Reducing Application Code Complexity With Vocabulary-Specific XML Language Bindings

Jules White, Boris Kolpackov, Balachandran Natarajan, and Douglas C. Schmidt
{jules,boris,bala,schmidt}@dre.vanderbilt.edu
Department of Electrical Engineering and Computer Science,
Vanderbilt University, Nashville

Abstract

The eXtensible Markup Language (XML) has become a ubiquitous data exchange and storage format. A variety of tools are available for incorporating XML-based data into applications. The most common XML tools (such as parsers for SAX and DOM) provide low-level vocabulary-independent interfaces, which can make it hard to develop and debug robust applications. This paper examines tools for generating vocabulary-specific XML-to-C++ language mappings and shows how they can reduce key sources of complexity associated with developing object-oriented XML-based applications. The paper also presents criteria for evaluating tools that generate vocabulary-specific language mappings and applies these criteria to compare five tools for this purpose: XML Spy, Xbinder, Object Link, Liquid XML Data Binding Wizard, and XML Schema Compiler (XSC). Our results show that XSC is the only tool that provides a complete vocabulary-specific mapping, alignment with the C++ Standard Library, and code portability, while also providing the most manageable generated code base.

Keywords: XML, C++, Vocabulary-Specific Language Binding, W3C XML Schema, DOM, SAX

1. Introduction

XML [1] has become one of the most prevalent formats for data exchange and storage for computer-based systems. It provides a loose tree-based structure well suited for semi-structured data [2,3]. XML's self-describing nature, human readable element names, and ability to reference external document specifications allows applications to exchange and dynamically interpret data without a shared set of assumptions, such as interface definitions via header files. Applications therefore become less dependent on strict, statically defined interfaces provided by their peers [4].

Developing applications that exploit the flexibility of XML data can be complex, however, since XML only specifies a common format for data it encapsulates and does not specify semantics or type information [5]. This paper evaluates two approaches to accessing XML data from inside applications: *vocabulary-independent* data access interfaces (DAIs) and *vocabulary-specific* DAIs. An XML *vocabulary* is a specialization of XML for a particular type of application or format that describes the names of elements and attributes, their meaning, and the structural relationship between them. Languages such as XML document type definitions (DTD) and XML Schema [6,7,8] are commonly used to define XML vocabularies, such as TeXML and XML Schema.

1.1 Vocabulary-Independent DAIs

We begin our discussion of vocabulary-independent DAIs by showing an example implementation of a C++ program designed to print the title of each book in a library written by Tolstoy. The following XML file is used as the input to the program:

```
<library>
 <book>
  <title>War and Peace</title>
  <author>Tolstoy</author>
 </book>
</library>
```

The following implementation uses the vocabulary-independent DOM interface provided by the Apache Software Foundation's C++ DOM implementation (Xerces C++) [10]:

```
DOMNodeIterator i = ...//Get a book iterator
xstring title;

for (DOMNode* n = i->nextNode ();
       n != 0; n = i->nextNode ()) {
  xstring name (n->getNodeName ());
  if (name == "title") (
    title = static_cast<DOMText*>
       (i->nextNode ())->getNodeValue ();
  }
  else if (name == "author") {
    xstring author = static_cast<DOMText*>
        (i->nextNode ())->getNodeValue ();
    if (author == "Tolstoy") {
      cerr << title << endl;
    }
  }
  else {
    // error
  }
}
```

As shown above, vocabulary-independent DAIs focus on generalized tree-based concepts of XML. The two most widely used vocabulary-independent DAIs are the Simple API for XML (SAX)[11] and the Document Object Model (DOM) [9]. SAX uses an event-based archi-

tecture that parses XML and notifies registered observers as the parser encounters XML elements of interest, such as tag pairs. DOM creates an in-memory representation of the relationships within the XML.

Using vocabulary-independent DAIs in applications can be tedious and error-prone [24]. In particular, application code that performs computations on XML data utilizing a vocabulary-independent DAI can be complex and in many instances tightly coupled to the data layout specified by the Schema/DTD (e.g., the element order). The vocabulary-independent interfaces are low-level and rarely provide enough application-specific semantics to allow direct computations on the data. Developers who use XML in applications written with third-generation programming languages (such as C++, C#, and Java) have traditionally been responsible for devising *ad hoc* ways of implementing transformation of the data structures in the language binding to ones more suitable to their computations [12,13]. These implementations are typically achieved by extracting data from generic XML DAIs and placing it into containers with interfaces specific to the vocabulary.

1.2 Vocabulary-Specific DAIs

An alternative approach to vocabulary-independent XML data access is the vocabulary-specific DAI which bridges generic XML concepts and the application-specific ones. The following example presents a concise and type-safe vocabulary-specific interface for accessing books in an object-oriented language, such as C++:

```
class Book {
public:
  string title () const;
  string author () const;
}
```

Vocabulary-specific DAIs relieve developers from the burden of mapping data from a vocabulary-independent DAI to application-specific data structures. Developers can focus on the semantics of the data they are manipulating, while leaving the type conversion to the vocabulary-specific DAI implementation. Vocabulary-specific DAIs can be generated automatically from Schema definitions, and when specific to a particular class of documents, provide developers with a more robust and intuitive interface to the underlying data. For example, our DOM-based application to print the titles of books written by Tolstoy could be rewritten in a more intuitive fashion with a vocabulary-specific DAI.

The XML Schema Compiler (XSC) [14], developed by Vanderbilt University's Distributed Object Computing (DOC) Group, was used to generate a vocabulary-specific DAI for books. The following code fragment uses this DAI to print the titles of the books by Tolstoy:

```
Book b = ...
if (b.author () == "Tolstoy") {
  cerr << b.title () << endl; }
```

This vocabulary-specific code is considerably shorter, simpler, and more readable than its vocabulary-independent DOM equivalent. Even in this small example, the vocabulary-specific DAI simplifies development significantly. More importantly, as the complexity of underlying data increases, there is a proportional decrease in coding complexity using the vocabulary-specific approach as opposed to the vocabulary-independent approach.

The remainder of this paper discusses our experience using and evaluating different vocabulary-specific DAIs. This paper makes two main contributions: (1) we propose metrics to be used for evaluating vocabulary-specific DAIs and (2) we use those metrics to evaluate five different tools that provide this capability for C++. Our experience suggests that despite the popularity of XML, the tools that provide such advanced capabilities have different levels of maturity, which affects the quality of next-generation applications being developed.

Section 2 of the paper provides the motivation and overview of the different metrics used to evaluate vocabulary-specific DAIs. Section 3, provides the empirical results and analysis of the data collected from the evaluation of five different vocabulary-specific DAIs, and we conclude this paper in Section 4.

2. Evaluating Vocabulary-specific DAI Tools for C++

Tools such as Java Architecture for XML Binding (JAXB) [15], Castor [16], and Xbind[17] that generate vocabulary-specific DAIs are available for many popular programming languages, including Java, C++ and Python. These tools can use a large number of validation languages to generate the interface, ranging from W3C XML Schema to DTD. Our analysis in this paper centers on tools for generating C++ mapping using W3C XML Schema.[1] We focus on C++ since it is the language of choice for many application domains, in particular distributed real-time and embedded (DRE) systems, which is the focus of our research.

There are many tools to generate vocabulary-specific C++ mapping, including Rogue Wave's Object Link [18], Liquid Technologies' XML Data Binding Wizard [19], Objective System's Xbinder [20], XML Spy's class generation [21], and the XML Schema Compiler (XSC).. Although there is no standardized interface for C++ mapping, each tool offers a basic set of accessors, mutators, and sequence traversal methods for types described in a Schema. Each tool also hides the conversion from lexical space (i.e., XML text) to the semantic space (i.e., instances of types).

All five tools help reduce the difficulty of accessing XML data. To make our discussion more rigorous, however, this section presents a series of criteria that enable

[1] For brevity, we will refer to W3C XML Schema as simply *Schema* throughout the rest of the paper.

systematic evaluation of the tools listed above. In particular, we describe our criteria for quantifying the level of assistance each tool provides to ease the development and maintenance effort associated with manipulating XML data.

Criterion 1: Completeness of vocabulary-specific mapping

Our first criterion allows comparison of tools based on completeness of vocabulary-specific interface, i.e., an interface should allow manipulation of each item in an XML document via the equivalent C++ type mapped from the declared type in schema. For example, a complete vocabulary-specific DAI should map any XML Schema supported simple type to an equivalent C++ type (if one exists) and map any aggregate type to a corresponding generated C++ class. The generated class should contain member variables for each of the child elements encompassed by the mapped XML element. A comprehensive mapping is important to relieve application developers from the burden of type conversion and dynamic type checking. It also allows application developers to work with abstract data types meaningful to their application's working vocabulary. Vocabulary-specific DAIs reduce the number of possible errors caused by dynamic conversion of XML text to instances of types in the target language, thereby eliminating error-prone constructs, such as *string-based flow of control*, where conditional statements are based on comparison of two strings.

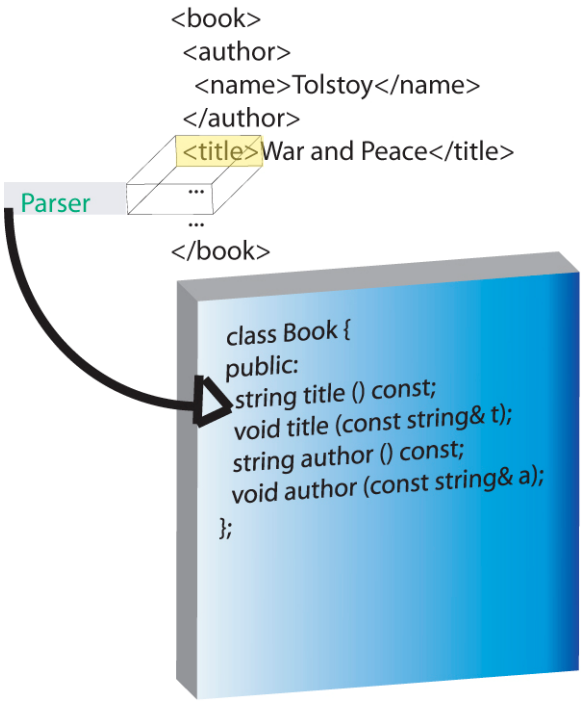

Figure 1: Vocabulary-Specific Interface

Figure 1 shows a correspondence between an XML instance and a C++ interface. C++ and Schema share most of the same numeric types, such as short, long, and double. Schema's simple types also include types that are not native to C++, including IDREF, which is a mechanism similar to C++ pointers. The vocabulary-specific DAI should also map each of these types to a useful type in the target language. Although it is not strictly a type in Schema, we will discus the Schema restriction "enumeration" in our analysis of the completeness of the vocabulary-specific mapping since "enumeration" can be conveniently mapped to a C++ enum and logically can be viewed as part of the Schema type system.

An interface can have the following levels of completeness:

- Enumerations, aggregate types, and basic types are mapped to C++ types. IDREF is mapped to a mechanism for obtaining a handle for the referenced data (complete mapping).
- Enumerations, aggregate types, and basic types are mapped to C++ types (partial mapping).
- Aggregate types and basic types are mapped to C++ types (minimal mapping).
- Aggregate types and/or basic types are not properly mapped to C++ types (no mapping).

Criterion 2: Alignment with the C++ Standard Library. Since the generated vocabulary-specific DAI must ultimately be integrated and used by C++ applications, the generated code should provide a standard and intuitive interface. This criterion therefore evaluates how well aligned the generated interface is with the C++ Standard Library. A high degree of alignment saves developers from learning a new set of containers and utility classes and creating complex code to plug them into the C++ Standard Library algorithms. A desirable goal is to provide a DAI that reduces interface complexity, but not one that reduces complexity in one area while increasing complexity in another. Finally, C++ Standard Library alignment allows applications to benefit transparently from future enhancements.

For example, Xerces C++ uses its own implementation of strings, which forces developers to learn not only how to use Xerces C++ strings but how to map from Xerces C++ strings to C++ Standard Library strings. This mapping introduces unnecessary conversion code that can be a source of information loss and memory leaks or corruption.

A vocabulary-specific language mapping can have one of the three levels of alignment listed below:

- Applications can directly use provided containers and iterators with the C++ Standard Library algorithms and classes. (complete alignment).
- Interfaces generated by the tool can provide container and iterator concepts that are similar to the Standard C++ Library, but cannot be used with C++ Standard Library facilities without adaptation logic within the application code (partial alignment).

- Complex code is required to convert from containers used by the interface to those used by the C++ Standard Library (no alignment).

Criterion 3: Code portability. The third criterion we examine is code portability, which is defined as the ability to compile the generated DAI code on multiple OS platforms that support standard-compliant C++ compilers. Although not all applications require code portability, it is desirable that generative tools for DAIs produce portable code which can be compiled and executed on a variety of OS platforms. In particular, the generated code should not restrict the application code any more than the underlying XML parsing layer (if any). We also consider it a positive attribute if the code generator itself can be executed on multiple OS platforms.

Vocabulary-specific DAI tools can have the following levels of portability:

- The tool can generate one code base that can compile on multiple platforms without modification (complete portability).
- The tool supports generating code for several platforms but a separate code base must be generated for each platform (partial portability).
- The tool does not generate portable code out of the box but can do so through customization (no portability).

Since the goal of these tools is to simplify development they should not force developers to maintain multiple code bases or spend time customizing the tool output .

Criterion 4: Manageability of generated code. Our fourth and final criterion considers manageability by measuring the amount of generated code needed to provide an equivalent set of vocabulary-specific interface features. For each line of code generated by the tools, there will be a corresponding complexity increase in each of the following five areas:

1. ***Source control***, i.e., each line of generated code will need to be integrated into the source control procedures for the application
2. ***Documentation***, i.e., the purpose and use of the generated interface will need to be incorporated into the API documentation
3. ***Build configuration***, i.e., the code will need to be integrated into the application build process
4. ***Compilation time***, i.e., poorly designed headers and interfaces when included into the application source can increase compilation (and recompilation) time during development.
5. ***Binary size***, i.e., the more code integrated into the application, the larger the application binary.

To enhance manageability, therefore, it is important that the generated DAIs are efficient and require as few files and lines of code as possible.

Another measure of manageability of code is how schema definition inclusions (file inclusion of an external schema) are handled. Anywhere that the schema has been broken into modular pieces, and aggregate documents have been constructed through file inclusion, the generated code should also be separated into reusable libraries. Decoupling the generated interfaces in this way helps to (1) mirror the intent of the developer to separate the schema into manageable pieces and (2) create more modular and reusable code.

Vocabulary-specific DAI tools can have various levels of manageability:

- The tool generates separate interfaces for each schema and file inclusion is used to aggregate the functionality (complete manageability/extensibility).
- One interface is generated that aggregates the functionality by directly combining the interfaces in the same file (partial manageability/extensibility).
- Schema inclusion is not supported (no manageability/extensiblity).

In general, a more manageable way to generate code for schema A and B would be to have separate interfaces and let A's interface use B's. If separate interfaces are not generated and the application later needed to use B's interface separately, an entirely new interface would need to be generated for B, which would increase the size of the generated code base and create unneeded code duplication. Generating separate interfaces avoids this problem.

3. Empirical Tool Comparison and Analysis of Results

This section reports the results of evaluating five tools that generate vocabulary-specific DAIs XSC, Object Link, XML Spy's Code Generator, Xbinder, and the Liquid XML Data Binding Wizard for a simple library example against the criteria presented in Section 2. In particular, we expand our simple book example from Section 1.1 to create a library that contains 1…N books. Each book stores the author, main characters, ISBN, genre, title, availability, and a unique id. The complete schema is available at www.dre.vanderbilt.edu/~jules/library.xsd. Particular items of interest in our schema are (1) the "Genre," which maps to an enumeration, and (2) the "id," which allows other elements to refer to a particular book. Since books are catalogued in a library, we want to ensure that all of our books are only classified by genres that we use to organize the library. Our genre enumeration will help enforce this constraint. The "id" will be used to implement a recommendation system that allows a reader to see books each author recommends.

Each tool handled the mapping of schema basic types to their equivalent C++ types. The interfaces they generated and the completeness of the vocabulary-specific mapping varied significantly, however. The remainder of this section presents the results of applying the four criteria described in Section 2 to the code generated by the tools to evaluate provided reductions in code complexity and ease of integration into C++ applications. The results in this section show that the tools possess a wide range of capabilities. In particular, the completeness of the vocabulary-specific mapping varies from

poor to excellent between the tools. Figure 2 summarizes the results of our evaluation.

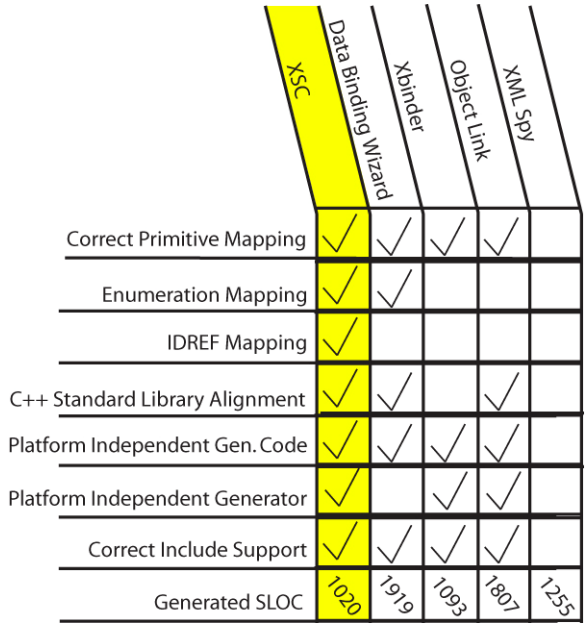

Figure 2: Summary of Experimental Results

Criterion 1 results: Completeness of vocabulary-specific mapping. Although all tools correctly converted the simple types to their corresponding C++ types and aggregate types to classes, XML Spy's generated code created a DAI that disregarded the quantifier restrictions within the library schema. For example, a `long` and an array of type `long` are different types. A compilation error would occur if one attempted to index into a `long` using the subscript operator. The "title" element of "book" in our example schema can occur exactly once but the XML Spy generated code provided two interfaces to access the title: `CTitle GettitleAt(int nIndex)` and `CTitle Gettitle()`, which gives a developer the ability to write code such as:

```
CTitle book_title = book.GettitleAt (2).
//Equivalent to :
//long mylong =…;
//long b = mylong[2];
```

As discussed in Section 2, Criterion 1, the purpose of the vocabulary-specific DAI is to prevent developers from creating run-time errors by migrating type-checking to compile-time rather than run-time. By allowing application developers to write code that violates the cardinality restrictions within the schema, the generated code was creating a less complete vocabulary-specific interface.

The code shown above should be flagged as a typing violation at compile-time. With XML Spy's generated code, however, there is no compile-time error since the interface provides a method to get any title from 1...N. For the completeness of the vocabulary-specific mapping criterion, therefore, all except XML Spy passed the conversion test from schema types to C++ types. XML Spy provides developers with the ability to customize the code generation template, which allows developers to fix this problem with additional effort. All other tools, however, generated proper mappings without customization.

We next consider a more complex type mapping from schema enumeration to C++ enumeration. The conversion from enumeration was much less uniform between tools. XSC and the Liquid XML Data Binding Wizard mapped enumerations to C++ enumerations. XML Spy and Object Link mapped the enumerations to strings. Finally, Xbinder mapped the enumerations to integer types. Bindings to C++ enumeration alone offer true vocabulary specificity. For example, application developers could produce code such as the following with the enumeration-to-string mapping:

```
if (strcmp (book.genre (), "suspense") == 0)
{
  //…
}
```

If "suspense" was not a valid enumeration value, the code would still compile; the same is true for the integer mapping. An application developer could easily test the value of genre against any integer regardless of whether it was a valid enumeration value. With XSC and the Liquid XML Data Binding Wizard's enumeration mapping, a comparison to any value other than the ones defined by the schema enumeration results in a compilation error.

A more complex case for the vocabulary-specific mapping arises with `ID`'s and `IDREF`. The `ID`/`IDREF` attributes provide a mechanism by which one element can refer to another element using its unique `ID`. This presents an interesting challenge for the mapping tools. Since the `IDREF`'s can refer to any arbitrary element, it is hard for a tool to generate a vocabulary-specific mapping. Tools therefore use one of two approaches:

- ***Return the string value of the referenced ID***. This approach does not provide any actual vocabulary-specificity. Once the `ID` string is obtained, the application must iterate over all the nodes that could be referred to (possibly every node in the document) doing a string comparison on each element's `ID` to find the referenced instance. This approach is similar to DOM-based implementations, though it can be even more complex than a DOM-based approach since the reference can be to any other element in the document and we no longer have a generic interface with which to manipulate the data.
- ***Return the actual instance referenced***. This approach is more strongly typed, but requires the instance be returned in the form of a reference to a common base class. This reference can then be cast to the desired derived class, which alleviates developers from writing generic traversal code to search for the referenced instance. XSC is the only tool we tested that returns the actual instance referenced rather than the string value of the `IDREF`.

Criterion 2 results: Alignment with the C++ Standard Library. The interfaces generated by XSC, Object Link, and the Liquid XML Data Binding Wizard are all aligned with the C++ Standard Library conventions, i.e., they either directly use C++ Standard Library containers or they provide iterators that are interoperable with the C++ Standard Library. XML Spy and Xbinder, however, use proprietary containers and do not provide mechanisms to use C++ Standard Library

Criterion 3 results: Code portability. All the tools tested have the ability to generate platform-independent DAIs, though XML Spy does not generate platform-independent code by default (its generated code is specific to Windows). Instead, developers must customize its proprietary code templates to obtain platform portability. One code base from each tool supports multiple platforms. The developer is not required to maintain a different set of binding code for different platforms. The code generators themselves, however, were not all platform independent, i.e., XML Spy and the Liquid XML Data Binding Wizard only support Windows.

Criterion 4 results: Manageability of generated code**.** We used the *SLOCCount* (Source Lines of Code Count) tool [22] to determine how large of a code base developers would need to incorporate into their applications. We counted the total lines of code for each of the vocabulary-specific DAIs generated for our library example. The results are listed in Figure 3. The generated code varied quite a bit in size. The largest code base, generated by the Liquid XML Data Binding Wizard, was almost twice the size of the smallest generated by XSC. The larger code bases also resulted in a larger number of source files. The smallest code bases compacted the code into two source files, while the larger code bases generated a dozen or more. XSC and Xbinder generated the smallest code bases. XSC's code base was roughly 70 lines smaller than Xbinder.

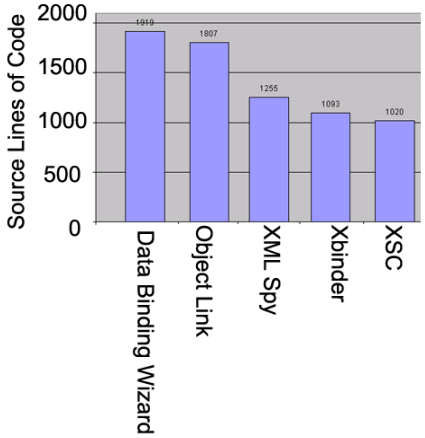

Figure 3: Generated Lines of Code

We also tested the schema inclusion mechanism discussed in Section 2, Criterion 4. We created a copy of the original schema with a different target namespace. We then included the copy within the original schema and changed the book element's character child to reference the `Character` type in the copy schema. XSC, Object Link, and the Liquid XML Data Binding Wizard all generated separate interfaces for the two schemas and included the copy's interface, via file inclusion, within the original. XML Spy also handled the schema file inclusion, but generated the included schema's interface within the main interface, which created redundant classes and made the copy's interface unusable on its own. Xbinder also tried to include the referenced schema but did not properly handle namespaces. Its generated code allowed a naming collision between the original `Character` and the copy's `Character`.

4. Concluding Remarks

The flexibility of XML motivates its popularity as a useful data exchange and storage format. The most prevalent approaches to XML however, use vocabulary-independent data access interfaces (DAIs) geared towards low-level manipulation of XML with generic XML concepts (elements/attributes). It can therefore be non-intuitive, tedious, and error-prone to incorporate these DAIs into object-oriented applications. This paper describes and evaluates a set of tools that help to solve this problem by providing richer *vocabulary-specific DAIs*. These tools generate vocabulary-specific code for accessing XML data via a developer's programming language of choice. These generated DAIs can reduce the complexity of data access code significantly.

To systematically evaluate vocabulary-specific DAI tools, we devised four criteria that evaluate the reduction in data access complexity. These criteria evaluated the ease with which both the tool and generated code can be incorporated into object-oriented C++ applications. The criteria also analyzed the types of errors common to vocabulary-independent DAI approaches for XML that can be eliminated via vocabulary-specific DAIs. By applying our criteria to five of tools in the context of a sample application, we found significant variation in the tools available to generate C++ DAIs.

Our tests found that XSC was the only tool that generated comprehensive vocabulary-specific DAIs. The Liquid XML Data Binding Wizard had the second most complete mapping with its support for enumerations. Almost all of the tools provided data access using the C++ equivalents of the Schema declared types. We also found that XSC produced the smallest code base.

Our tests relied on an example schema that covered a range of possible schema elements. Our future work will extend this testing to cover an even broader range of schemas, along with tests to evaluate other important quality and performance measures, such as compilation time, data access time, and memory footprint. In addition to conducting these tests, we also plan to enhance XSC to support generation of interfaces in other target languages, including Java and the Object Management Group's Interface Definition Language.

References

[1] T. Bray, J. Paoli, C. Sperberg-McQueen, "Extensible Markup Language (XML) 1.0 (Second Edition)," World Wide Web Consortium (W3C), October 2000.

[2] F. Simeoni, P. Manghi, R. Connor, D. Lievens, S. Neely, "An Approach to High-level Language Bindings to XML". Special Issues of the *Information & Software Technology*, Elsevier Ed. (4), 2002.

[3] P. Buneman, "Semistructured Data," in *Proceedings of the Sixteenth ACM SIGACT - SIGMOD - SIGART Symposium on Principles of Database Systems*, 1997.

[4] F. Simeoni, D. Lievens, et al., "Language bindings to XML," *IEEE Internet Computing*, Jan 2003.

[5] P. Buneman, W. Fan, J. Simeon, S. Weinstein, "Constraints for Semistructured Data and XML," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 2001

[6] D. Lee, W. Chu, "Comparative Analysis of Six XML Schema Languages," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 2000

[7] P. Biron, A. Malhotra, "XML Schema Part 1 Structures" *World Wide Web Consortium (W3C) Working Draft*, December 1999.

[8] P. Biron, A. Malhotra, "XML Schema Part 2 Datatypes" *World Wide Web Consortium (W3C) Working Draft*, December 1999.

[9] A. Le Hors, P. Le Hgaret, et al, "Document Object Model (DOM) Level 2 Core Specification (Version 1.0)," *W3C Recommendation, World Wide Web Consortium* (http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/), 2000.

[10] The Apache Software Foundation, "Xerces-C++," 2004 (http://xml.apache.org/xerces-c/)

[11] Megginson Technologies Ltd, "SAX 2.0: The Simple API for XML (http://megginson.com/SAX/)", (2000).

[12] T. Milo, S. Zohar, "Using Schema Matching to Simplify Heterogeneous Data Translation," in Proceedings of the 24th International Conference on Very Large Data Bases, VLDB, 1998

[13] C. Beeri, T. Milo, "Schemas for Integration and Translation of Structured and Semi-Structured Data," *Lecture Notes in Computer Science*, 1999

[14] Distributed Object Computing Group at Vanderbilt, "XML Schema Compiler," 2004 (http://www.dre.vanderbilt.edu/~boris/xsc/).

[15] Sun Microsystems, Inc., Java Architecture for XML Binding (JAXB), 2001 (http://java.sun.com/xml/downloads/jaxb.html).

[16] ExoLab Group, The Castor Project, 2004 (http://www.castor.org).

[17] P. Prescod, Xbind 0.7 Tutorial, 2004 (http://www.prescod.net/xml/xbind/).

[18] Rogue Wave Software, "LEIF: Data Tier, Object Link," 2004 (http://www.roguewave.com/products/leif/data.cfm).

[19] Liquid Technologies, "XML Data Binding Wizard 3," 2004 (http://www.liquid-technologies.com/Products.htm).

[20] Objective Systems, "Xbinder," 2004 (http://www.obj-sys.com/products_xbinder.shtml).

[21] Altova, "XML Spy," 2004 (http://www.xmlSpy.com).

[22] D. Wheeler, "SLOCCount," 2004 (http://www.dwheeler.com/sloccount/).

[23] F. Simeoni, D. Lievens, R. Connor and P. Manghi, "Language Bindings to XML", *IEEE Internet Computing*, Vol 7, Issue 1, 2003.