XJ: Facilitating XML Processing in Java$^{\text{TM}}$

Matthew Harren
University of California
Berkeley, CA 94720
matth@cs.berkeley.edu

Mukund Raghavachari
IBM Research
Yorktown Heights, NY 10598
raghavac@us.ibm.com

Oded Shmueli
Technion
Haifa, Israel
oshmu@cs.technion.ac.il

Michael G. Burke
IBM Research
Yorktown Heights, NY 10598
mgburke@us.ibm.com

Rajesh Bordawekar
IBM Research
Yorktown Heights, NY 10598
bordaw@us.ibm.com

Igor Pechtchanski
IBM Research
Yorktown Heights, NY 10598
igorp@us.ibm.com

Vivek Sarkar
IBM Research
Yorktown Heights, NY 10598
vsarkar@us.ibm.com

ABSTRACT

The increased importance of XML as a data representation format has led to several proposals for facilitating the development of applications that operate on XML data. These proposals range from runtime API-based interfaces to XML-based programming languages. The subject of this paper is XJ, a research language that proposes novel mechanisms for the integration of XML as a first-class construct into Java$^{\text{TM}}$. The design goals of XJ distinguish it from past work on integrating XML support into programming languages — specifically, the XJ design adheres to the XML Schema and XPath standards. Moreover, it supports in-place updates of XML data thereby keeping with the imperative nature of Java. We have built a prototype compiler for XJ, and our preliminary experiments demonstrate that the performance of XJ programs can approach that of traditional low-level API-based interfaces, while providing a higher level of abstraction.

Categories and Subject Descriptors: D.3.3[Software]: Programming Languages; I.7.2[Computing Methodologies]: Document and Text Processing

General Terms: Languages, Design

Keywords: XML, Java, Language Design

1. INTRODUCTION

The desire to integrate applications with heterogeneous data models on the Web has driven the development of XML-based standards such as XHTML, XForms, Web Services, etc. The development of applications that process XML data, however, can be tedious and error-prone. Programmers use low-level APIs such as DOM [28] or SAX [22], which provide minimal support for ensuring that programs that process XML are correct with respect to the XML Schemas governing the XML data. Furthermore, the runtime performance of XML processing has been a limitation of XML. Runtime libraries do not take advantage of XML Schema information or static analysis of programs to optimize accesses to XML data.

Copyright is held by the International World Wide Web Conference Committee (IW3C2). Distribution of these papers is limited to classroom use, and personal use by others.
WWW 2005, May 10-14, 2005, Chiba, Japan.
ACM 1-59593-046-9/05/0005.

Given the widespread use of XML, there have been several efforts aimed at facilitating the development of XML processing applications. XML-based languages for processing and transforming data, for example, XSLT [27] and XQuery [29], can simplify the development of certain classes of XML processing applications. The drawback of these languages is that the interface between programming languages used most commonly to develop applications, such as Java, and these XML-based languages is low-level. Runtime libraries, such as XQJ [14], allow Java applications to invoke XQuery or XSLT engines, where the invocation of an XPath or XQuery expression is typically expressed by passing a string to the engine. Since the Java compiler has no built-in support for XML, an error in the string passed to the XQuery engine cannot be caught at compile time, and may not be caught at runtime either (due to the semantics of XPath, a mistyped XPath expression may return no results at runtime, rather than raise a runtime exception). Moreover, since XQuery has no built-in knowledge of the Java data model, there is no clean mechanism by which XQuery programs can utilize the wealth of Java libraries that exists. Finally, since the interaction between Java and XQuery is through runtime libraries, there is no means by which a compiler can optimize accesses to XML data in the context of the Java application that performs these accesses.

Recognizing the need for better support for XML processing in languages such as Java and C♯, researchers have studied augmenting these languages with native XML support — for example, XTATIC [10], Cω [4], and other languages [16, 17]. A drawback of these languages is that they define their own type system for XML data. To develop applications that interoperate with standards such as XML Schema, a programmer is forced to understand the mapping (which is typically not isomorphic) between the internal type system and XML Schema. A key advantage of XML — that standards such as XML Schema allow producers and consumers to agree on the structure of data interchanged — is somewhat lost. A programmer is given a guarantee by the language that data produced/consumed by programs respect the internal type system's view of the desired XML Schemas, not the XML Schemas themselves. Another drawback of previous approaches is their lack of support for updates of XML data — a feature that is both expected and desired in an imperative language such as Java.

The subject of this paper is XJ, a research language that integrates XML as a first-class construct into Java. What sets XJ apart from previous efforts on integrating XML data into programming languages is its consistency with XML standards such as XML Schema and XPath, and its support for in-place updates of XML data. The focus of this paper is on the design of XJ — how the type system of XML Schema and the expression syntax of XPath can be integrated into Java in a manner that is intuitive to both Java and XML programmers. We have built a prototype compiler and a runtime system for XJ. The current output of the XJ compiler is standard Java code that accesses XML data using DOM. The XJ data model is defined independently of DOM or any other runtime representation of XML, thus allowing easy retargeting of the compiler to different runtime systems. We provide results of experiments that indicate that the added flexibility of XJ over APIs such as DOM comes with minimal overhead in performance. We also discuss optimizations that could further improve the performance of XJ programs.

The contributions of the paper are the following:

1. A description of the XJ language, exploring the design issues involved and rationale for the choices taken.

2. A detailed exploration of the issue of updating XML, including a description of the challenges in integrating the XML Schema type system with Java. Some of the issues that we raise, such as *covariant subtyping*, are applicable to any XML-based programming language, for example, XQuery.

3. An overview of a static analysis framework that can be used both for the static detection of type validation errors and for optimizations.

4. Preliminary results with XJ that demonstrate that the performance of XJ programs approaches that of traditional DOM-based programming.

In Section 2, we introduce XJ through a sample XJ application. In Section 3, we examine XJ's data model and type system. We discuss XJ expressions in Section 4. Given the importance of updates, we discuss assignment and updates separately in Section 5. Section 6 gives an overview of a static analysis framework that can be used to determine type errors statically and optimize XJ programs. In Section 7, we describe the current implementation of the XJ compiler. Section 8 provides preliminary experimental results. In Section 9, we discuss projects related to XJ and the characteristics that distinguish XJ from these efforts. We conclude in Section 10.

```
1  import po.*; // Corresponds to ‘‘po.xsd’’
2  public class Discounter {
3     public void giveDiscount(int discQuantity){
4        purchaseOrder po =
           new purchaseOrder(new java.io.File("po.xml"));
5
6        Sequence<item> bulkPurchases =
               po[| /item[quantity > $discQuantity] |];
7        for (int i = 0; i < bulkPurchases.size(); i++){
8           item current = bulkPurchases.get(i);
9           current[| /USPrice |] *= 0.80; // Deduct 20%
10       }
11       po.serialize(System.out);
12    }
13 }
```

Figure 1: An XJ program that reduces the price of certain items in a purchase order.

2. AN XJ EXAMPLE

We introduce the XJ language with the program listed in Figure 1. The complete schema for this example is given in Appendix A. The language features used by this program are described in detail in Sections 3–5.

The `import` statement (Line 1) processes XML element and type declarations from the specified XML Schema file. Given an import declaration, the compiler first attempts to find packages and types as would any Java compiler. If no package or type is found, the compiler appends the extension "`.xsd`" (in this case "`po.xsd`"), and attempts to discover an XML Schema of this name using the `CLASSPATH`. The compiler treats the declarations in this schema, such as `purchaseOrder` and `item`, as classes in XJ. Line 4 loads an XML document, validates it with respect to `purchaseOrder`, and stores a reference to the root element in `po`.

Line 6 uses XPath notation to navigate the XML tree and selects those `item` nodes for which more than the value of the variable `discQuantity` were ordered; the XPath expression can refer to Java variables such as the formal parameter `discQuantity`. At runtime, the value of `discQuantity` will be substituted during the evaluation of the XPath expression. Suppose the XPath expression were mispelled, for example, the programmer had mistyped `po[| /items[...] |]` (that is, `items` instead of `item`). The compiler would detect this as a static error because in the schema, `purchaseOrder` does not contain any element named `items`. Contrast this with DOM, where if a mistyped XPath expression were to be passed to a runtime XPath library, the engine would silently return no results.

`Sequence<item>` on Line 6 denotes an ordered list of zero or more `item` elements. We use generic types introduced in Java 5.0 for such collections. Line 9 uses XPath notation to update the value of an atomic-typed element. Finally, Line 11 emits the document to an external file.

3. XJ DATA MODEL

We begin by outlining how XML Schema declarations are integrated into the Java type system as *logical XML classes*. We then describe how XML documents are represented as well-typed XML values that are instances of these classes. The key properties underlying the XJ data model are:

- The serialization of an XJ XML value that is an instance of an XML class is valid (using XML Schema validation rules) according to the XML Schema declaration corresponding to the XML class.

- An XML tree that is valid with respect to an XML Schema declaration is converted into an instance of the XML class corresponding to the declaration.

3.1 Logical XML Classes

XJ extends the Java type system to allow programmers to declare variables, methods, and fields using types derived from XML Schema declarations.[1] All the built-in atomic types defined by XML Schema as well as elements and atomic types declared in imported XML Schemas are available to an XJ developer. An element or atomic type declaration in an XML Schema is represented in XJ as a *logical XML class*, which is a subclass of a special XJ class `com.ibm.xj.XMLObject`. These classes are *logical* in the sense that there are no corresponding class files at compile time. They may be used wherever a Java class type is expected. In particular, instances of these classes may be constructed using the `new` operator. Logical XML classes are used for type declarations and for static type checking, but are eventually converted into more appropriate runtime classes during code generation, as will be described in Section 7. At the moment, introspection and reflection are not supported on these classes.

Logical XML classes support a notion of containment — an instance of a logical XML class may contain an ordered sequence of instances of other classes, where the containment relationship structure forms an *ordered tree.* Containment relationships cannot, however, be observed with conventional Java mechanisms such as field access or method calls. The only means of observing containment relationships is through XPath expressions.

The derivation of logical XML classes from XML Schema declarations is straightforward, complicated only by XML Schema's classification of declarations into element and type declarations. An element declaration declares an element name and its type. Type declarations declare what values an element may contain. XJ uses element names for generating logical XML class names. We use "e" as the logical XML class name for a declaration of an element e.

We also derive logical classes for atomic types defined or referred to in an XML Schema. Analogous to Java's dichotomy of classes and primitive types, XJ supports the use of logical classes derived from atomic types in addition to those derived from the more structured element declarations. Unlike the logical classes derived from element declarations, for which we describe update semantics in Section 5, instances of the logical classes corresponding to atomic types are immutable. An XML Schema may also contain anonymous atomic type declarations. Since an XML Schema itself is an XML document, which can be viewed as an ordered tree, XJ orders type declarations in a canonical manner and assigns generated names to each such type.

For example, using the declarations in the sample schema in Appendix A, `purchaseOrder` is a logical class derived from an element declaration, and `anon1` and `SKU` are logical classes derived from atomic type declarations. The `anon1` class refers to the anonymous atomic type declaration in the definition of `quantity`. XML namespaces can be used to distinguish XJ logical class names, though, for the most part, we ignore the issue of namespaces in this paper.

The import of an XML Schema behaves similarly to a type-import-on-demand declaration in Java. As with type-import-on-demand declarations, imported types could have names identical to names already used in the compilation unit, or identical to other imported names. The rules for disambiguating identical names when importing multiple schemas are the same as with type-import-on-demand declarations.

Within an XML Schema, element declarations may be global or local in scope. Global element declarations appear at the top level of the schema document. Local element declarations appear within a complex type definition. For the schema given in Appendix A, the declaration of element `purchaseOrder` is global. A local declaration of a `quantity` element occurs in the definition of the complex type `Item`. Suppose the schema is altered so that a local declaration of a `quantity` element, identical to the one in `Item`, also occurs in the definition of `POType`. For a pair of local element declarations of the same name, name disambiguation is required. Local element declarations are treated in a manner analogous to nested classes in Java; such names are disambiguated by qualifying the names with the sequence of names of containing elements, starting from any unambiguous element name, where each name in the sequence is separated by a ".". The code sample below illustrates name disambiguation. Unlike nested classes in Java, however, when the name of local element declaration is unambiguous, the programmer need not qualify names with the enclosing element declarations. In our example schema, the name `productName` is unambiguous and a programmer may use `productName` instead of `purchaseOrder.item.productName` where desired.

```
purchaseOrder po = new purchaseOrder
                      (new File("po.xml"));
purchaseOrder.quantity q1 = po[| /quantity |];
purchaseOrder.item.quantity q2 =
                      po [| /item/quantity |];
```

3.2 Subclassing

All logical XML classes are subclasses of `XMLObject`, which itself is a subclass of `java.lang.Object`. Figure 2 depicts how logical XML classes are integrated into the Java class hierarchy. Each built-in atomic type is a subclass of the `XMLAtomic` class, which serves as the supertype for all atomic types. Atomic types declared or referred to in an imported XML Schema are inserted into the hierarchy as appropriate, as are element declarations.

Subclassing relationships between logical XML classes derived from an XML Schema are implied by subtyping and substitution group declarations in the schema. XML Schema supports a powerful subtyping mechanism, where one type may be declared to be a subtype of another type either by *extension* or *restriction.* A subtype relationship between atomic types imply a subclass relationship between the logical XML classes corresponding to the types. Similarly, if an element e' is in the substitution group of element e, then the logical XML class corresponding to e' is a subclass of the logical XML class corresponding to e.

[1]XJ supports all features of XML Schema, except for identity constraints and redefinition of declarations. Due to conflict with Java syntactic constructs, XJ currently does not support XML element names with "." or "-" in them.

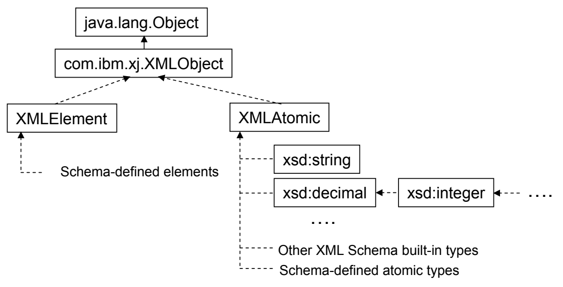

Figure 2: Class hierarchy visible to an XJ programmer. "Schema-defined" refers to element and atomic type declarations in imported schemas. Arrows depict inheritance relationships.

3.3 XJ Types

As mentioned previously, logical XML classes may be used wherever Java classes are allowed, notably to declare types and formal parameters and in the construction of new values. For better static typing of XPath expressions, XJ supports generic types as specified in the Java 5.0 specification [24]. The standard mechanism in XJ for expressing the type of an ordered sequence of XML values is the `com.ibm.xj.Sequence<·>` type. The type of an ordered sequence that contains `items` would be "`Sequence<item>`". Similarly, an atomic type in XML Schema corresponds either to a subtype of `XMLAtomic` or, if it is a simple type, to a `Sequence` of the appropriate subtype of `XMLAtomic`.

XML Schema supports a richer notion of types than Java, based primarily on regular expressions. One alternative considered in the design of XJ was to allow programmers the use of regular expression types in declarations. For example, "`Sequence<item+>`" would refer to a list of one-or-more `items`. Our position is that the declarations needed for regular expression types are too complex, with little added practical value in terms of typing. Our internal typing rules are predicated on the stronger typing system of XML Schema. We plan to use type inference to calculate richer types where possible.

3.4 XJ XML Values

XJ XML values correspond to instances of the logical XML classes defined previously, where these instances are related to each other by containment as appropriate. An XML value in XJ is an *XML item*, where each XML item is either an *atomic value* or a *node* and is an instance of a subclass of `XMLObject`. An atomic value is an instance of an atomic logical class derived from an atomic XML Schema type and stores a value from the set of values denoted by the corresponding atomic type. A node is either an element node or an attribute node. Element nodes are instances of the appropriate logical XML class derived from an element declaration. Each element node may *contain* a sequence of zero or more XML items, and each attribute node may *contain* a sequence of zero or more atomic values. In the XJ data model, attribute nodes cannot occur independently, but are always contained by an element node. XML data that are untyped, that is, not associated with any XML Schema are represented as instances of `XMLElement` or `XMLAtomic`, as appropriate.

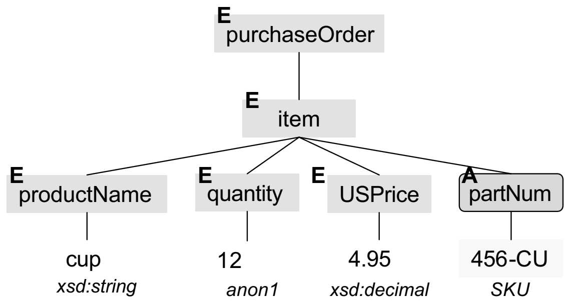

Figure 3: Example of an XJ XML value. E denotes element nodes and A an attribute node. Atomic values are shown with their logical classes in italics.

3.4.1 Well-Typed Values

We restrict the set of XML values to those where the content of a value satisfies appropriate XML Schema constraints. For example, an instance of a logical class corresponding to an XML Schema element declaration must contain a sequence of instances of logical XML classes such that the sequence satisfies the content model of the corresponding XML Schema declaration. Let `toXML()` be a function that converts XJ XML values into canonical XML documents. Given this function, we define well-typed XML values as follows:

- An element node n that is an instance of a logical class e of XML Schema type t is well typed if the content of element e in n`.toXML()` can be validated successfully according to XML Schema rules with respect to the XML Schema type t (ignoring key and keyref constraints) or any declared XML Schema subtype of t.
- An instance of a logical class derived from an atomic type is well typed if it stores a value from the set of values denoted by the corresponding atomic type.
- Any other instance of `XMLObject`, that is, an instance of `XMLAtomic` or `XMLElement` is always well typed.
- An ordered sequence of XML values, $l_1, l_2, ..., l_k$, is well typed if each $l_i, 1 \leq i \leq k$, is well typed.

Updates, construction, and loading of XML values all guarantee that the resulting XML values are well typed.

Every well-typed instance of a logical class is a well-typed instance of any of its superclasses. This property is an outcome of the fact that our subclassing mechanism is based on that of XML Schema, and XML Schema ensures that a XML tree valid according to a subtype of a type is also valid with respect to the type.[2]

4. EXPRESSIONS

The major extension in XJ to Java expressions is the addition of XPath expressions. The semantics of XPath in XJ is consistent with that of the XPath 1.0 specification. Let the execution of an XPath expression on an XML document

[2]Strictly speaking, validating an instance of a subtype with respect to a supertype requires the insertion of `xsi:type` annotations in the XML document. It can be shown that `toXML()` can place these annotations appropriately.

result in a sequence of nodes S in the document. The execution of the same XPath expression on an instance of an XJ XML class corresponding to the root of the XML document results in a sequence of instances of XML classes corresponding to the nodes in the sequence S. In addition to XPath expressions, we discuss the construction of XML data in XJ. Assignment and update are discussed in Section 5.

4.1 XPath Expressions

A programmer may use XPath 1.0 expressions wherever a Java expression is expected and, as we describe in Section 5, on the left-hand side of an assignment as well. The syntax for an XJ XPath expression is

```
expr [| query |]
```

where `expr` is any XJ expression, and *query* is an arbitrary relative XPath location path. For the expression to be valid, the type of `expr` must be τ or `Sequence<`τ`>`, where τ is a logical XML class. At runtime, the content of `expr` defines the XPath context in which the query is evaluated.

The static typing rules for XPath expressions in XJ are described in terms of those defined for XQuery [29]. We briefly sketch a straightforward approach to typing XPath expressions in XJ. Each XJ logical class τ can be mapped to an XQuery item type σ. To determine the type of an XJ XPath expression, the type of `expr` is first converted into an XQuery type:

- If the type of `expr` is a logical class τ, the XQuery type is $\sigma?$, where σ is the XQuery type corresponding to τ.
- If the type of `expr` is `Sequence<`τ`>`, the XQuery type is $\sigma*$, where σ is again the XQuery type for τ.

The static typing rules of XQuery are applied to the XPath expression *query* (after populating XQuery's static environment appropriately), and the result type is converted back into an XJ type:

- If the result type is σ, $\sigma?$, $\sigma*$ or $\sigma+$, where σ is an XQuery item type, then the XJ type is `Sequence<`τ`>`.
- If the result type is (), which is XQuery's empty sequence type, the result is a static type error in XJ. Expressions that always return an empty result are likely the result of programmer error.
- Otherwise, if XQuery returns a complex type such as a sequence type, the XJ type is a conservative approximation of the complex type. For example, if the result type is a *sequence* type σ_1, σ_2, where σ_1 and σ_2 are item types, the XJ type is `Sequence<`τ`>`, where τ is the least common supertype of the XJ types of σ_1 and σ_2 (the least common supertype, τ, may be `XMLObject`).

The straightforward application of the XQuery type system loses information in that complex types returned by XQuery may be converted into `Sequence<XMLObject>`. To obtain more precise typing, we plan to implement type propagation in XJ so that these more complex types are preserved across expressions.

4.2 Construction of XML

XJ introduces two extensions of the `new` operator for constructing XML data. In all cases, if a value cannot be constructed because validation fails, an exception is raised.

First, one can construct XML data by loading an XML document from an `File` using the `new` operator. For example:

```
purchaseOrder p = new purchaseOrder(
                        new File("po.xml"));
```

One may also construct XML values by inlining XML directly, that is "`new` τ`(` *literal XML* `)`", where the literal following the `new` operator is a well-formed block of XML. If τ is a logical class defined in one of the imported schemas, the literal XML block will be checked for validity with respect to that logical class. If τ is `XMLElement`, the XML block is assumed to be untyped and an instance of `XMLElement` is constructed without any validation. As in XQuery, braces can be used within the XML block to delimit XJ expressions that will be evaluated at runtime to provide values for the construction. For example, the following creates a new `item` element (Figure 3 depicts the XJ XML value that is constructed as a result of this XJ code fragment). Braces in this example delimit a reference to the variable `price`, whose value will be substituted for the enclosed expression at runtime:

```
USPrice price = new USPrice(<USPrice>4.95</USPrice>);
item cup_order =
   new item(<item partNum='456-CU'>
                <productName>cup</productName>
                <quantity>12</quantity>
                {price}
            </item>);
```

5. UPDATES

Most of the difficult issues in the integration of XML into Java revolve around the issue of assignment and updates. A central question in defining the semantics of updates in XJ is whether assignments copy values or references to values. The semantics of updates in XQuery update proposals [25] are copy-based. In some sense, copying values has cleaner semantics, since it is easier to guarantee that values are always trees. With reference-based semantics, one must ensure that every value inserted into another does not already have a parent. Otherwise, a node might have more than one parent, and the semantics of XPath expressions (for example, those using the `ancestor` axis) is unclear. On the other hand, since assignment in Java is reference-based, assignment by reference is more intuitive to a Java programmer. Moreover, reference-based semantics simplify the preservation of node identity (since assignment does not change the identity of nodes by copying them). We have chosen to be consistent with Java's reference semantics.

In this section, we describe the syntax and semantics of assignments in XJ. We detail various complications that arise and describe how they are handled in XJ. These complications, which include covariant subtyping, could arise in any XML language based on the XML Schema type system, such as XQuery, but to our knowledge, have not been addressed yet elsewhere.

5.1 Syntax and Semantics

XJ supports three kinds of updates: simple assignments, bulk assignments, and complex updates. The syntax and the semantics for simple assignment to XML types is essentially as that of Java. The type of the left-hand side must be a reference to a logical class that is a supertype of the type of the right-hand side of the assignment. As in Java, when the left-hand side of an assignment is a subtype of the right-hand side, a cast is required. As mentioned previously, XPath expressions can be used in the left-hand side of assignment operations. So, for example, the following updates an `item` element in place:

```
item[| /productName |] = "Widgets";
```

In addition to simple assignment, XJ provides a bulk assignment operation, "`:=`". Bulk updates allow for efficient updates by searching and modifying many portions of an XML value within one traversal. A bulk assignment is valid if the left-hand side is an XPath expression with type `Sequence<`τ`>` and the type of the right-hand side is compatible with the logical class τ. A bulk update operation `expr [| /query |] := y` is semantically equivalent to:

```
τ tmp = y;
for (int i = 0; i < ( expr[| /query |].size();
i++) {
     expr[| /query[$i] |] = tmp;
}
```

For example, the following statement deducts 20% from the price of all items.

```
po[| /item/USPrice |] *:= 0.80;
```

For structural changes, such as inserting a new subtree, we provide methods, such as **`insertAfter`**, **`insertBefore`**, **`insertAsFirst`**, **`insertAsLast`**, etc., that are defined on every logical class. To delete a subtree, a `detach` method is provided. The execution of the `detach` method on an instance of a logical class n removes n from the parent node of n. `detach` has no effect when n has no parent. Each update using one of these commands executes as an atomic operation, and after the execution of the update, the value being updated must remain a tree and remain well typed. The methods provided by XJ for complex structural changes are similar in feature to those proposed for XQuery. The key distinction is that XJ's semantics are in-place updates, whereas XQuery's semantics are copy-based. As an example of a complex update, consider the code sequence:

```
1  purchaseOrder po = ...
2  Sequence<items> purchases = po[| /items |];
3
4  item newitem = ...;
5  item current = purchases.get(0);
6  current.insertAfter(newitem);
```

Line 6 uses `insertAfter` to insert a new `item` after the current (in this case, the first) `item` in the list of `items`.

5.2 Issues with Updates

In this section, we discuss issues related to updates that arose in the design of XJ.

5.2.1 Duplicate Parents and Acyclicity

When constructing or updating an XML value, one must ensure that a value inserted into another value does not already have a parent. Allowing an XML value to be contained within more than one XML value would imply that the XML value is no longer a tree. When XML values are not trees, the dynamic semantics of XPath expressions is unclear, for example, if axes such as `parent` are used.

In XJ, an update that results in an instance of a logical class having more than one parent results in a runtime exception. A programmer may avoid such exceptions by performing `detach` (or alternatively, cloning it) on an XML value before attempting to insert it into another value. In many cases, we can determine statically whether a `detach()` operation is necessary. For example, the results of most XPath expressions must be detached before they can be used as the argument to an insert function.

To ensure that cycles do not arise in XML values, one must verify that the root of an XML value is never inserted into one of its descendants. Again, a runtime check raises an exception if this were to occur.

5.2.2 Complex Type Updates

After an insertion into or deletion from an XML value with a complex type, one must ensure that the value is still well typed; we must verify that the content of the value still satisfies the content model of the type. In general, static type checking of this property is impractical. For example, XML Schema allows one to state that an element a should contain 2 to 7 instances of element b. Statically, it is impossible to determine for an instance of a that is created from a `File` how many b children a would have.

One approach to this issue is to disallow updates where statically it cannot be proven to be safe at runtime. We feel, however, that this solution is too pessimistic. Our solution is to allow updates that cannot be typechecked statically, with the caveat that such updates may cause a dynamic type error. A check is inserted at compile time to ensure the validity of the update. The check is relatively inexpensive in that all that must be verified is that the resulting content of a value satisfies a regular expression. Incremental validation techniques [21] can be used to perform these checks efficiently. In Section 6, we propose a preliminary static analysis framework that can be used to obviate some runtime checks for complex updates.

5.2.3 Covariant Subtyping

XML Schema allows one to declare a subtype of another type by *restriction*, where the values denoted by the subtype form a subset of the values denoted by the supertype. Suppose that in an XML Schema, an element `S` is in the substitution group of an element `E`, where the type of `S` is a subtype of the type of `E` by restriction.

Now consider the following code segment:

```
E x1;
S x2 = new S(<S>...</S>);
...
x1 = x2;
```

Since `S` is a subclass of `E` in XJ, the assignment `x1 = x2` is safe statically. A subsequent update of `x1`, however, may cause a problem. The update may be valid with respect to `E`, but not with respect to `S`. This situation is similar to that of covariant subtyping [18] where an assignment that appears type-safe statically may be unsafe at runtime.

The problem of covariant subtyping already exists in Java for arrays, where at runtime, type errors may occur. Improper use of subtypes by restriction may result in similar type errors at runtime. This issue is not isolated to XJ, but would arise in any language that supports updates on XML Schema types, for example, XQuery.

6. STATIC ANALYSIS

In this section, we present a brief overview of issues related to performing static analyses specific to XML in the context of an imperative language.

6.1 XPath Normalization

The first step in our static analysis is XPath normalization. To normalize XPath expressions, we remove syntactic distinctions that are semantically irrelevant. For example, for a variable `a` of type `purchaseOrder`, the expression `a[| //quantity[parent::item] |]` is equivalent semantically to `a[| //item/quantity |]`. We use a construct called an XDAG, which was introduced in the Xaos system [2]. An XDAG is a directed, acyclic graph where vertices are labeled, and edges between vertices are labeled as either `child` edges or `descendant` edges (all references to backward axes such as `parent` are eliminated). Furthermore, the only vertex with no incoming edge is a distinguished vertex labeled *root*. Having removed all occurrences of backward axes, we perform further normalization by using XML Schema information to rewrite XDAGs. Specifically, we convert occurrences of the `descendant` axis into uses of the `child` axis. For example, `a[| //item/quantity |]` is equivalent to `a[| /item/quantity |]` for the schema of Appendix A.

Given normalized XPath expressions and an XJ program in SSA form [7], we infer a conservative approximation of the structure of the XML data used in the program and relations between variables in the program. This approximation is itself an XDAG, where the vertices are labeled with logical class names. This inference is guided by both schema information and abstract interpretation of the normalized XPath expressions. For example, for an assignment `q = a [|/item/quantity|]`, we infer that `q` will point to the `quantity` descendant of the node pointed to by `a`. Moreover, using schema information, we infer that `q` will have a sibling node labeled `USPrice`.

6.2 Static Type Checking

In many circumstances, an XML Schema provides sufficient information for detecting type errors statically. Suppose `q` and `u` were of type `quantity`. It is clear that an update of the form `q.insertAfter(u)` would be illegal according to the XML Schema of Appendix A; each `quantity` must be followed by a `USPrice`. Schema information, however, may not be sufficient. Assume that in the content model of `item`, `quantity` were replaced by `quantity+`. The update, `q.detach()`, is valid if `q` is not the only `quantity` child of its parent, and invalid otherwise. To detect such errors statically, one must infer more refined information about the state of a variable than can be determined from its type. One might be able to infer that while the content model of `item` specifies `quantity+`, at the program point where the `detach` statement is executed, `q` would be the only `quantity` child of its parent and an error would occur.

An analogous problem arises for assignments to XML types. In XML Schema type definitions, a simple type modifies its base type by applying facets, such as `length`, `enumeration`, and `minInclusive`, to restrict its values. One would like to ensure that updates to instances of such simple types will respect the constraints of their facets and that dynamic casts to simple types are feasible at runtime. It is again necessary to infer information about a variable that is a refinement of the information provided by the variable's type.

In our points-to framework, a reference variable can point to a static instance of an XML document, in which case the points-to information is augmented by an XPath expression. We also augment each such abstract heap location with a *formula*, which is a conservative encoding of the current state of that location, including its type constraints. For most locations that correspond to instances of simple types, the constraints associated with a location can be represented as a regular expression (most facets in XML Schema can be represented in this manner). For locations corresponding to complex types, again, a regular expression represents the current knowledge about the content of the location. For example, for a statement using inline XML construction, one can statically obtain an accurate description of the structure of the value, unless the statement contains a delimited XJ expression that will be evaluated at runtime. This description can be used to check that future updates on the constructed value are valid according to the XML Schema.

6.3 Partial Redundancy Elimination

When multiple XPath expressions are evaluated over a document, and each expression is evaluated independently, there can be significant overhead in redundant traversals of portions of the document. If two XPath expressions `x = p[|/b/c/d|]` and `y = p[|/b/c|]` that share common traversals occur on the same control path, it is possible to compute the XPath expression `y` and use the results to *partially optimize* or *strength reduce* the computation of `x`.

The normalization of XPath expressions can identify many cases where such common subexpressions occur. More instances of redundant traversals can be determined using the XDAG. For example, given `q[|/c/d|]`, where the XDAG shows that `q` points to the `b` child of `p`, we can determine that the traversal of `c/d` from `q` is redundant.

7. IMPLEMENTATION

We have built a prototype compiler for XJ that generates Java source from XJ source programs.[3] The compiler is implemented with Polyglot [19], which provides a framework for parsing and type checking Java source code and implementing extensions to Java. XML Schemas imported by XJ programs are parsed using the XML Schema Infoset Model plugin for Eclipse [8].

The type checking of XJ programs relies on the XAEL engine [9]. The inputs to XAEL are an XPath expression, an XML Schema, and the type of the context node for the XPath expression. XAEL uses abstract evaluation of the XPath expression on the XML Schema to infer the least type such that the result of evaluating the XPath expression on any document conforming to the XML Schema would be an instance of that type. Given this information, our algorithm for type checking XJ expressions and constructors is straightforward. We do not yet perform static analysis to eliminate runtime tests or for other optimizations.

Once an XJ program has passed static type checking, the XJ compiler emits Java code where the syntactic constructs

[3]The XJ compiler and runtime system will be released on Alphaworks (`http://alphaworks.ibm.com`) in early 2005.

introduced by XJ are converted to appropriate calls to the XJ runtime system. For example, all references to logical classes are converted to the appropriate DOM type or `List`.

The compilation of XPath expressions has varying degrees of sophistication. The straightforward compilation of XPath expressions translates the expressions into invocations of the Xalan XPath engine. We have also implemented an optimized code generator, XJ_{direct}, that generates direct DOM traversals for XPath traversals that involve only `child` and `attribute` axes. Finally, we have implemented a version of the compiler, $XJ_{rewrite}$, that uses XAEL to rewrite complex XPath expressions into simple XPath expressions. XAEL uses schema information to convert complex XPath expressions involving `descendant` axis into those using only `child` axis where possible. For example, given the schema of Appendix A, the XPath expression, `purchaseOrder//productName` can be translated into the XPath expressions `purchaseOrder/item/productName`. This rewriting enables us to apply XJ_{direct} to more XPath expressions than we would otherwise.

Currently, separate compilation of XJ classes is not supported since in the generated code, that is, the Java classes, all references to logical XML classes are removed and replaced with runtime system types (for example, a logical XML class corresponding to an element is replaced with `org.w3c.dom.Element`). Separate compilation of another XJ source file with respect to the generated Java class file will not succeed since the signatures of methods and fields in the generated Java class will have the wrong types (the runtime types). We are investigating mechanisms by which information about the logical XML class may be embedded in the Java class files using Java's metadata annotation mechanism so as to support separate compilation.

8. EXPERIMENTS

We provide results of experiments using the prototype XJ compiler and runtime system on the XMark benchmark set [23]. We have rewritten the 20 XQuery benchmarks provided in the XMark set into XJ. The translation is fairly straightforward — `for` loops in XQuery can be translated readily into corresponding `for` loops in Java, and so on.

We compare the performance of the XJ compiler with handwritten DOM code. The DOM code represents code that a normal programmer with knowledge of the XML Schema may write. We compare the performance of the handwritten DOM code with the three versions of the XJ compiler : XJ_{unopt}, where all XPath expressions are evaluated by invoking the Xalan XPath engine; XJ_{direct}, where the compiler generates direct DOM navigations to evaluate simple XPath expressions; and $XJ_{rewrite}$, where XML Schema information is used to rewrite XPath expressions involving the `descendant` axis into those using the `child` axis (where possible). In $XJ_{rewrite}$, after the rewrite step, the compiler again emits direct DOM navigation code to evaluate simple XPath expressions.

For each XMark query, the numbers reported are an average of 100 consecutive runs on a 10MB document generated using the XMark generator. The results for other document sizes are similar. We ran our tests using Xerces version 2.5.0 for XML parsing [1], Xalan version 2.3.1 for runtime XPath processing with caching, and IBM's Java 1.4.1 virtual machine on a 1.6 GHz Pentium 4 with 512 megabytes of RAM. Each test was run repeatedly so that we could obtain performance measurements after the Java virtual machine had warmed up. We excluded parsing times from the numbers provided because all cases use the Xerces parser, and the parsing time was the same in all versions.

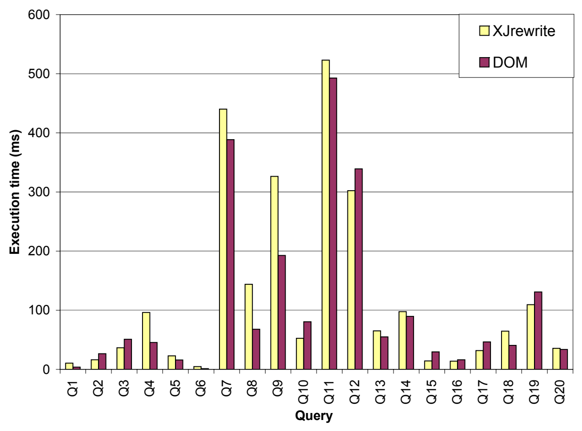

Figure 4: Comparison of performance of DOM handwritten code with $XJ_{rewrite}$. All times in milliseconds on a 10 MB XMark document.

As can be seen from Figure 4, the optimized XJ compiler, $XJ_{rewrite}$, is on average 10% slower than the handwritten DOM version. On certain benchmarks involving the `descendant` axis, XJ does better than the handwritten DOM code. For these benchmarks, the handwritten code uses the DOM `getElementsByTagName` function to search the descendants of a node, while the XJ compiler used schema information to generate more efficient `child` axis DOM navigations automatically. On other benchmarks, where a query has some XPath expressions that cannot be rewritten by the compiler into direct DOM navigations, the XJ generated code performs worse than DOM. In this case, the XJ compiler generates a call to the Xalan XPath engine, which has a much higher overhead than handwritten DOM code.

The cost of invoking Xalan for XPath evaluation is more visible in Figure 5. In this figure, we compare the performance of the three levels of XJ compilation. As expected, the $XJ_{rewrite}$ option outperforms XJ_{direct} for those queries with `descendant` axes that can be rewritten into `child` axes using schema information. For other queries, there is little difference between the two. On some queries, for example Query 7, $XJ_{rewrite}$ does worse than XJ_{direct}. Here the rewriter converts an XPath expression in these queries into a complex XPath expression involving unions, which XJ does not compile into direct DOM navigation. The cost of evaluating this complex XPath expression using Xalan is higher than the cost of evaluating the original XPath expression. The cost of unoptimized XJ code, that is, using Xalan for all XPath evaluations is extremely high. This performance corresponds to what a DOM programmer may obtain if one were to use the XPath API provided with DOM.

9. RELATED WORK

The approaches for integrating XML into programming languages can be divided into three areas: XML-based programming languages, extensions to object-oriented languages, and data-binding approaches.

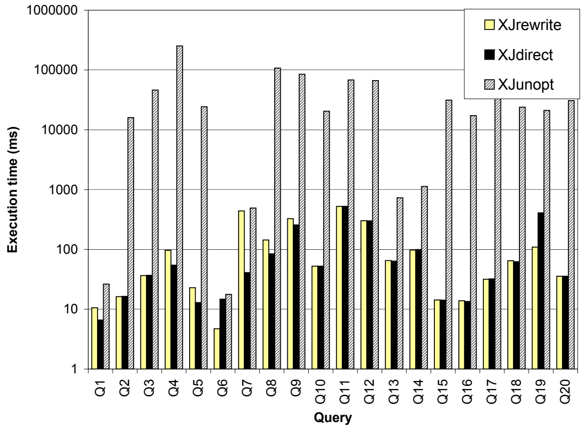

Figure 5: Comparison of XJ$_{rewrite}$, XJ$_{direct}$, and XJ$_{unopt}$ versions of the XJ compiler. All times in milliseconds on a 10 MB XMark document. Note the chart is logarithmic scale.

XML-Based Programming Languages: XQuery [29] is a functional language designed to facilitate processing of XML data. The reader familiar with XQuery will notice similarities between the XJ data model and the XQuery data model. This similarity is intentional. Both data models, in a sense, define XML values as ordered trees where nodes of the tree contain type information and/or labels (in XJ, the label of an instance of class is e and its type is defined by the corresponding schema declaration). Both XJ and XQuery also support subtyping based on XML Schema's subtyping and substitution group mechanisms. It is straightforward to map XJ values into XQuery values and XJ logical classes into XQuery item types. The existence of these mappings is used to ensure that the semantics of XPath expressions in XJ is consistent with that of XQuery. Unlike XJ, XQuery currently does not support updates. The integration of XML into an existing language such as Java raises challenges, especially in the support for updates, in that the abstractions must be intuitive to both XML and Java programmers. Other XML processing languages, such as XSLT [27], have been designed to allow for easy expression of certain patterns of XML processing, but are difficult to use as general-purpose programming languages.

Extensions to OO languages The languages most similar to XJ in design are Cω [4], Xtatic [10], and XOBE [16], each of which integrate XML as a data type into an imperative object-oriented language. Cω integrates XML types deeply into C♯ so that every class may be considered an XML type and vice-versa. This deep integration, however, requires sacrifices to the XML Schema type system — the semantics of XML types in Cω do not match those of XML Schema, but are based on it. Moreover, XML Schema subtyping by name and full XPath navigation are not supported. XJ is more faithful to standards such as XML Schema and XPath by encapsulating XML types as logical XML classes.

Xtatic is an extension of C♯ based on ideas developed in the design of XDuce [12, 13]. It is functional and the data model and the semantics of XML types and values do not correspond exactly to those of standards such as XML Schema. For example, in Xtatic, types correspond to non-deterministic top-down regular tree automata and subtyping is structural, whereas XML Schema types correspond (in some sense) to deterministic top-down regular tree automata and subtyping is defined by name through restrictions and extensions. Navigation of XML values in Xtatic is accomplished by pattern matching, which has different characteristics than those of XPath expressions. XOBE is an extension of Java, which does support XPath expressions, but subtyping is structural. ℂDuce [3], and Scala [20] are functional languages for writing programs that operate on XML. Updates in these languages are generally functional, and not in the imperative style of languages such as Java.

JWIG [6] is a Java extension designed to support web services by dynamically producing well-typed XML (and XHTML), based on a "gap filling" technique. JWIG ensures at compile time that no run-time errors will occur while constructing documents and that constructed documents will conform to their XHTML DTD. JWIG uses Document Structure Description 2.0 as its schema language. JWIG is geared more towards the generation of XML (especially, XHTML) data than full-scale XML-Java integration. A similar "gap filling" approach is exhibited by Xact, a Java library [17] developed in the context of JWIG. Xact provides various operations for creating and filling "named holes" as well as extracting XML fragments. Xact uses static typing to check for DTD output conformance. While Xact is a powerful XML transformation tool, it is not as tightly integrated with Java as XJ is.

Data-Binding Approaches: Frameworks for Java-XML bindings [5, 15] generate Java classes statically from XML Schemas. JAXB [15] covers most of XML Schema and it supports (in theory) evaluation of XPath expressions over the represented objects. An application may modify the in-memory object tree through interfaces generated by the JAXB binding compiler. Due to differences between the Java and XML Schema data models, the generated Java classes do not correspond exactly to the source XML Schema, especially when complex content models are involved. A programmer must understand the mapping rules used by the engine in order to use the generated Java classes as proxies for the XML data. Another drawback is that the programmer is bound to a particular framework — switching to another framework may require drastic changes to applications since the mapping rules may change. In contrast, a programming language such as XJ allows the programmer to develop applications natively in XML — the runtime implementation, which may use a framework such as JAXB, is hidden from the programmer. This allows applications to be more portable since switching to another framework requires only a one-time reengineering of the compiler.

10. CONCLUSIONS

We have designed a new language, XJ, that integrates XML into Java. The distinguishing characteristics of XJ are its support for in-place updates and its consistency with XML standards such as XQuery and XML Schema. We have built a prototype compiler for XJ, structured as a source-to-source translator that uses DOM to access XML data in the compiled code. Our experiments indicate that XJ allows one the flexibility of developing applications using the high-level XPath syntax, while obtaining performance close to that of handwritten DOM code.

11. REFERENCES

[1] Apache Software Foundation. *Xerces2 Java and Xalan Java*. http://xml.apache.org.

[2] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 455–466, 2003.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the Eighth International Conference on Functional Programming*, pages 51–63. ACM Press, 2003.

[4] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. http://research.microsoft.com/~emeijer.

[5] *Castor*. http://castor.exolab.org.

[6] A. S. Christensen, A. Møller, and M. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, November 2003.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Eclipse project. XML schema infoset model. http://www.eclipse.org/xsd/.

[9] A. Fokoué. XAEL: XML abstract evaluation library. Unpublished Manuscript.

[10] V. Gapeyev and B. Pierce. Regular object types. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 151–175, July 2003.

[11] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, V. Sarkar, and R. Bordawekar. XJ: integration of XML processing into Java. In *Proceedings of the 13th International World Wide Web conference on Alternate track papers & posters*, pages 340–341, 2004.

[12] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[13] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the Fifth International Conference on Functional Programming*, pages 11–22, 2000.

[14] Java Community Process. *XQJ: XQuery API for Java*, May 2004. http://jcp.org/aboutJava/communityprocess/edr/jsr225/index.html.

[15] *Java architecture for XML binding*. http://java.sun.com/xml/jaxb/.

[16] M. Kempa and V. Linnemann. Type checking in XOBE. In *Datenbanksysteme fur Business, Technologie und Web*, 2003.

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

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 2000.

[19] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. *LNCS 2622*, pages 138–152, April 2003.

[20] M. Odersky. Programming in Scala. http://lamp.epfl.ch/scala.

[21] M. Raghavachari and O. Shmueli. Efficient schema-based revalidation of XML. In *Proceedings of Extending Database Technology (EDBT)*, volume 2992 of *LNCS*. Springer-Verlag, March 2004.

[22] Simple API for XML. http://www.saxproject.org.

[23] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, pages 974–985, 2002.

[24] Sun Microsystems, Inc. *Java 2 Platform, Standard Edition (J2SE) version 1.5*, 2004. http://java.sun.com/j2se/1.5.0/.

[25] G. Sur, J. Hammer, and J. Siméon. UpdateX - an XQuery-based language for processing updates. In *PLAN-X*, January 2004.

[26] World Wide Web Consortium. *XML Schema, Parts 0,1, and 2*.

[27] World Wide Web Consortium. *XSL Transformations*, November 1999.

[28] World Wide Web Consortium. *Document Object Model Level 2 Core*, November 2000.

[29] World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, August 2003. W3C Working draft.

A. APPENDIX

This schema, derived from that in the XML Schema specification [26], is used for the examples in this paper.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="purchaseOrder" type="POType"/>
 <xsd:complexType name="POType">
  <xsd:sequence>
   <xsd:element name="item" type="Item" minOccurs="0"
    maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="Item">
   <xsd:complexType>
     <xsd:sequence>
      <xsd:element name="productName" type="xsd:string"/>
      <xsd:element name="quantity">
       <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
         <xsd:maxExclusive value="100"/>
        </xsd:restriction>
       </xsd:simpleType>
      </xsd:element>
      <xsd:element name="USPrice"  type="xsd:decimal"/>
     </xsd:sequence>
     <xsd:attribute name="partNum" type="SKU"
                    use="required"/>
 </xsd:complexType>

 <xsd:simpleType name="SKU">
  <xsd:restriction base="xsd:string">
   <xsd:pattern value="\d{3}-[A-Z]{2}"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```