static validity, of all generated XML structures, i.e. XML objects, at compile time.

The present paper focuses on the XOBE type system. The main problem of the XOBE type system is in deciding if an XML object is allowed at the position it appears. This decision problem for subtyping is algorithmically difficult. We introduce an algorithm which checks the subtype relationship of XML objects. The algorithm can be viewed as an improvement of Antimirov's algorithms for the decision problem of regular expressions. Even on XOBE applications that involve quite large types, such as the complete schema of XHTML, our algorithm completes in reasonable time.

Our algorithm enables XOBE to guarantee static validity at compile time which implies the following advantages:

1. XOBE programs are more efficient because we avoid expensive run time checks to guarantee validity.
2. XOBE programs are more reliable because we can omit the programming of recovery procedures which are needed if run time checks fail.
3. XOBE enables a faster development of implementations, because we can avoid extensive test runs, which are necessary to make the validity of the generated XML documents feasible.
4. XOBE improves the maintenance of web services and web applications because it leads to a simpler source code structure.

The paper is organized as follows. In the next section, we give an introduction to the programming with XML objects. In Section 3, we describe the connection between XML objects and regular hedge expressions and introduce subtyping. In Section 4, we present our subtyping algorithm, the main part of the paper. Section 5 explains our implementation techniques and Section 6 discusses some performance measurements. We survey related work in Section 7, i.e. we summarize the state of the art of programming web applications and web services. Section 8 concludes the paper and gives an outlook on future work.

2 XML Objects

In this section we introduce the syntax and semantics of XML Objects (XOBE) briefly. A more detailed introduction can be found in [KL02]. XOBE extends the object-oriented programming language Java by a compile time validation mechanism for dynamically generated XML structures.

Constructing XML Objects

XML structures, which are trees corresponding to a given schema, are represented by *XML objects* in XOBE. Therefore XML objects are first-class data values that may be passed and stored like any other data values. The given schema is used to type different XML objects.

XOBE programs import schema definitions, declared by an import statement `ximport`, to use *XML objects* in the source code. The schema SIF is imported with "`ximport SIF.xsd;`" for example. The classes of these objects are not generated explicitly as Java source code. In fact the element declarations and type definitions in the imported schema define the available classes directly. Every element declaration and every type definition in the schema corresponds to an implicit XML object class. In other words, this means that XML objects are instances of an element declaration or a type definition of an imported schema. Therefore XML objects are XML structures having the corresponding element or elements as the root node.

New XML objects can be created in XOBE programs by expressions which we call *XML constructors*. These expressions are denoted in valid XML syntax where other XML objects can be inserted in places which are allowed according to the imported schema. These values are separated from the surrounding XML syntax by braces.

With the concept of XML objects there is no distinction between the string representation and the object representation of XML structures in XOBE. Thus XML structures in a program always denote objects. Using only XML objects guarantees the property well-formedness as well as static validity for the dynamically generated XML objects at compile time.

We will explain our concepts by the following running example. A web service implementing a shopping application communicates with the outside world using an XML data format. The data format called Shop Interchange Format (SIF) is given in Appendix A.

Our examples focus on the class `Cart` depicted in Figure 1. The class has a member field

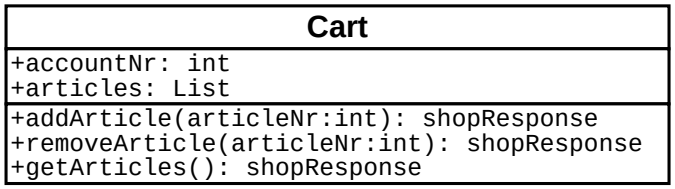

Figure 1: Class Cart

`accountNr` of type `int` which identifies the customer to which this cart belongs and a member field `articles` of class `List` saving already selected items. Further, the class has the three methods `addArticle`, `removeArticle`, and `getArticles`. They make it possible to add and remove certain articles and to display already selected articles.

The following XOBE method generates an SIF `shopResponse`-object containing the account number being taken from the member field `accountNr`.

```
shopResponse removeArticle(int articleNr){
   request done;
   shopResponse response;
   if (this.articles.remove(articleNr))
      done = <request>processed</request>;
   else
      done = <request>fail</request>;
   response = <shopResponse><shoppingCart>
                 <account>{this.accountNr}</account>
                  {done}
              </shoppingCart></shopResponse>;
   return response;
} // removeArticle
```

Listing 1: Method `removeArticle`

We allow `int`-values in places where `String`-values are expected according to the underlying schema. The `int`-value is converted automatically to its decimal notation as a `String`-value by calling method `toString`. It should be obvious that method `removeArticle` is guaranteed statically to generate valid SIF.

A XOBE program generates XML only by XML constructors, i.e. there is no string representation during generation. String generation is necessary only when the document is communicated to the outside world, for example as the result of a Java servlet. Only for this purpose a method `toString` is provided for XML objects.

Although XOBE can ensure most predicates of the property validity statically, runtime checks are necessary in some exceptions. Similar to an array of constant length, where the index has to be in the declared range, the number of occurrences of a specific element has to be between the values set by the attributes `minOccurs` and `maxOccurs` in the defining schema. Additional runtime checks are necessary for identity constraints, restricted string types and facets on numeric types.

Accessing XML Objects

XOBE incorporates XPath [W3C99] for accessing XML objects in XOBE programs. XPath provides a mechanism to express path expressions extracting some nodes in an XML structure. A path expression in general consists of a *location path*, selecting subnodes of a given *context node*. An additional *node test* restricts the selected nodes to specified element names. Further restrictions can be performed with optional *predicates*. In XOBE, the context node is denoted by an XML object variable. XPath expressions in XOBE return a list of nodes, which is regarded as an XML object.

Listing 2 shows the implementation of method `processRequest` processing an incoming shop request. The request is passed to the addressed cart, which returns the resulting response. Note that a global variable `allCarts` is used which gives access to all registered `Cart`-objects.

```
1  shopResponse processRequest(shopRequest rq) {
2     Cart c;
3     shopRequest.shoppingCart sc;
4     sc = rq/shopRequest/shoppingCart[1];
5     c = allCarts.get(sc/account[1]);
6     if (sc/add.getLength() == 1) return c.addArticle(sc/add[1]);
7     else if (sc/remove.getLength() == 1) return c.removeArticle(sc/remove[1]);
8     else if (sc/get.getLength() == 1) return c.getArticles();
9  } // processRequest
```

Listing 2: Method `processRequest`

From the incoming request `rq` the method extracts the shopping cart and assigns it to variable `sc`. Afterwards the targeted cart is chosen from the set of available carts `allCarts`. A nested conditional statement differentiates the three possible requests and calls the suitable methods. The required parameters are gathered from the request.

In the method `processRequest` we use XPath constructs for accessing XML object content. Using the path notations (4-8) it is possible to navigate through XML objects. Because the XPath node test returns a list of resulting XML objects, we have to access the first item of the sequence with the XPath integer predicate (4-7). We determine the size of such a list using method `getLength` (6-8).

The impact of our approach on the host language Java is an XML consistent extension of the type system. Because of the outstanding role of XML in the web application and web services programming world, and maybe the whole software development world in the future, this seems to be a consequent step. We believe that the trade-offs between the extension of an existing programming language on one hand and the approach of defining a new programming language around XML on the other are minimal. Instead the benefits of using already developed code are significant.

3 Basic Definitions

As seen in the last section XOBE allows XML syntax in expressions, assignments and method parameters. During compilation the XOBE system verifies the correctness of the

assignment in two steps. First it determines the types of the right and left hand sides using type inference. Secondly, the subtype relationship of the inferred types is checked by a subtyping algorithm.

In XOBE 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.

Definition 1 (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 $T \cup N$.

The rules in the production set P have to fulfill the following two constraints: [1]

1. Recursive nonterminals may appear in tail positions only.
2. Recursive nonterminals must be preceded by at least one non-nullable expression.

□

A non-nullable expression is a regular hedge expression which does not contain the empty hedge.

We define the regular hedge expressions, referred to in short as regular expressions, similar to the notation used in [W3C01a].

Definition 2 (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:

$$
\begin{array}{rll}
\emptyset & \in Reg & \text{the empty set,} \\
\epsilon & \in Reg & \text{the empty hedge,} \\
b & \in Reg & \text{the simple types,} \\
n & \in Reg & \text{the complex types,} \\
e[r] & \in Reg & \text{the elements,} \\
r|s & \in Reg & \text{the regular union operation,} \\
r, s & \in Reg & \text{the concatenation operation, and} \\
r* & \in Reg & \text{the Kleene star operation.}
\end{array}
$$

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

As an example we formalize the schema SIF (Appendix A) introduced in the last section as a regular hedge grammar as $G = (B \cup E, N, (shopRequest|shopResponse), P)$ with:

$$B = \{\text{integer}, \text{string}, \text{t\_request}\},$$
$$E = \{\text{shopRequest}, \text{shopResponse}, \text{shoppingCart}, \text{account}, \text{add},$$

[1] The two constraints ensure regularity.

$$
\begin{aligned}
&\text{remove}, \text{get}, \text{request}, \text{items}, \text{article}, \text{description}\},\\
N = \{&t\_shopRequest, t\_cartRequest, t\_shopResponse, t\_cartResponse,\\
&t\_items, shopRequest, shopResponse, t\_shopRequest.shoppingCart,\\
&t\_shopResponse.shoppingCart, account, add, remove, get, request,\\
&items, article, description\}, \text{ and}\\
P = \{\quad & t\_shopRequest \rightarrow t\_shopRequest.shoppingCart;\\
& t\_cartRequest \rightarrow (account, (add|remove|get));\\
& t\_shopResponse \rightarrow t\_shopResponse.shoppingCart;\\
& t\_cartResponse \rightarrow (account, request, (items|\epsilon));\\
& t\_items \rightarrow (article*, (description|\epsilon));\\
& shopRequest \rightarrow \text{shopRequest}[t\_shopRequest];\\
& t\_shopRequest.shoppingCart \rightarrow \text{shoppingCart}[t\_cartRequest];\\
& account \rightarrow \text{account}[\text{integer}];\\
& add \rightarrow \text{add}[\text{integer}];\\
& remove \rightarrow \text{remove}[\text{integer}];\\
& get \rightarrow \text{get}[\epsilon];\\
& shopResponse \rightarrow \text{shopResponse}[t\_shopResponse];\\
& t\_shopResponse.shoppingCart \rightarrow \text{shoppingCart}[t\_cartResponse];\\
& request \rightarrow \text{request}[\text{t\_request}];\\
& items \rightarrow \text{items}[t\_items];\\
& article \rightarrow \text{article}[\text{integer}];\\
& description \rightarrow \text{description}[\text{string}]\}.
\end{aligned}
$$
[23]

As in XML Schema we do not demand that the set of element names E and the set of complex types N have to be disjunct. In this paper we use different fonts to separate element names from complex type names. Mixed content is formalized as a simple type string. We deal with attributes similar to elements, but with a specially marked name, rewriting them with regular hedge constructors. Any-types are rewritten to a regular union of all declared element types as well.

As mentioned above XOBE infers at compile time both types of the right and left hand sides of an assignment. Because all variables have to be declared, the type inference of variables is simple. In our example of Listing 1 a variable `done` is declared of type *request* and a variable `response` of type *shopResponse*. Based on the variable types, the type of the whole XML constructor on the right hand side can be inferred. In our example it is shopResponse[shoppingCart[account[integer], *request*]].

After inferring the types of the left and right hand sides, the XOBE type system checks if the type of the right hand side is a subtype of the type of the left hand side. For this

[2] Because the comma (,) is used as concatenation operation in regular expression, we use the semicolon (;) as separator in sets where regular expressions appear as elements.

[3] Element name `shoppingCart` has two different types which we have to distinguish.

example XOBE has to check the so-called *regular inequality*

$$\text{shopResponse}[\text{shoppingCart}[\text{account}[\text{integer}], request]] \leq shopResponse$$

where $\leq$ stands for the subtype relationship. Note, that the name $shopResponse$ on the right hand side stands for the complex type $shopResponse$. The name shopResponse on the left hand side is an element name.

4 Subtyping Algorithm

Checking the subtype relationship between two regular hedge expressions is the main task in proving type correctness of XOBE programs. For this we adopt the Antimirov algorithm [Ant94] for checking inequalities of regular expressions and extend it to the hedge grammar case. The idea behind Antimirov's algorithm is that for every invalid regular inequality there exists at least one reduced inequality which is *trivially inconsistent*. An inequality is trivially inconsistent if the empty hedge is in the language represented by the regular expression on the left hand side but not in the language represented by the right hand side regular expression.

The algorithm operates as follows: It takes the regular inequality to prove as argument and retrieves the leading simple type names and element names from the left hand side regular expression using operation `leadingNames`. The operation leading names is defined as follows.

Definition 3 (leading names)
Given a regular expression $r \in Reg$ the operation $leadingNames$ returns the set of all leading terminal symbols:

$$\begin{aligned}
leadingNames(\emptyset) &= \{\} \\
leadingNames(\epsilon) &= \{\} \\
leadingNames(e[r]) &= \{e\} \\
leadingNames(b) &= \{b\} \\
leadingNames(n) &= leadingNames(r) \text{ with } n \rightarrow r \in P \\
leadingNames(r|s) &= leadingNames(r) \cup leadingNames(s) \\
leadingNames(r,s) &= \begin{cases} leadingNames(r) \cup leadingNames(s) & \text{if } isNullable(r) \\ leadingNames(r) & \text{if } \neg isNullable(r) \end{cases} \\
leadingNames(r*) &= leadingNames(r)
\end{aligned}$$

with $b \in B, n \in N, e \in E, r, s \in Reg$. □

It is defined recursively and returns the leading simple type names and element names. If a complex type name occurs the operation uses the production definition. For a regular concatenation the operation needs the $isNullable$ predicate to check the empty hedge inclusion.

For each determined name the algorithm tries to reduce both sides of the inequality by this name. The resulting reduced inequalities are simpler than the starting inequality in the majority of cases and can be checked by a recursive application of the algorithm. The algorithm tracks already treated inequalities in a set of inequalities, which is empty in the beginning. This ensures termination if we encounter the same inequality later on. We do not prove subtyping directly, instead we calculate the set of all possible reduced inequalities of the given inequality. If we receive a trivially inconsistent inequality we conclude that the given inequality is incorrect. In all other cases we assume that the given inequality holds. This is a standard proceeding in subtyping algorithms of recursive types.

There are two different results of our recursive algorithm. First the algorithm responds false if the inequality in question is trivially inconsistent using the `isNullable` predicate. Secondly, the algorithm terminates with true when it processes an inequality which is in the set of already processed inequalities. This means that our algorithm cannot produce any new inequality in this branch of recursion. If there are still inequalities to derive, the algorithm continues with the operation `partialDerivatives` which is explained after the algorithm definition.

Definition 4 (subtyping algorithm)
Given a set A of already processed inequalities the algorithm to prove $r \leq s$ is defined by the following pseudo code:

```
bool prove(r ≤ s, A) {
  if ((isNullable(r) && ~isNullable(s)) || s == ∅) return false;
  elsif ((r ≤ s) ∈ A) return true;
  else {
    ok := true; pd := ∅; ns := leadingNames(r);
    forall (n ∈ ns) pd := pd ∪ partialDerivatives(n,r ≤ s);
    A := A ∪ {r ≤ s};
    forall ((r1 ≤ s1) ∨ (r2 ≤ s2) ∈ pd)
      ok := ok && (prove(r1 ≤ s1,A) || prove(r2 ≤ s2,A));
    return ok;
  } // else
} // prove
```

Listing 3: Subtyping Algorithm

□

Antimirov introduces so-called *partial derivatives of regular expressions* to express reduced regular expressions. A partial derivative reduces a regular expression by a given type name or element name. In the hedge grammar setting we modify partial derivatives concerning type names and element names. For example, if we have the regular expression $(\text{account}[\text{integer}], \text{request}[\text{t\_request}])$ and calculate its partial derivatives with respect to the given element name account, we receive the result $\{(\text{integer}; \text{request}[\text{t\_request}])\}$. The result is a set in general, because we can get multiple derivatives for a given regular expression. The elements of the set are pairs corresponding to the two dimensions in a regular hedge, the parent-child dimension and the sibling dimension. In the example the first component of the pair is the type of the content of element account. The second is the regular expression reduced by element account.

Additionally Antimirov introduces so-called *partial derivatives of regular inequalities* to express reduced inequalities. In the hedge grammar case these become more complicated. Because the partial derivatives of regular expressions are pairs we have to perform a set-

theoretic theorem observed by Hosoya, Vouillon and Pierce [HVP00].

Theorem 5 (subset relation on Cartesian product)
Given some sets $a, b, c_1, \ldots, c_n, d_1, \ldots, d_n$ the following holds:

$$a \times b \subseteq (c_1 \times d_1) \cup \cdots \cup (c_n \times d_n)$$
$$\Leftrightarrow$$
$$(a \subseteq \cup^{i \in I_1} c_i \vee b \subseteq \cup^{i \in \overline{I}_1} d_i) \wedge \cdots \wedge (a \subseteq \cup^{i \in I_{2^n}} c_i \vee b \subseteq \cup^{i \in \overline{I}_{2^n}} d_i)$$
$$\text{with } \mathcal{P}(\{1,\ldots,n\}) = \{I_1, \ldots, I_{2^n}\} \text{ and } \overline{I}_i = \{1,\ldots,n\} \backslash I_i \,.$$

As we can see, this theorem reduces the subset relation on Cartesian products to a subset relation on sets.

Because types can be interpreted as sets of values, we can write the following relation emerging after reducing the left and right hand sides of an inequality by the leading name n

$$(c_r \times r_r) \subseteq (c_s^1 \times r_s^1) \cup \cdots \cup (c_s^k \times r_s^k) \text{ with}$$
$$partialDerivatives(n,s) = \{(c_s^1; r_s^1), \ldots, (c_s^k; r_s^k)\}$$

for every $(c_r; r_r) \in partialDerivatives(n,r)$. To this relation we can apply the given theorem and receive

$$(c_r \subseteq \cup^{i \in I_1} c_s^i \vee r_r \subseteq \cup^{i \in \overline{I}_1} r_s^i) \wedge \cdots \wedge (c_r \subseteq \cup^{i \in I_{2^k}} c_s^i \vee r_r \subseteq \cup^{i \in \overline{I}_{2^k}} r_s^i)$$
$$\text{with } \mathcal{P}(\{1,\ldots,k\}) = \{I_1, \ldots, I_{2^k}\} \text{ and } \overline{I}_i = \{1,\ldots,k\} \backslash I_i \text{ and}$$
$$partialDerivatives(n,s) = \{(c_s^1; r_s^1), \ldots, (c_s^k; r_s^k)\}.$$

In the definition of the partial derivatives of regular inequalities we collect the disjunctions separately in a set. Additionally we rewrite the relation using the regular expression operators regular union $|$ instead of union on sets $\cup$ and inequality $\leq$ instead of subset on sets $\subseteq$.

Definition 6 (partial derivatives of regular inequality)
Given a regular inequality $r \leq s$ with $r, s \in Reg$ and a terminal symbol $n \in T$ the partial derivatives of that inequality is defined as:

$$partialDerivatives(n, r \leq s) = \{(c_r \leq \bigmid_{i \in I} c_s^i) \vee (r_r \leq \bigmid_{i \in \overline{I}} r_s^i) \mid$$
$$(c_r; r_r) \in partialDerivatives(n,r) \wedge$$
$$partialDerivatives(n,s) = \{(c_s^1; r_s^1), \ldots, (c_s^k; r_s^k)\},$$
$$I \in \mathcal{P}(\{1,\ldots,k\}) \text{ and } \overline{I} = \{1,\ldots,k\} \backslash I\}$$

□

A partial derivative of a regular inequality is a set the elements of which have the form of two inequalities connected with the boolean operation or. This means for our algorithm, that we can apply our procedure `prove` recursively.

In the remaining section we apply our subtyping algorithm to a small example. Consider the example where we have the two types $r \equiv (description, (\text{account}[\text{integer}], description)*)$ and $s \equiv ((\text{description}[\text{string}], \text{account}[\text{integer}])*, description)$ which we denote with r and s for a concise description. We want to check the regular inequality $r \leq s$ in the following, for which we start the subtyping algorithm with an empty set of inequalities $A = \{\}$.

In the first recursion the algorithm calculates the set of leading names $ns = \{\text{description}\}$. Additionally the set A of already processed inequalities is enlarged to $A := A \cup \{r \leq s\}$. We get

$$partialDerivatives(description, r) = \{(\text{string}; (\text{account}[\text{integer}], description)*)\}$$
$$partialDerivatives(description, s) = \{(\text{string}; \text{account}[\text{integer}], s); (\text{string}; \epsilon)\}$$

and Definition 6 leads to

$$\begin{aligned} pd = \{&(\text{string} \leq \text{string} \mid \text{string} \vee && (1)\\ &\quad (\text{account}[\text{integer}], description)* \leq \emptyset); && (2)\\ &(\text{string} \leq \text{string} \vee && (3)\\ &\quad (\text{account}[\text{integer}], description)* \leq \epsilon); && (4)\\ &(\text{string} \leq \text{string} \vee && (5)\\ &\quad (\text{account}[\text{integer}], description)* \leq \text{account}[\text{integer}], s); && (6)\\ &(\text{string} \leq \emptyset \vee && (7)\\ &\quad (\text{account}[\text{integer}], description)* \leq (\text{account}[\text{integer}], s) \mid \epsilon)\}. && (8) \end{aligned}$$

The inequalities 1, 3 and 5 evaluate trivially to true, while inequalities 2, 4, 6 do not hold. Because inequality 7 is false, inequality 8 has to be checked in another recursion of our algorithm.

Let be $r' \equiv (\text{account}[\text{integer}], description)*$ and $s' \equiv (\text{account}[\text{integer}], s) \mid \epsilon)$ for further description. For proving inequality 8 the algorithm calculates the sets of leading names $ns = \{\text{account}\}$. Again the set of inequalities is enlarged to $A := A \cup \{r' \leq s'\}$. The following partial derivatives are generated during the algorithm:

$$\begin{aligned} pd = \{&(\text{integer} \leq \text{integer} \vee && (9)\\ &\quad r \leq \emptyset); && (10)\\ &(\text{integer} \leq \emptyset \vee && (11)\\ &\quad r \leq s)\}. && (12) \end{aligned}$$

It is easy to see, that inequality 9 holds and inequalities 10 and 11 evaluate to false. The last inequality 12 is true, because it is already in our set of already processed inequalities

$r \leq s \in A$. Finally the algorithm accepts $r \leq s$ as correct, because we did not derive inconsistent inequalities in both branches of any disjunction.

To prove the correctness of the algorithm we use the set of all derivable inequalities, which in fact is a family of sets, one set for every recursion branch. It can be shown, that an inequality $r \leq s$ is not valid, if and only if all sets derivable from $r \leq s$ contain false. Secondly it follows that after a finite number of steps each set of inequalities either contains false or saturates. This property ensures termination. Compared to standard subtyping based on regular tree automata which involves the computation of automata intersection and automata complement, our algorithm is more efficient. Although our algorithm has a potential exponential inefficiency as the automata procedure, there are cases where our algorithm is exponentially faster.

5 Implementation Issues

The previous sections presented the representation of types in XOBE. We now describe our XOBE implementation architecture [Kra02], which is shown in Figure 2. Although it

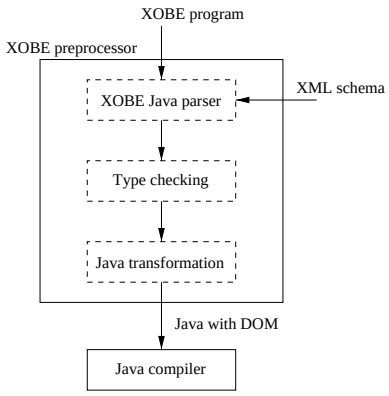

Figure 2: XOBE Java Architecture

is possible to integrate the functionality of XOBE into the Java compiler, we have chosen to implement XOBE as a Java preprocessor. The XOBE preprocessor consists of the following three components:

1. The XOBE parser,
2. the type checking analysis and
3. the transformation to standard Java code.

The XOBE parser reads the XOBE program and converts the XML portions of the program to an internal representation. The parser includes in addition to a standard Java parser

a schema parser and a slightly modified XML parser. The schema parser is necessary to scan the schemas imported by the XOBE program. The XML parser is needed to recognize the XML constructors distributed over the program source code. Because the XML constructors can include XML variables we have to modify the standard XML parser. In our implementation we utilize the Java compiler compiler JavaCC [Web02] to generate the XOBE parser. Additionally we use the XML parser Xerces [The01] to recognize the used schemas. The internal representation of the processed XOBE program is done with the Java tree builder JTB [TWP00].

In the type analysis phase the preprocessor determines whether the parsed program is well-typed or not. Well-typed in XOBE means that the processed XML objects are valid according to the declared schemas. At first the type analysis phase validates the imported schemas. Afterwards, the type check of Java expressions using XML objects, like assignments or method calls, is performed according to the description of the previous section. The type inference of XML constructors and XML variables is followed by subtyping proofs to verify the expressions. Because the type system of standard XML is strict and can be formalized by restricted regular hedge expressions, we can use the regular hedge grammar based algorithm to decide the equivalence of regular expressions for that purpose. XML Schema weakens the strictness by introducing type extension and type restriction, which requires a more sophisticated type inference strategy. The detailed description of this extended algorithm will be introduced in thesis [Kem03].

The last task the preprocessor performs is the transformation of the XOBE program into Java source code, which is accepted by the standard Java compiler. For this resulting Java code several implementation alternatives exist, depending on the XML representation. We chose the standard representation of the Document Object Model, or DOM [W3C98a], recommended by the W3Consortium. The transformation replaces the XML constructors and XPath expressions of the XOBE program with suitable DOM code. The exact transformation rules will be presented in [Kem03]. Please note that even though DOM is an untyped XML implementation not guaranteeing static validity, the transformed XML objects in the XOBE program are valid. This holds because our type checking algorithm guarantees this property. In our implementation the transformation is performed on the internal JTB representation, where we replace the subtrees representing XOBE constructs by newly created subtrees, which represent the suitable DOM code.

6 Experimental Results

In this section we present some preliminary performance measurements of the XOBE implementation. Our interests concerning the performance is twofold. First, we want to know the time which is spent for precompiling XOBE programs and especially for the type checking algorithm. Second, we measure the evaluation time of our resulting DOM-based servlets which we compare to a standard non-DOM servlet implementation. The programs are executed on a Sun Blade 1000 with two Ultra Sparc 3 (600 Mhz) processors running Solaris 8 (SunOS 5.8).

Estate is a small program which generates XHTML web pages for an estate broker. Real estates are stored in an XML file following a small non-standard schema. These files are taken as input and converted into free-standing XHTML documents.

Login is a servlet which realizes the login for an academic exercise administration system. It requests login and password and passes the input to the system.

MobileArchive is a servlet-based web application realizing a WML-connection to a medical media archive. A navigation through the archive structure similar to a file system tree is possible, in addition to a media search. Some media objects of specific formats can be viewed as well.

The first application is written as several simple iterative methods, performing a straightforward traversal of the input tree. The second application communicates over JDBC with an Informix database management system. The client input is passed to the database the respond of which is wrapped into XHTML and sent to the client. The last application is a WML connection to a media archive. The media archive is accessed through a given API. The application memorizes the position of the actual client in the structure of the archive.

We use XHTML (more precise XHTML-transitional) as schema in programs Estate and Login and WML in MobileArchive. The last two applications have been migrated from a standard Java servlet implementation. Many silly mistakes in the non-XOBE implementation have been found doing this task. Despite careful test runs of the previous implementation, many mistakes have been overlooked.

	lines of code		compile time (s)		execution time (s)	
Application	XOBE	schema	total	subtyping	standard	XOBE
Estate	158	1231	2.16	0.05	-	0.9
Login	195	1196	4.61	0.07	0.01	0.01
MobileArchive	1045	355	4.49	0.16	0.03	0.04

In the table the number of lines of the whole XOBE programs and the number of lines of the imported schemas is presented in the first two columns. In the second group of columns the table shows the time spent for precompiling the XOBE program. It includes the parsing, the type inference, the subtyping algorithm and the code transformation into standard Java, as described in Section 5. The time spent during the subtyping algorithm is shown in column “subtyping”. The third column group of the table gives an impression of how the performance of the servlets is affected by the DOM-based implementation. The column “standard” shows the time which has elapsed evaluating the standard non-XOBE servlet implementation. The last column gives the running time of the XOBE program.

As indicated by the table our type checking algorithm runs at acceptable speed for these applications. Even the applications which use quite large types from the XHTML or WML schema are compiled in encouraging time. The execution speed of our DOM-based servlet implementations is slower than the standard servlets as expected but still in a convenient range.

7 Related Work

A number of approaches have been presented facilitating XML processing and generation in existing programming languages. They mainly differ in managing the structure of XML documents. Only a few similar ideas compared to our approach have been proposed.

String processing

The most elementary way to deal with XML documents is to use the string operations which are provided by the programming language, i.e. XML documents are treated as ordinary strings without any structure. The most prominent representative of this technique is given by Java Servlets [Wil99]. In former CGI scripts [Gai95] the programming language Perl [WS92] was used. The technique is rather tedious, when constant XML fragments are being generated. At compile time string operations neither guarantee well-formedness nor static validity.

Java Server Pages [PLC99] are translated by a preprocessor into Java servlets. They allow to switch between XML parts and Java parts for generating XML documents. This switching is done by special markings. Compared to string operations, this technique provides some progress especially when constant XML fragments are being generated. Java Server Pages share with string operations the disadvantage that not even well-formedness is checked at compile time.

Low-level binding

An improvement is to provide classes for nodes of an XML document tree thus allowing to access and manipulate arbitrary XML documents by object-oriented programming. Representatives of this approach, sometimes called low-level binding, are the Document Object Model (DOM) [W3C98a] and JDOM [JDO]. They are widely accepted and supported. It is the only standardized and language independent way for XML processing. Constant XML fragments can be programmed in a pure object-oriented manner, which is rather tedious, or by parsing an XML fragment into the object structure, which requires runtime validation. Low-level bindings ensure well-formedness of dynamically generated documents at compile time, but defer validation until runtime.

JavaScript [Net97] is embedded in HTML and runs on the browser side. It allows, among others, to generate HTML parts dynamically. This can be done either on a pure string basis or on the basis of the DOM.

High-level binding

Recently a series of proposals [Bou02], called high-level bindings, have been presented. With Sun's JAXB, Microsoft's .Net Framework, Exolab's Castor, Delphi's Data Binding Wizard, Oracle's XML Class Generator [Sun01, Mic01, Exo01, Bor01, Ora01] we only mention the better known products. These approaches deal with the assumption that all processed documents follow a given schema. This description is used to map the document structure onto language types or classes, which reproduce directly the semantics intended by the schema. Like the low-level binding, high-level binding provides no facilities to cope with constant XML fragments. Therefore the formulation of constant XML fragments has

to be done by nested constructor or method calls, or by parsing of fixed documents, called unmarshalling. The first procedure is tedious for the programmer the second one needs validation at run-time. High-level bindings ensure well-formedness of dynamically generated documents at compile time. Static validity is only supported to a certain limited extent depending on the selected language mapping. Additionally they have been developed only for specific programming languages and are far away from becoming a standard.

With Validating DOM (VDOM), an extension of DOM, we introduced a high-level binding in previous work [KL01]. We coupled this binding with a mechanism called Parameterized XML (PXML) to support constant XML fragments guaranteeing static validity at compile time. The mechanism we use to guarantee the correctness in PXML is similar to an idea introduced in the setting of program generators about 20 years ago [Lin81]. The basic idea of that work was to introduce a data type for each nonterminal symbol of a context free grammar. So called generating expressions allow the program generator to insert values of these data types in places where the corresponding nonterminal symbol is allowed according to the underlying grammar. This mechanism guarantees the syntactical correctness of all generated programs statically.

Compile time validation

We are aware of only three approaches which are really comparable to XOBE.

The XDuce language [HVP00] is a functional language developed as an XML processing language. It introduces so called regular expression types the values of which are comparable to our XML objects. Elements are created by specific constructors and the content can be accessed through pattern matching. XDuce supports type inference for patterns and variables and performs a subtyping analysis to ensure validity of regular expression type instances at compile time.

BigWig [BMS01] is a programming language for developing interactive web services. It compiles BigWig source code into a combination of standard web technologies, like HTML, CGI, applets, and JavaScript. 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 all templates BigWig validates all dynamically computed documents according to a given DTD. This is done by two data flow analyses constructing a graph which summarizes all possible documents. This graph is analyzed to determine validity of those documents. In comparison to our approach templates can be seen as methods returning XML objects. The arguments of the methods correspond to the gaps of the templates.

Another challenging approach is presented by the language specification of XL [FGK02]. XL is an XML programming language for the implementation of web services. In contrast to our approach, it is a stand-alone programming language, i.e. it is not an extension of an existing language like Java. It provides high-level and declarative constructs adopting XQuery. Additionally imperative language statements are introduced making XL a combination of an imperative and declarative programming language.

Additionally it is worth noticing that the upcoming standard of an XML query language XQuery [W3C02] has to support static validity as well.

Evaluation

In the following table we show how the different approaches give compile time guarantees and facilitate constant XML fragments.

	constant XML fragments	compile time guarantees well-formedness	static validity
CGI, Java Servlet	-	-	-
JSP	+	-	-
DOM, JDOM	-	+	-
JAXB, CASTOR	-	+	+
VDOM	+	+	+
XOBE, XDuce, BigWig, XL	+	+	++

The main observation of the table is that only XDuce, BigWig and XL are really comparable with our proposal. Other approaches either use only a string-based representation of XML structures or make a strict distinction between the string representation and the object representation of an XML structure.

XDuce implements a subtyping algorithm which is based on regular tree automata. It operates on an additional internal representation for regular expression types which is a source of inefficiency. This internal representation is avoided in our algorithm. Furthermore, because XOBE is an extension of Java, it is easier to couple it with other components like database systems.

BigWig's type checking algorithm is based on data flow analyses and is therefore totally different from our algorithm. Compared to XOBE we believe that our type system is more expressive because we can incorporate XML Schema's extension and restriction mechanisms quite naturally into the subtyping algorithm. This seems to be difficult in BigWig.

XL is defined as a special language for web services. In contrast, XOBE is defined as an extension of Java. Java is an already established programming language for web services. Thus XOBE can significantly benefit from Java by using already developed code.

8 Concluding Remarks

This paper presented the type checking algorithm of XOBE, an extension of the programming language Java. XOBE addresses the mismatch between Java on the one side and XML on the other by introducing XML objects which are defined by XML schema. It was shown that in XOBE the distinction between XML documents and XML objects no longer exists. XOBE is defined such that a running program works only with XML objects. XML documents in text form with explicit tagging are needed only for communicating with the outside world. The validity of all XML objects within a program can be checked at compile time by using XML constructors. In XML constructors, previously generated XML objects can be inserted in places which are allowed according to the un-

derlying XML schema. The type checking algorithm which was presented in this paper is used among others for checking the correctness of assignment statements involving XML objects. This correctness is checked by proving that the set of all XML objects which can be derived by evaluating the right hand side of an assignment is contained in the set of XML objects which are allowed as content of the variable on the left hand side according to the underlying XML schema. It was shown that well known notions from the literature can be enhanced for this purpose. In XOBE, the content of XML is accessed by XPath-expressions. A first prototype of the XOBE language extensions including the type system was implemented and some performance measures were provided.

In the future we plan to use XOBE in various application areas. One area will be the media archive software developed in Lübeck [Beh00]. The media archive implementation in its present state primarily uses Java Server Pages for generating HTML and XML. Other application areas will be looked at also. Moreover, we plan to integrate XQuery into the language. Allowing XML objects to be persistent results then in an XML-based database programming language.

Literaturverzeichnis

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

[AG98] Ken Arnold and James Gosling. *The Java Programming Language*. The Java series. Addison Wesley Longman Limited, second edition, 1998.

[Ant94] Valentin Antimirov. Rewriting Regular Inequalities. In Reichel, editor, *Fundamentals of Computation Theory*, volume 965 of *LNCS*, pages 116–125. Springer Verlag Heidelberg, 1994.

[Beh00] Ralf Behrens. MONTANA: Towards a Web-based infrastructure to improve lecture and research in a university environment. In *Proceedings of the 2nd Int. Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2000), Milpitas, California*, pages 58–66. IEEE Computer Society, June 2000.

[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, April 3 2001. Theoretical Computer Science Center.

[BMS01] Claus Brabrand, Anders Moller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001), June 18-19, Snowbird, Utah, USA*. ACM, 2001.

[Bor01] Borland. *XML Application Developer's Guide, JBuilder*. Borland Software Corporation, Scotts Valley, CA, 1997,2001. Version 5.

[Bou02] Ronald Bourret. XML Data Binding Resources. web document, `http://www.rpbourret.com/xml/XMLDataBinding.htm`, 28. July 2002.

[Exo01] ExoLab Group. Castor. ExoLab Group, `http://castor.exolab.org/`, 11 December 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. ACM, 2002. ISBN 1-880672-20-0.

[FK00] Duane K. Fields and Mark A. Kolb. *Web Development with Java Server Pages, A practical guide for designing and building dynamic web services*. Manning Publications Co., 32 Lafayette Place, Greenwich, CT 06830, 2000.

[Gai95] M. Gaither. *Foundations of WWW-Programming with HTML and CGI*. IDG-Books Worldwide Inc., Foster City, California, USA, 1995.

[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.

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

[Kem03] Martin Kempa. *Schema-abhängige Programmierung von XML-basierten Web-Anwendungen*. PhD thesis, Institut für Informationssysteme, Universität zu Lübeck, 2003. to appear, in german.

[KL01] Martin Kempa and Volker Linnemann. V-DOM and P-XML – Towards A Valid Programming Of XML-based Applications. In Akmal B. Chaudhri and Awais Rashid, editors, *OOPSLA '01 Workshop on Objects, XML and Databases, Tamba Bay, Florida, USA*, October 2001.

[KL02] Martin Kempa and Volker Linnemann. On XML Objects. In *Informal Proceedings of the Workshop on Programming Language Technologies for XML (PLAN-X 2002), PLI 2002, Pittsburgh, USA*, pages 44–54, 3.-8. October 2002.

[Kra02] Jens 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.

[Lin81] Volker Linnemann. Context-free Grammars and Derivation Trees in Algol 68. In *Proceedings International Conference on ALGOL68, Amsterdam*, pages 167–182, 1981.

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

[Mic01] Microsoft Corporation. .NET Framework Developer's Guide. web document, `http://msdn.microsoft.com/library/default.asp`, 2001.

[Net97] Netscape Communications Corporation. JavaScript 1.1 Language Specification. `http://www.netscape.com/eng/javascript/index.html`, 1997.

[Ora01] 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.

[PLC99] Eduardo Pelegrí-Llopart and Larry Cable. Java Server Pages Specification, Version 1.1. Java Software, Sun Microsystems, `http://java.sun.com/products/jsp/download.html`, 30. November 1999.

[Sun01] Sun Microsystems, Inc. The Java Architecture for XML Binding, User Guide. `http://www.sun.com`, May 2001.

[The01] The Apache XML Project. Xerces Java Parser. `http://xml.apache.org/xerces-j/index.html`, 15. November 2001. Version 1.4.4.

[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.

[W3C98a] 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.

[W3C98b] W3Consortium. Extensible Markup Language (XML) 1.0. Recommendation, http://www.w3.org/TR/1998/REC-xml-19980210/, 10. February 1998.

[W3C99] W3Consortium. XML Path Language (XPath), Version 1.0. Recommendation, http://www.w3.org/TR/xpath, 16. November 1999.

[W3C01a] W3Consortium. XML Schema: Formal Description. Working Draft, http://www.w3.org/TR/2001/WD-xmlschema-formal-20010925/, 25. September 2001.

[W3C01b] W3Consortium. XML Schema Part 0: Primer. Recommendation, http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/, 2. May 2001.

[W3C02] W3Consortium. XQuery 1.0: An XML Query Language. Working Draft, http://www.w3.org/TR/2002/WD-xquery-20021115/, 15. November 2002.

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

[Wil99] A. R. Williamson. *Java Servlets by Example*. Manning Publications Co., Greenwich, 1999.

[WS92] L. Wall and R. L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastipol, California, 1992.

A Shop Interchange Format

```
<schema>
   <element name="shopRequest" type="t_shopRequest"/>

   <complexType name="t_shopRequest"><sequence>
      <element name="shoppingCart" type="t_cartRequest"/>
   </sequence></complexType>

   <complexType name="t_cartRequest"><sequence>
      <element name="account" type="integer"/>
      <choice>
         <element name="add" type="integer"/>
         <element name="remove" type="integer"/>
         <element name="get"><complexType/></element>
      </choice>
   </sequence></complexType>

   <element name="shopResponse" type="t_shopResponse"/>

   <complexType name="t_shopResponse"><sequence>
      <element name="shoppingCart" type="t_cartResponse"/>
   </sequence></complexType>

   <complexType name="t_cartResponse"><sequence>
      <element name="account" type="integer"/>
      <element name="request" type="t_request"/>
      <element name="items" type="t_items" minOccurs="0" />
   </sequence></complexType>

   <complexType name="t_items"><sequence>
      <element name="article" type="integer" minOccurs="0" maxOccurs="
           unbounded"/>
      <element name="description" type="string" minOccurs="0"/>
   </sequence></complexType>

   <simpleType name="t_request"><restriction base="string">
      <enumeration value="processed"/>
      <enumeration value="fail"/>
   </restriction></simpleType>
</schema>
```