XJ: Integration of XML Processing into Java™

Matthew Harren
Dept. Comp. Sci.
University of California
Berkeley, CA
matth@cs.berkeley.edu

Mukund Raghavachari
IBM Research
19 Skyline Drive
Hawthorne, NY
raghavac@us.ibm.com

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

Michael Burke
IBM Research
19 Skyline Drive
Hawthorne, NY
mgburke@us.ibm.com

Vivek Sarkar
IBM Research
19 Skyline Drive
Hawthorne, NY
vsarkar@us.ibm.com

Rajesh Bordawekar
IBM Research
19 Skyline Drive
Hawthorne, NY
bordaw@us.ibm.com

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.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Design

Keywords

XML, Java, Data Integration

1. INTRODUCTION

XML [12] 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 [9] 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 [10], which builds an in-memory tree from an XML document, or SAX [8], 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 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 [11], 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 poster (and demonstration) 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 [4], XACT [5] and other languages [**?**, 6, **?**] 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 languages like Java.

This poster will introduce XJ, describe issues that arose in its design, and compare 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

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-912-8/04/0005.

```
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.

is standard Java code that accesses XML data using DOM. We will 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.

Brief Example We introduce the XJ language with the sample program listed in Figure 1. The schema used in this example (`po.xsd`) is that listed in the XML Schema primer [9].

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. IMPLEMENTATION

We have built a prototype compiler for XJ that generates Java source from XJ source programs (the architecutre of the XJ compiler is shown in Figure 2). The compiler is implemented with Polyglot [7], which provides a framework for parsing and typechecking Java source code, and implementing extensions to Java. XML Schemas imported by XJ

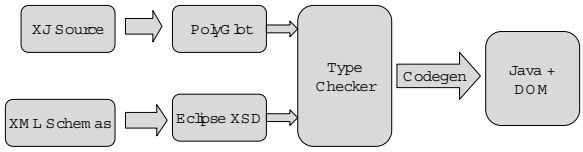

Figure 2: The structure of the XJ prototype compiler.

programs are parsed using the XML Schema Infoset Model plugin for Eclipse [2].

The type checking of XJ programs relies on the XAEL engine [3]. 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.

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. All references to XML types 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, which invokes Xalan [1] to evaluate XPath expressions on the provided context node.

3. REFERENCES

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

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

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

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

[5] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. Technical Report RS-03-19, BRICS, May 2003.

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

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

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

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

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

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

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