RC23007 (W0311-138) November 18, 2003
Computer Science

IBM Research Report

XJ: Integrating XML Processing into Java™

Matthew Harren[1], Mukund Raghavachari[2], Oded Shmueli[3], Michael Burke[2], Vivek Sarkar[2], Rajesh Bordawekar[2]

[1]University of California
Berkeley, CA

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

[3]Technion - Israel Institute of Technology

XJ: Integration of XML Processing into Java™

Matthew Harren [*]
University of California, Berkeley

Mukund Raghavachari
IBM T.J. Watson Research Center

Oded Shmueli [*]
Technion – Israel Institute of Technology

Michael Burke
IBM T.J. Watson Research Center

Vivek Sarkar
IBM T.J. Watson Research Center

Rajesh Bordawekar
IBM T.J. Watson Research Center

ABSTRACT

The increased importance of XML as a universal data representation format has led to several proposals for enabling 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™. 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, and supports in-place updates of XML data thereby keeping with the imperative nature of Java. We have also built a prototype compiler for XJ, and our preliminary experimental results demonstrate that the performance of XJ programs can approach that of traditional low level API-based interfaces, while providing a higher level of abstraction.

1. INTRODUCTION

XML [32] has emerged as the de facto standard for data interchange. One reason for its popularity is that it defines a standard mechanism for structuring data as ordered, labeled trees. The utility of XML as an application integration mechanism is enhanced when interacting applications agree on the structure and vocabulary of labels of the XML data interchanged. This requirement has led to the development of the XML Schema [28] standard — an XML Schema specifies a set of XML documents whose vocabulary and structure satisfy constraints in the XML Schema.

Despite the increased importance of XML, the available facilities for processing XML in current programming languages are primitive. Programmers often use runtime APIs such as DOM [29], which builds an in-memory tree from an XML document, or SAX [23], where an XML document parser raises events that are handled by an application. None of the benefits associated with high-level programming languages, such as static type checking of operations on XML data are available to a programmer. The responsibility of ensuring that operations on XML data respect the XML Schema associated with it falls entirely on the programmer.

The alternative approach to using standard interfaces to process XML data is to embed support for XML within the programming language. For example, a widely used XML-based standard is XPath [30], a language for navigating and extracting XML data. Support for XPath in the programming language provides a natural, succinct and flexible construct for accessing XML data. Extending current programming languages with awareness of XML, XML Schema, and XPath through a careful integration of the XML Schema type system and XPath expression syntax can simplify programming and enables useful services such as static type checking and compiler optimizations.

The subject of this paper is XJ, a research language that integrates XML as a first-class construct into Java. The design goals of XJ distinguish it from other projects that integrate XML into programming languages. The goal of introducing XML as a type into an object-oriented imperative language is not new — XTATIC [11], XACT [17] and other languages [16, 20, 27] have studied the integration of XML into C♯ and Java. What sets XJ apart from these and other languages is its consistency with XML standards such as XML Schema and XPath, and its support for in-place updates of XML data, thereby keeping with the imperative nature of general-purpose languages like Java.

This paper introduces XJ, explores the issues that arose in its design, and compares its abstractions with those of other languages. 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. We provide experimental results that demonstrate that the added flexibility of XJ over APIs such as DOM come 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 discussion of the semantics of subtyping and in-place updates in XJ.
3. An exploration of the compiler optimizations enabled by a high-level language approach such as that used by XJ.

[*]Work performed at the IBM T.J. Watson Research Center

```
 1 import "po.xsd";
 2 public class Discounter {
 3    public void giveDiscount(){
 4       purchaseOrder po =
             (purchaseOrder)XMLItem.load("po.xml", null);
 5
 6       XML<item*> bulkPurchases =
             `po/item[quantity/text() > 50]`;
 7       for (int i = 0; i < bulkPurchases.size(); i++) {
 8          item current = bulkPurchases.get(i);
 9          `current/USPrice/text()` *= 0.80; // Deduct 20%
10       }
11       XMLItem.serialize(po, "po.xml");
12    }
13 }
```

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

4. Preliminary experimental results with the XJ compiler and runtime system that demonstrate that the performance of XJ programs approaches that of traditional DOM-based programming, with some simple optimizations.

We describe the XJ data model and type system in Section 2. We discuss XJ expressions in Section 3. Section 4 describes our support for mutating XML data. In Section 5, we describe the current implementation of the XJ compiler. Section 6 investigates possible optimizations that can improve the performance of XJ programs substantially. Section 7 contains some preliminary experimental results. In Section 8, we discuss projects related to XJ and the characteristics that distinguish XJ from these efforts, and we conclude in Section 9.

Brief Example We introduce the XJ language with the sample program listed in Figure 1. The complete schema for this example is given in Appendix A. The language features used by this program are describe in detail in Sections 2 and 3. In Section 5, Figure 5 shows the DOM-based Java code generated for this XJ example. The generated code is representative of the programming models currently available for XML processing, and highlights the contrast with the XJ approach.

An `import` statement at the start of the program processes XML element and type declarations from the specified XML Schema file. The compiler treats the declarations in this schema, such as `purchaseOrder` and `item`, as types in XJ. Line 4 loads an XML document, ensures that it is valid 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 50 were ordered. For convenience, the current XJ design uses the backquote, "\`", to delimit XPath expressions. The backquote delimiter helps avoid ambiguity over uses of the "//" token, which represents the start of a comment in Java, but has a special meaning in XPath (it represents a descendant-or-self axis traversal).

`XML<`τ`>`, where τ is generally a regular expression, is a predefined keyword in XJ that denotes an ordered sequence such that the types of the contents of the sequence satisfy τ. In this particular example, `XML<item*>`, denotes an ordered list of zero or more `item` elements. Each such ordered sequence is also an instance of `java.lang.List`, and the methods defined in this interface can be used to traverse the sequence, for example, the `get()` method is used in Line 8 to access contents of the list.

Line 9 uses XPath notation to update the value of a atomic-typed element. Finally, Line 11 invokes a procedure to serialize the document back to an external file.

2. XJ DATA MODEL

In this section, we examine the XJ data model and the issues involved in its design. We begin with a description of how XML Schema declarations are integrated into the Java type system as *logical XML classes*. These logical XML classes form the basis for the XJ data model. We then describe how XML data are represented in XJ and the XJ type system.

The reader familiar with the XQuery 1.0 and XPath 2.0 data model will find several similarities between it and XJ's data model. This similarity is intentional. One can define a function `toXQuery()` that injects XJ XML values into values in the XQuery data model. Due to space limitations, we do not define this function explicitly, but the mapping of XJ constructs to those of XQuery is straightforward. This function is useful in defining the semantics of XPath expressions over the XJ data model as will be described in Section 3.

2.1 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. All the built-in atomic types defined by XML Schema as well as elements, attributes and atomic types declared in imported XML schemas are available to an XJ developer. Element, attribute and atomic type declarations are viewed as *logical XML classes*. These classes are logical in the sense that they are not true Java classes and may not be tangible at runtime. Instances of these logical XML classes are instances of `java.lang.Object`. Logical XML classes are used for type declarations and for static type checking, but are eventually erased during code generation into more appropriate runtime classes, as will be described in Section 5. Furthermore, 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, attribute, and type declarations. An element/attribute declaration declares an element/attribute name and its type. Type declarations declare what values an element or an attribute may contain. XJ uses element-type and attribute-type pairs for generating logical XML class names. We use "$e :: t$" as the logical XML class name for a declaration of an element e with type t. In XML Schema, one can declare an element and an attribute, where both have the same name and type. In order to distinguish elements from attributes, the logical class name derived from attribute declarations are prefixed by a "@", that is, they are of the form $@a :: t$.

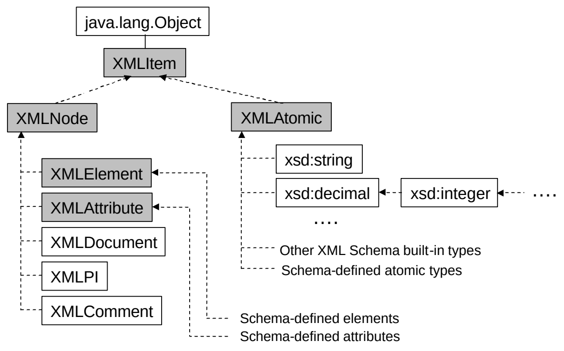

Figure 2: Class hierarchy visible to an XJ programmer. "Schema-defined" refers to element and atomic type declarations in imported schemas. Arrows depict inheritance relationships. Shaded classes denote abstract logical XML classes.

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, it is useful to be able to use logical classes derived from atomic types in addition to those derived from the more structured element and attribute declarations. There are coercions defined between logical classes derived from certain atomic types and Java primitive types, for example, between `xsd:int` and `int`, and between `xsd:boolean` and `boolean`. An XML Schema may 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.

As an example, given the declarations in the sample schema in Appendix A, `purchaseOrder::POType` is a logical class derived from an element declaration, `@partNum::SKU` is a logical class derived from an attribute 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 element, attribute, and type names, though for the most part, we ignore the issue of namespaces in this paper. Where the logical XML class is unambiguous by using the element name, as a shortcut, XJ allows programmers to discard the type portion of logical class names. For example, a programmer may use `purchaseOrder` interchangeably with `purchaseOrder::POType`.

2.2 Subtyping and Substitution Groups

The XJ subtyping relation on logical XML classes is defined independently of the Java subclassing mechanism. The logical XML classes are integrated into the Java class hierarchy as shown in Figure 2, where `XMLItem` is the supertype of all logical classes. The shaded classes denote *abstract* logical XML classes — no direct instances of these classes exist. Logical XML class inheritance is shown with dotted lines to distinguish it from normal Java subclassing. Each built-in atomic type is a subtype of an `XMLAtomic` type, 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 and attribute declarations.

Use of the XML Schema substitution groups and subtyp-

```
<xsd:complexType name = "NewPOType">
  <xsd:complexContent>
    <xsd:extension base = "POType">
      <xsd:sequence>
        <xsd:element name = "NumCustomer"
                     type = "xsd:int"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Figure 3: Example of XML Schema subtyping.

ing mechanisms(that is, restriction or extension of types)[1] in an imported schema results in more logical XML classes than those derived directly from declarations in the XML Schema. Let $e :: t$ be a logical class corresponding to an element declaration in an XML Schema. For each type t' declared to be a subtype of t through the use of one or more restriction and/or extension steps, a logical class $e :: t'$ is available. The class $e :: t'$ is a subtype of $e :: t$ in XJ. The use of the substitution group mechanism results in a set of classes $e' :: t'$ where elements of label e' with type t' are declared to be in the substitution group of $e :: t$. Each such class $e' :: t'$ is also a subtype of $e :: t$ in XJ.

Figure 3 gives an example of the use of XML Schema subtyping, where `POType` declared in Appendix A is extended to contain a new element called `NumCustomer`. The presence of this declaration in the XML Schema of Appendix A would result in a logical XML class `purchaseOrder::NewPOType`, which would be a subtype in XJ of `purchaseOrder::POType`.

2.3 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 *item*, where each item is either an *atomic value* or a *node*. An atomic value is an instance of the 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, an attribute, a comment, a processing instruction, or a document node. Element nodes are instances of the appropriate logical XML class derived from an element declaration. Similarly, attribute nodes are instances of a logical XML class derived from an attribute declaration. Each element node may *contain* a sequence of zero or more items, and each attribute node may *contain* a sequence of zero or more atomic values. Document, comment, and processing instruction nodes are translated into instances of `XMLDocument`, `XMLComment`, and `XMLPI`, respectively. We focus on the elements, attributes, and atomic values and types in this paper. The handling of document, comment, and processing instruction nodes is straightforward.

2.3.1 *Well-Typed Values*

XJ XML values are restricted to only those where the containment hierarchy of a value satisfies relevant XML Schema constraints. More precisely, we define a function `toXML()` that converts XJ XML values into canonical XML docu-

[1] In XML Schema, *restriction* subtyping refers to defining a type whose possible values are a subset of the possible values of the base type, while *extension* subtyping adds new fields to the base type.

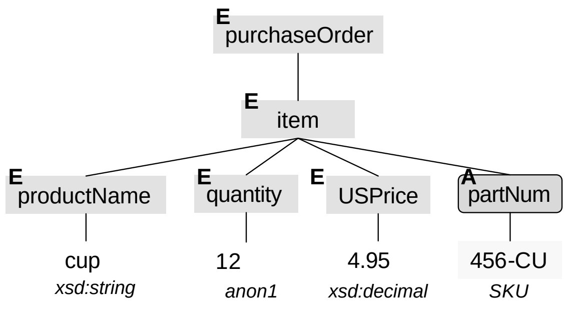

Figure 4: Example of an XJ XML value. E denotes element nodes and A an attribute node. Atomic values are shown with their logical classes in italics.

ments. Given this function, we define well-typed XML values as follows:

- An element node n that is an instance of $e :: t$ is well typed if `toXML(n)` can be validated successfully according to XML Schema rules with respect to the XML Schema type t (ignoring key and keyref constraints).
- Similarly, an attribute node n that is an instance of $@a :: t$ is well typed if `toXML(`n`)` can be validated successfully according to XML Schema rules with respect to the XML Schema type t.
- An instance of a logical class corresponding to an atomic type is well typed if it stores a value from the set of values denoted by the corresponding atomic type.

We shall be interested in ordered sequences of well-typed XML values. 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. We shall use $\mathcal{S}$ to denote the set of all well-typed ordered sequences.

Updates, construction, and loading of XML values all guarantee that the resulting XML values are well typed. Where it is not possible to guarantee statically that values are well typed, dynamic checks will be used.

2.3.2 Updates

Once an XJ XML value is created, its contents may be changed, but the class of a node or atomic value may not be changed (without changing node identity). This invariant helps preserve type safety in the presence of updates, aliasing, and subtyping, as will be discussed later.

2.4 Type System

The logical XML classes corresponding to XML Schema built-ins or element, attribute, and atomic type declarations in imported schemas form the building blocks for the XJ type system. The primary type constructor in XJ is that of a sequence `XML<`τ`>`, where τ is typically a regular expression over logical class names (we use $\mathcal{L}$ to range over logical class names). The set of XJ XML types, $\mathcal{T}$, are all types of the form `XML<`τ`>`, where τ is defined as follows:

$$\tau ::= \tau,\tau \;\Big|\; \tau|\tau \;\Big|\; \&\tau \;\Big|\; (\tau) \;\Big|\; \tau* \;\Big|\; \tau+ \;\Big|\; \tau? \;\Big|\; \mathcal{L}$$

As syntactic sugar, we allow $\mathcal{L}$ to stand for `XML<`$\mathcal{L}$`>`, where such use does not collide with Java class names. The interleaving operator, &, is similar to the `all` construct of XML Schema. Unlike the XML Schema `all` construct, it may be used in a nested fashion.

2.4.1 Semantics

Given the definition of well-typed XML values, the semantics of XJ types is straightforward. The denotation of a sequence type `XML<`τ`>`, $[\![\cdot]\!]:\mathcal{T} \to 2^{\mathcal{S}}$, is a set of ordered sequences of well-typed XJ XML values. It is defined in terms of $[\![\tau]\!]$, whose definition follows. In the definitions, the concatenation of ordered sequences results in a flat ordered sequence. More formally, let $s_1 = u_1, u_2, \ldots, u_i$ and $s_2 = v_1, v_2, \ldots, v_j$ represent ordered sequences. The concatenation of ordered sequences, $s_1 \cdot s_2$, is the ordered sequence $u_1, \ldots, u_i, v_1, \ldots, v_j$. The interleaving of the sequence s_1, $\mathcal{P}(s_1)$, is the set whose members are ordered sequences $s_3 = w_1, \ldots, w_i$ such that $w_1, \ldots, w_i$ is a permutation of $u_1, \ldots, u_i$.

- $[\![\tau, \tau']\!] = \{s_1 \cdot s_2 | s_1 \in [\![\tau]\!] \wedge s_2 \in [\![\tau']\!]\}$
- $[\![\tau|\tau']\!] = \{s | s \in [\![\tau]\!] \vee s \in [\![\tau']\!]\}$
- $[\![\&\tau]\!] = \bigcup_{s \in [\![\tau]\!]} \mathcal{P}(s)$
- $[\![(\tau)]\!] = [\![\tau]\!]$
- $[\![\tau*]\!] = [\![\tau+]\!] \cup \{()\}$
- $[\![\tau+]\!] =$ All finite sequences $s_1 \cdot s_2 \cdot \ldots \cdot s_k$, where $k \geq 1$, such that $s_i \in [\![\tau]\!], 1 \leq i \leq k$.
- $[\![\tau?]\!] = [\![\tau]\!] \cup \{()\}$.
- $[\![\mathcal{L}]\!] = \{l | l$ is a well-typed instance of logical class $\mathcal{L}\}$

In the above definitions, $()$ refers to the empty sequence. If τ is one of `XMLElement`, `XMLAttribute`, `XMLNode`, `XMLAtomic` and `XMLItem`, $[\![\tau]\!]$ is defined to be the union of the denotations of all subtypes of τ. For example, the semantics of `XMLElement` is $\bigcup [\![e :: t]\!]$ where $e :: t$ is an XJ subtype of `XMLElement`.

2.4.2 Examples

The following examples demonstrate the syntax of type declarations in XJ. Assume that we have imported the XML Schema in Appendix A:

- `xsd:int` declares a variable using the built-in schema type `xsd:int`.
- `XML<xsd:int>` is identical to the previous declaration.
- `XML<XMLAttribute?>` declares a sequence that is either empty or contains a single attribute.
- `XML<item::Item>` or `XML<item>` or `item` all declare a sequence that contains a single XJ value, where the root element has label `item` and type annotation `Item`. Since the type of `item` is unambiguous in the schema, one can omit the type declaration "`::Item`".
- `XML<(productName | quantity)*>` refers to a sequence of zero or more `productName` and `quantity` elements.

2.4.3 Subtyping

Given the semantics of XJ types, subtyping can be defined by a simple rule: an XJ type τ is a subtype of another XJ type τ' if the set of values denoted by τ is a subset of the set of values denoted by τ'. While this rule may appear to be similar to the structural subtyping defined in XTATIC, subtyping in XJ and XTATIC are quite disparate. In XTATIC, the language allows arbitrary structural subtyping through extension and restriction, whether or not the relationship is declared by name in a schema (substitution groups, however, are not recognized). Since well-typed XJ XML values are dictated by XML Schema, XJ permits only those subtyping relationships defined by a schema using the substitution group, extension, and restriction mechanisms.

For example, consider the complex type extension shown in Figure 3. The following XML document is a valid instance of `NewPOType`, where the XML Schema subtype feature is used to indicate that the type of this instance of `purchaseOrder` is `NewPOType`.

```
<purchaseOrder xsi:type = "NewPOType">
   ...
   <NumCustomer> 999 </NumCustomer>
</purchaseOrder>
```

The following XJ code makes use of `POType` and its subtype `NewPOType`. Assume that `poNew` is a variable of type `purchaseOrder::NewPOType`:

```
purchaseOrder::POType po = poNew;
```

The statement that assigns the reference value of `poNew` to `po` is legal, as `poNew` yields an instance of the type of `po` according to the subtyping rules of XML Schema.

3. EXPRESSIONS

We now describe how values are manipulated in XML. This section describes construction, querying, and serialization; update statements are described in the next section.[2]

3.1 Queries and Expressions

Programmers may use XPath 2.0 expressions, arithmetic and relational operators to manipulate XML data. The typical mechanism for specifying XPath expressions on XJ variables is

`id/*query*`

where `id` is an identifier in the current context that is declared with an XJ XML type, and *query* is an arbitrary XPath expression. `id`, which may correspond to a single XJ XML value or a sequence of XJ XML values, is used to define the context in which `query` is evaluated. The result of executing an XPath expression is a sequence of XML values, the type of which is determined from the XPath expression and the type of `id`. For example,

```
purchaseOrder p = ...;
// Select the last item.  This returns a list of
// zero or one item elements.
XML<item?> results = `p/item[last()]`;
// This statement will throw an exception if results
// is empty (i.e. p contained no items).
item last = results.get(0);

// Now get all of the other item elements in this
// purchase order.
XML<item*> theRest = `last/preceding-sibling::item`;
```

[2]The issue of error handling is beyond the scope of the paper — appropriate Java exceptions or extensions thereof are thrown which may be caught.

The semantics of XPath expressions in XJ is defined in terms of XPath 2.0. As mentioned previously, there is a function `toXQuery()` that can inject XJ XML values into the XPath 2.0 and XQuery 1.0 data model. Since we are interested only in XPath expressions (and not general XQuery), the result of executing an XPath expression can be mapped back into a sequence of well-typed XJ XML values in a straightforward manner. By basing our semantics for XPath expressions on that defined by XPath 2.0, we ensure that the result of executing an XPath expression in XJ on a file loaded into an XJ program is the same as if the XPath query had been executed by an XQuery engine on that file.

A side effect of the decision to use XPath 2.0 as the basis for the semantics of XPath expressions is that subtyping by extension has different semantics in the Java and XML components of XJ. Consider two Java classes c_1 and c_2, where c_2 extends c_1 with some fields. If a value of type c_2 is assigned to a variable of type c_1, the variable cannot "see" the c_2 fields. On the other hand, given two XML types x_1 and x_2 where x_2 is an extension of x_1, and a value of type x_2 assigned to a variable of type x_1, the extensions in x_2 will be visible to all XPath expressions executed on the variable because XPath is defined on the document structure, and only marginally aware of XML Schema types. As a result, XML extension behaves differently from class inheritance.

3.2 Loading and Serializing XML Documents

Instances of XJ XML types can be created in two ways: reading an XML document, or constructing an XJ XML value procedurally. We describe the first method here; construction of XML values is outlined in Section 3.3.

The class `XMLItem` defines a static function

```
XMLItem.load(String URI, Properties properties)
```

for parsing and validation of external XML documents. The `URI` parameter specifies the location of the XML document, and the `properties` parameter controls the parsing of the XML document. The set of properties allow the programmer control over certain aspects of parsing such as whitespace preservation. The result of the execution of this operation is a singleton sequence which contains an XJ XML value as defined by the XJ data model. For example, the following statement loads a document:

```
purchaseOrder p = (purchaseOrder)
    XMLItem.load("po.xml", properties);
```

Since the type returned by `XMLItem.load` is not known until runtime, it will be checked dynamically with respect to `purchaseOrder`.

To handle untyped documents (that is those that do not correspond to any schema), it is necessary to introduce a family of logical classes of the form e :: *anyType* and a :: *anySimpleType* for elements and attributes, where e may be any possible element name and a may be any possible attribute name. A detailed discussion of the handling of untyped XML data is beyond the scope of the paper.

There is an analogous static method in `XMLItem` for serializing XJ values to XML documents: `XMLItem.serialize( XMLNode value, OutputStream writer)`. The serialization of an XJ XML value will satisfy the property that serializing an XJ XML value and loading it back again (validating it against the proper schema if necessary) will result in the same XJ XML value (other than changes introduced by parsing properties).

The serialization of an XJ XML value is not necessarily unique. The information preserved about the source document from which an XJ XML instance is generated depends on the properties specified to the `XMLItem.load` method. While roundtripping is guaranteed in serializing XJ values and then loading the serialized value (XJ adds `xsi` attributes where needed), it is not guaranteed that loading an XML document into XJ and serializing it back into an XML document will result in the same document. Whitespace may not be preserved, and values of built-in types may be normalized. For example, leading zeros may be removed from numeric values. One could ensure full roundtripping by storing enough auxiliary information with XJ XML values, though we do not do so at the moment.

3.3 Construction of XJ XML values

The `new` operator can be used to construct new XJ XML values. For each non-abstract logical XML class, a program can create a new instance of the logical class by invoking the `new` operator. If the logical class corresponds to an element, $e :: t$, the arguments to the constructor contain a sequence `XML<`τ`>`, where τ is a regular expression corresponding to the content model of the complex type, t. As syntactic sugar for constructing an instance of `XML<`τ`>`, the program may provide a sequence of arguments to the constructor, from which an instance of `XML<`τ`>` will be constructed. Attribute values may be specified by `attrname=value` arguments to the constructor. For example, the following code creates a new `item` element(Figure 4 depicts the XJ XML value that is constructed as a result of this XJ code fragment):

```
quantity quan = new quantity(12);
productName pn = new productName("cup");
USPrice price = new USPrice(4.95);
item cup_order = new item(pn, quan, price,
                          partNum="456-CU");
```

A constraint placed on the arguments to element constructors is that none of the arguments can belong to another XML value (that is, if any of the arguments is an element, attribute, or atomic value, it cannot have a parent node). Otherwise, the situation might arise where a given node has more than one parent, which would mean that the XML data is no longer a tree and complicate the semantics of XPath and other constructs of XJ. For example, the following construction is illegal, and will be detected dynamically or, where feasible, statically:

```
quan = `cup_order/quantity`;
pn = new productName("saucer");
price = new USPrice(2.95);
item saucer_order = new item(pn, quan, price,
                             partNum="456-SC");
// now quan has two parent elements.
```

The problem is that if the construction were allowed, the node referred to by `quan` would have two parents: the nodes referred to by `saucer_order` and `cup_order`. This is not an issue for the nodes `pn` and `price` because they have no parent pointers before being passed to the constructor. XJ provides a `clone` method on each `XMLNode`, with deep-copy semantics that can be used to construct values from other XJ XML values. Using the `clone` operator, the following alternative would be legal:

```
...
item saucer_order = new item(pn, quan.clone(),
                    price, partNum="456-SC");
```

4. 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 processors such as Galax [10] 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, as seen to some degree in Section 3.3, one must ensure that every value inserted into another does not already have a parent node. Otherwise, a node might have have more than one parent, and the semantics of XPath expressions, serialization, etc. 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; however, to preserve the invariant that all XML values are trees, it is a runtime type error in XJ to insert a node with a parent into another node.

4.1 Updating XML Atomic Values

The mechanism for updating an XML value with an atomic type is to use XPath expressions to specify the element or attribute that is to be modified on the left-hand side of an assignment expression, and then, to provide a value of the appropriate type on the right-hand side. An update statement is legal only if the left-hand side evaluates to a singleton ordered sequence containing an XML value at runtime. For example, if `po` is a variable with XJ type `purchaseOrder`, the following statement updates a `po` element in place by performing a `replace` operation:

```
`po/item/productName/text()` = "Lawn mower";
```

One of the complications of supporting updates is the semantics of XJ with respect to updates, subtyping, and aliasing. For example, consider the following XJ fragment:

```
purchaseOrder::POType  po;
purchaseOrder::POSubtype  pos;
...
po = pos;
`po/item[1]/quantity/text()` = 750;
```

Assume that `POSubtype` is a subtype of `POType` derived by restriction with the constraint that `quantity` must be less than 500. Since POSubtype is a subtype of POType, one might assume that the assignment $po = pos$ should succeed. The update of the quantity element, however, causes a problem. The update is valid with respect to POType, but not with respect to POSubtype. To preserve type safety, this update would not be allowed and would cause a runtime error. If the compiler can statically determine that po refers to an instance of POSubtype, a compile-time error will be raised.

In XJ, as in Java, casting an XML value into a supertype does not change the value or the types associated with nodes. All updates and XPath expression evaluation are typed with respect to the runtime type of the value. This is analogous to overloading in Java where one may cast an object to a supertype but method calls invoke the overloaded implementation in the runtime class associated with the object rather than the implementation in the supertype.

4.2 Updating XML Complex Values

We now discuss how structural changes, such as inserting a new subtree or deleting nodes, can be performed on XML values. Consider the code sequence:

```
1  purchaseOrder po = ...
2  XML<item*> purchases = `po/item`;
3  item newitem = new item(new productName("Lawn Mower"),
               new quantity(1), new USPrice(148.95),
               partNum="123-LM");
4  item current = purchases.get(0);
5  current.insertAfter(newitem);
```

Line 5 uses an insertAfter operation to insert a new item after the current (in this case the first) item in the list of items. An insertBefore operation can be used to insert a new item before the current item: current.insertBefore (newitem);

Nodes may be deleted by performing a delete operation: current.delete();

When a sequence of updates is applied to an XML value, the constraint that an XML value must always be valid with respect to its type may be too rigid. It may be desirable to treat the entire sequence of updates as an indivisible operation, and perform validation on successful completion of the sequence of updates. The notion of deferred validation leads to an *unit of work* or *transaction* abstraction, where a sequence of updates is treated as an atomic unit. An area of future work is integrating such a concept into the language in a natural manner. One possibility is a lexically-scoped deferred validation block that ensures atomicity of modifications to a certain tree even in the presence of exceptions (which will undo the operation) or multithreading.

5. IMPLEMENTATION

We have built a prototype compiler for XJ that generates Java source from XJ source programs. The compiler is implemented with Polyglot [22], which provides a framework for parsing and typechecking 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 [7].

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 the least type. Given this information, our algorithm for typechecking XJ expressions and constructors is relatively straightforward. We do not, yet, typecheck the more complex aspects of the XJ type system, that is, the static typechecking of XML Schema constraints, such as *facets*, or updates of complex types (such as the insertion or deletion of a subtree). An obvious, but expensive, solution is to re-validate the entire document after each such expression. More efficient analysis is a subject for future research.

```
public void giveDiscount(){
   org.w3c.dom.Element po =
         XJImpl.loadLocal("po.xml", null);
   java.util.List bulkPurchases =
      XJImpl.searchList(po,
         "item[quantity/text() > 50]");
   for (int i = 0; i < bulkPurchases.size(); i++) {
      org.w3c.dom.Element current = bulkPurchases.get(i);
      java.util.List _tmp1 =
         XJImpl.searchList(current, "USPrice/text()");
      BigDecimal _tmp2 = new BigDecimal(0.80);
      BigDecimal _tmp3 =
         new BigDecimal(_tmp1.get(0)).multiply(_tmp2);
      XJImpl.updateAtomic(current,"USPrice/text()",_tmp3);
   }
   XJImpl.serialize(po, "po.xml");
}
```

Figure 5: Generated Java code for the example in Figure 1.

Once an XJ program has passed static typechecking, the XJ compiler emits Java code where the syntactic constructs introduced by XJ are erased to appropriate calls to the XJ runtime system. For example, Figure 5 depicts the code generated from the XJ code sequence of Figure 1. The key points to note are that all references to logical classes are erased to the appropriate DOM type or List (if one cannot determine that the result will be a singleton). XPath accesses are translated into calls into the runtime system (accessed through the class XJImpl), which invokes Xalan [1] to evaluate XPath expressions on the provided context node.[3]

A disadvantage of the simple code generation shown in Figure 5 is that explicit calls to an XPath engine like Xalan incur a significant overhead. We have implemented an optimized code generation scheme, XJ_{opt^0}, where simple (but common) XPath expressions are translated into explicit navigations of the DOM tree instead of calls to Xalan. For example the XPath expression USPrice/text() with the context node defined by current would be converted into a DOM code that accesses all children of current with element tag USPrice and returns their text content. This technique mitigates the overhead of the evaluation of XPath expressions.

6. ANALYSIS AND OPTIMIZATION

An XML document, which is logically structured as a tree, may be traversed in an XJ program using a combination of XPath operators (implicit traversal) and standard Java-based control flow constructs (explicit traversal). Standard

[3]Xalan implements XPath 1.0 whereas our semantics are defined in terms of XPath 2.0. We aim to utilize XPath 2.0 implementations once they are readily available.

optimization techniques such as as common subexpression elimination (CSE), loop-invariant code motion, and partial redundancy elimination (PRE) must be extended to handle XPath expressions as well as interactions between XPath expressions and Java constructs. In this section, we first describe optimizations for reducing parsing overhead and XPath evaluation overhead. We then detail some analysis problems that must be solved to enable these optimizations and present a brief overview of the optimization framework developed to address these issues.

Parsing Optimizations The parsing and building of an in-memory version of an XML document can have significant performance and memory overhead. In situations where only small fragments of a document are utilized, filtering the document and storing only the relevant portions in memory can improve performance substantially. By deriving a conservative estimate, in terms of XPath expressions, of the portions of an XML document that may be used by an XJ program, one can use streaming XPath processors, such as Xaos [2], to filter incoming documents. Such techniques have been shown to improve the performance of XQuery processors [18]. Extending such techniques to XJ requires the ability to analyze multiple XPath expressions in the context of Java control flow. Such information can also enable the use of techniques such as incremental parsing, where the program can selectively control parsing depending on control paths taken at runtime.

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. For example, 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.

Analysis Foundations In order to enable the optimizations described, it is necessary to be able to answer certain key XPath analysis questions. Given two XPath expressions x and y:

- Do the nodes in the result set of x precede (in document order) the nodes in the result set of y?
- Is the intersection between the result sets of x and y empty?
- Does the result set of x subsume that of y ?
- Can two XML references a and b be aliases, either by referring to the same tree or by one referring to a node within the tree referred to by the other?

Our analysis framework relies on transforming XPath expressions into a canonical representation for XPath expressions, called the XDAG, which was introduced in our prior work on the Xaos system [2]. The XDAG allows the determination of whether two XPath expressions are equivalent, even if they are not identical in structure. For example, the following two XPath expressions are equivalent, `p//a/b` and `p//b[parent::a]`, if p refers the root of an XJ XML value. Redundancy elimination can then be performed on sets of equivalent XPath expressions.

Benchmark	XJ$_{unopt}$	XJ$_{opt0}$	Java+DOM
Totals	>7,200,000	579	531
DBOneRow	4,124	47†	31
Periodic	388,660	156†	156
Mondial	1,023,078	62	46

Table 1: Running times in milliseconds for the tasks described in Section 7, excluding parsing. A (†) indicates that not all of the necessary optimizations described in Section 5 have been implemented yet, so some code was hand generated.

The XDAG and XML Schema information is then used to approximate the set of nodes returned by x and y by extents of the form, $E(x) = [lower(x), upper(x)]$ and $E(y) = [lower(y), upper(y)]$, where the *lower* and *upper* values are expressed using Dewey numbers [25]. Through these extents, one can determine ordering constraints between the results of XPath expressions. The types of the results of evaluating an XPath expression, which can be obtained by abstract evaluation of the XPath over the XML Schema [9], can be used along with the extent information to determine whether the result sets of two XPath expression may intersect. In order to determine whether the result of one XPath is contained in that of another, one can use standard XPath containment techniques [21]. Our situation is, however, complicated by issues such as aliasing. The presence of XML schema can help with the aliasing problem by providing rich type information. In addition, the use of XPath expressions increases the effectiveness of *escape analysis* and *ownership analysis*, since evaluation of an XPath expression cannot cause an XML node to escape. The presence of extents information can further assist with the calculation of alias information.

7. EXPERIMENTS

We have tested our prototype XJ compiler on a set of four (small) benchmark programs. Table 1 shows the results of our performance tests for the four benchmarks. Two of the benchmarks are translations of XSLT programs (Totals, DBOneRow) from the XSLTMark benchmark suite [6] into XJ. The other two benchmarks (Periodic, Mondial) represent XML processing for two publicly available XML documents:

Totals walks over a 10 megabyte XML document computing the sum of certain values. In XJ, this involves several simple XPath selections, which our prototype can compile into direct accesses of the DOM tree.

DBOneRow finds a certain row in a 10 megabyte, 48,700-row table, and displays the information that it contains. XJ finds the row with the query `table/row[id="0432"]`.

Periodic reads a 106 kilobyte document representing the periodic table of elements, which it sorts according to atomic weight.

Mondial manipulates the 1.28 megabyte MONDIAL geographic database [19]. It displays a summary of the information about each country in the database, and sorts the countries according to population.

Totals and DBOneRow were run on artificial datasets that were large enough for the running times to be measured with our instrumentation. However, Periodic and Mondial use real-world data for which a single pass did not take long enough for us to measure, so the numbers shown in Table 1 reflect 100 consecutive runs of these two tests.

We have written two versions of each test program, one in XJ using higher-level constructs for XML processing and the other in Java using the lower-level DOM API. We provide results for the Java version, "Java+DOM", and two sets of results for the XJ version: one for the simple code generation scheme, "XJ$_{unopt}$", and one for the optimized code generation scheme, "XJ$_{opt^0}$" (described in Section 5).

The results clearly show that processing all XPath queries at runtime (the "XJ$_{unopt}$" case) is very expensive, and that performance improvements of 100× to 10000× can be obtained by the optimized code generation represented by the "XJ$_{opt^0}$" case. Since the code generated by the XJ compiler for both the "XJ$_{unopt}$" and "XJ$_{opt^0}$" cases also uses DOM, the "Java+DOM" case represents a lower bound on the execution time that can be expected for the two XJ cases. The results in Table 1 indicate that the "XJ$_{opt^0}$" case can indeed approach the "Java+DOM" case in performance. The average gap for the four benchmarks is 23%, and we expect to further narrow this gap in future XJ compilers by incorporating the optimizations outlined in Section 6.

We ran our tests using Xerces version 2.5.0 for XML parsing [1], Xalan version 2.3.1 for runtime XPath processing, and IBM's Java 1.4.1 virtual machine on a 2.4 GHz Pentium 4 with 1 gigabyte 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 in Table 1 because all three cases use the Xerces parser, and the parsing time was the same in all versions. It is widely recognized that parsing is currently a major source of performance overhead in XML processing, and indeed our test cases each took at least 10 times longer to parse the data than to do the core tasks.

8. RELATED WORK

The languages most similar to XJ in design are XTATIC and XOBE, both of which integrate XML as a first-class construct into imperative object-oriented languages. Neither, however, support in-place modification of XML values. As a result, the manipulation of XML types in these languages is functional in nature, which does not mesh well with the imperative idioms of the base languages (C♯ and Java). Moreover, the data model and the semantics of XML types and values do not correspond exactly to those in XML 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 XPath expressions. While XOBE does support XPath expressions, subtyping is structural. Since a design goal of XJ is to provide abstractions intuitive to XML programmers, XJ is faithful to the XML Schema and XPath standards.

XQuery [15, 31] is a typed functional language for operating on XML documents. Like XJ, it is XML Schema aware. Currently, XQuery does not support updates. XL [8] is a language, based partially on XQuery, for programming web services. Both these languages are designed as stand-alone languages. 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. Where possible, however, a design goal of XJ has been to be close to XQuery semantics. For example, the data model, and subtyping, of XML values in XJ is similar to that of XQuery, and the semantics of XPath expressions are identical in XJ and XQuery.

XDuce [12, 13] and ℂDuce [3] are functional languages for writing programs that operate on XML data. Much of the foundation of integrating XML into programming languages was laid by the XDuce project, which also serves as the basis for the design of XTATIC. Neither XDuce nor ℂDuce support updates of XML values, nor are they consistent with XML standards such as XML Schema.

JWIG [5] 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.

Frameworks for Java-XML bindings [4, 14] generate Java classes statically from XML Schemas. JAXB [14] 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. A drawback is that 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 of binding approaches 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, programming language approaches such as XJ allow the programmer to develop applications natively in XML — the actual runtime implementation, which may use a binding 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.

9. 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. Changing the code generator to experiment with alternative binding mechanisms, for example JAXB classes, is one area of future research. Since the data model in XJ is similar to that of XQuery, integration with an XQuery engine such as Galax [10] as the backend is another option. Finally, we plan to explore static and dynamic optimizations in the XJ compiler.

10. 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 *ICDE*, pages 455–466, 2003.

[3] V. Benzaken, G. Castagna, and A. Frisch. ℂDuce: a white paper, 2002. PLAN-X Workshop, Pittsburgh.

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

[5] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 2003.

[6] DataPower Technology, Inc. XSLTMark 2.1.0. http://www.datapower.com/xmldev/xsltmark.html.

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

[8] D. Florescu and D. Kossman. An XML programming language for web service specification and composition. *IEEE Data Engineering Bulletin*, 24(2):48–46, June 2001.

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

[10] *Galax: An implementation of XQuery*. http://db.bell-labs.com/galax/optimization.

[11] V. Gapeyev and B. C. Pierce. Regular object types. In *ECOOP*, 2003.

[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 *ICFP*, 2000.

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

[15] H. Katz et al. *XQuery from the Experts. A Guide to the W3C XML Query Language*. Addison Wesley, 2003.

[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. I. Schwartzbach. Static analysis of XML transformations in Java. Technical Report RS-03-19, BRICS, May 2003.

[18] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, pages 213–224, 2003.

[19] W. May. Information extraction and integration with Florid: The Mondial case study. Technical Report 131, Universität Freiburg, 1999.

[20] E. Meijer and W. Schulte. Unifying tables, objects, and documents. http://research.microsoft.com/~emeijer/Papers/XS.pdf.

[21] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, pages 65–76, 2002.

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

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

[24] J. Siméon and P. Wadler. The essence of XML. In *Proceedings of POPL*, pages 1–13, Jan. 2003.

[25] I. Tatarinov et al. Storing and querying ordered XML using a relational database system. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, 2002.

[26] I. Tatarinov, Z. Ives, A. Halevy, and D. Weld. Updating XML. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, June 2001.

[27] A. Tozawa and M. Hagiya. Type-checking and node-completion for XML streams. In *In proceedings of 5th JSSST Workshop on Programming and Programming Languages*, May 2003.

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

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

[30] World Wide Web Consortium. *XML Path Language (XPath) 2.0*, November 2003.

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

[32] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000.

APPENDIX

A. SAMPLE SCHEMA

This schema, derived from that in the XML Schema specification [28], 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>
```