Issues in Generating Data Bindings for an XML Schema-Based Language

Eric M. Dashofy

Department of Information and Computer Science
University of California, Irvine
Irvine, CA, 92697-3425, U.S.A.
+1 949 824 2260
edashofy@ics.uci.edu

ABSTRACT

XML's meta-language aspect and extensive tool support make it an attractive way to build modularly extensible modeling languages. XML's original meta-language, the document type definition (DTD), is being replaced by the more expressive XML schema. Developing programmatic tools to manipulate models specified in XML schemas is made easier though the use of data bindings. Data-bindings model elements and attributes in XML schemas as objects in an object-oriented programming language. We have developed an XML-schema aware generator for Java data bindings called 'apigen.' While developing apigen, we encountered and worked through several issues, both essential and accidental, related to generating XML schema data bindings. These issues, and the solutions we developed, are described in this paper.

Keywords

XML schema, XML, data bindings, Java, apigen.

1. INTRODUCTION

One of XML's most powerful features is its meta-language aspect. While XML [4] primarily began as a document markup language, its usefulness as a common format for exchanging and representing all sorts of data structures has been well-proven. The document type definition (DTD) meta-language that accompanies XML 1.0 [4] allows the development of new XML-based languages. DTDs allow developers to specify the contents of elements and attributes in XML instance documents. However, the expressive power of DTDs is limited. DTDs do not allow specification of simple types like booleans, floats, or integers. DTDs do not allow patterns (like regular expressions) to be specified in simple attribute or element types. Most importantly, DTDs do not have a mechanism by which derived attribute and element types can be created to add or remove elements from a base type [13].

DTDs, to a certain extent, allow modular extensibility through a construct known as a *parameter entity.* A parameter entity allows one DTD to reuse (import) element and attribute definitions from another DTD. An example of using this method to create a modularly extensible metalanguage is the Modularization of XHTML [2] W3C recommendation. This recommendation breaks down the existing specification of XHTML into modules which provide various aspects of XHTML functionality. Language creators can compose these modules into *hybrid document types* that combine aspects of each module into a new document type. The main disadvantage of using parameter entities in this way is that a new hybrid document type definition must be created for each new combination of definitions required by XML instance document authors.

The development of the XML schema recommendation [14, 3] by the W3C and the emergence of associated tools have made it possible for developers to more easily create modularly extensible languages as XML applications. XML schemas have a type-inheritance mechanism similar to that found in object-oriented programming. Through this mechanism, extension schemas can define derived types that add and remove elements and attributes to/ from base types. The base schema need not be modified. An element of a derived type may be substituted anywhere an element of its base type is required. No hybrid document type definition is needed. This ensures that instance documents are valid to both the original schema and individual developers' extension schemas.

The creation of extensible languages with XML schemas necessitates the creation of equally flexible and extensible tools to support these languages. Each tool working with an extensible language must be able to quickly adapt to new language extensions and properly deal with unknown extensions without disturbing them.

Programmatic manipulation of XML documents is usually done through one of the available XML APIs: DOM [9], SAX [11], or JDOM [8]. These APIs provide generic interfaces that correspond to the semantics of XML. Constructs in these APIs include *elements*, *attributes*, and *documents*.

While XML schemas (and DTDs) are generally used as a basis for validating instance documents, they can also be used as a road map for generating friendly, syntax-based programmatic interfaces to XML documents. Because the syntax of an XML document can be defined completely by a schema (or set of schemas), automated tools can generate data bindings based on the syntax defined in the schema.

We have constructed one of the first data-binding generators that works on XML schemas, called 'apigen.' Given a set of schemas, apigen generates an object-oriented class library that allows manipulation of XML documents that conform to those schemas. When schemas change or new schemas are added, users simply re-run the tool to generate new bindings. While developing this generator, we encountered several salient issues that complicated our development. Some of these issues were accidental, such as immature XML tools. Some were more fundamental, such as XML's limited support for dealing with unknown types. The balance of this paper describes the motivation for our work, the issues and challenges we faced building apigen, and solutions to these issues that we incorporated into the apigen tool.

2. MOTIVATION

My research group developed a modularly extensible language when we were defining a new representation for software architectures [12]. Our research group and one from Carnegie Mellon University wanted to agree on a common core set of architectural constructs, to be defined in XML, that we could share. However, each group wanted to pursue its own research interests, adding new concepts to the core and experimenting with new modeling constructs. As such, we defined the core language, which

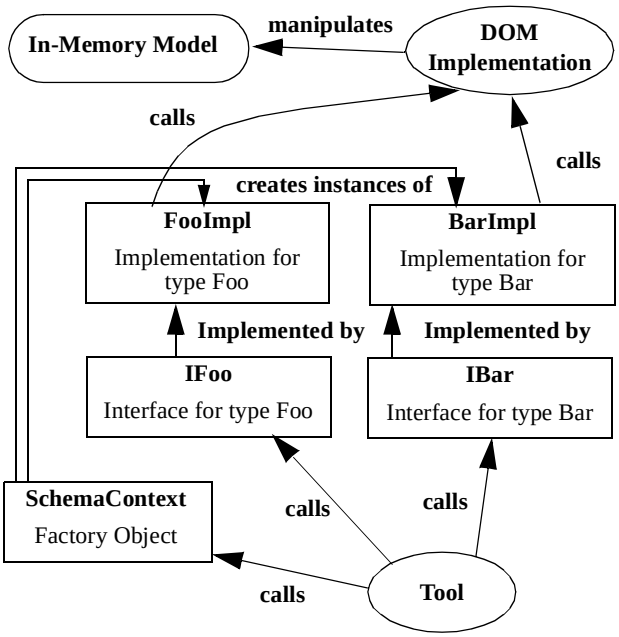

Figure 1. Relationship between tools, classes generated by apigen, the DOM interface, and the in-memory model. Rectangular boxes represent individual Java classes generated by apigen.

we call xArch [5], in an XML schema. Each research group has since built its own extensions to this core representation, leaving the core unchanged. Currently, our group has defined six extensions to the core representation of various sizes. Our set of extensions is growing as our research interests evolve.

3. APPROACH

Despite the fact that XML is mostly human-readable, the amount of namespace information and markup that is present in a typical XML document makes it unreasonable to write one by hand. Unlike traditional architecture description languages [10], which structurally resemble programming languages, xArch documents are intended to be read and written entirely through tools—the underlying XML basis of xArch documents is hidden from the end-user.

Developing such tools requires a programmatic interface to xArch documents. As noted above, APIs like DOM and JDOM can be used to manipulate XML documents in general. However, these APIs fail to take advantage of the additional structural information about XML documents provided by their XML schema(s). To take advantage of this information, we have developed a Java data-binding generator for xArch schemas, called 'apigen.' While apigen was designed specifically for xArch schemas, the techniques we used to develop it are generally applicable to generating data bindings for arbitrary XML schemas.

3.1 Apigen

Apigen takes, as input, a set of XML schemas. Because of the size of the XML schema language, apigen supports only a reduced subset of XML schema constructs. This subset includes simple and complex types, attributes, varying cardinalities of elements, type derivation, and enumerations. Additional constructs can be added to apigen at incremental cost. This subset of the XML schema language satisfies a large domain of schemas, and was sufficient for all the schemas we developed in the evolution of xArch.

Apigen produces, as output, a set of Java interfaces and their implementations. An interface/implementation pair is generated for each type defined in an XML schema. Each schema passed to apigen is assumed to declare its own target namespace. For each namespace, apigen generates a new Java package. All manipulation of the object model is done through the Java interfaces. The generated implementations of these interfaces are never directly instantiated or referred to by users. Rather, apigen generates a *factory* object for each package that can create instances of all interfaces used by that package. Derived types in an extension XML schema are represented as subinterfaces and subclasses in Java, providing a natural mapping between XML type inheritance and object-oriented inheritance.

The objects generated by apigen use an object-oriented model with a DOM interface as the underlying representation format for XML data. Operations on apigen-generated objects result in manipulations of the in-memory model. Apigen currently uses the Apache Xerces [1] XML parser and DOM implementation, but all parser-specific functionality is encapsulated in a single class. The relationship between the in-memory model, DOM, classes generated by apigen, and external tools is depicted in Figure 1.

DOM is a fairly ubiquitous interface, implemented in many languages, that allows random access to the XML tree. While DOM exposes functions generally useful for dealing with any

APIGEN-GENERATED OUTPUT

For a technical perspective on apigen's generated interfaces, consider the following example:

The following is an XML schema type, defined in the XML schema language:

```
<xsd:complexType name="TrivialArchitecture">
   <xsd:sequence>
   <xsd:element name="description"
    type="Description"/>
      <xsd:element name="component"
       type="Component" minOccurs="0"
       maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
```

This hypothetical type defines a trivial architecture for a software system, containing a description and zero or more components. Apigen, on encountering this type, would generate interface functions as follows in a new Java interface called ITrivialArchitecture:

```
public void setDescription(IDescription value)
public void clearDescription()
public IDescription getDescription()
public boolean hasDescription(IDescription
descriptionToCheck)
public void addComponent(IComponent
newComponentInstance)
public void addComponent(Collection components)
public void clearComponents()
public void removeComponent(IComponent
componentToRemove)
public void removeComponents(Collection components)
public IComponent getComponent(String id)
public Collection getComponents(Collection ids)
public Collection getAllComponents()
public boolean hasComponent(IComponent
componentToCheck)
public Collection hasComponents(Collection
componentsToCheck)
public boolean hasAllComponents(Collection
componentsToCheck)
```

Because the description has cardinality 1, apigen generates functions for setting, clearing, and retrieving a single description. Since the components have cardinality >1, apigen generates a larger set of functions that deal with a collection of components.

XML document, using the DOM interface to create XML instance documents valid to a particular set of schemas is difficult. DOM users must write an extensive amount of code to, for example, handle XML namespaces, add appropriate type declarations to elements, and maintain proper element sequences (when sequences are specified in the schemas). Apigen-generated classes hide all this detail, effectively insulating tool builders from these XML peculiarities.

4. ISSUES ENCOUNTERED

While developing apigen, we encountered a number of issues, both essential and accidental, related to generating data bindings based on XML schemas. These are detailed in this section.

4.1 Handling Unknown Types

Perhaps the most important and essential difficulty we discovered while developing apigen is the inherent difficulty in properly handling unknown XML schema types in instance documents. Ideally, tools should treat an unknown type as its closest known ancestor type (if any). This gives the tools maximal information about the structure of the unknown type. This problem seems relatively easy to solve, but the type inheritance mechanism of XML schemas introduces some subtle and pervasive problems.

Consider an XML schema element E that contains an element of type T. In the same XML schema, a type T_a is defined that extends type T. Assume that, in an unknown schema, the type T_a has been further extended into a type T_b. In an XML instance document, elements of type T_a and T_b can both be substituted for the element of type T in E. Certainly, without knowledge of the XML schema defining T_b, it is impossible to process whatever extensions were made in T_b. However, it is desirable to at least process elements of type T_b as if they were instances of T_b's known base type, T_a. This would give programs access to the subset of things in T_b defined in T_a.

When a program like an apigen-generated library encounters an XML instance document that contains an element of type T_b, it encounters a problem. Because type T_b is unknown, it is impossible to infer anything about its ancestors in the type hierarchy. The only thing that is known for sure is that T_b is a type derived from T, because an element of type T_b was used in place of an element of type T. Without additional knowledge of T_b, it is impossible to determine whether T_b is a type derived from T_a, a different type derived from T, or a type defined even deeper in the type hierarchy. This issue cannot be completely resolved within the current XML standard because an element in an XML document refers only to the single type of which it is an instance. The derivation of this type is never defined in the XML document, and can only be divined with full knowledge of all the schemas in which the type and all its ancestor types are defined.

This difficulty can impede interoperability between tools built to deal with different sets of xArch XML extensions. There are several approaches to work around this problem. In the case of apigen, the optimal solution is to acquire the schema that contains the unknown type and use apigen to generate libraries for that schema. However, this is not always possible or desirable. Without the schema, one approach to handle the unknown type is to guess its derivation heuristically by looking at element contents. Unfortunately, this is extremely unreliable, and the accuracy of the approach varies depending on the (mostly arbitrary) contents of the elements in question. Apigen-generated libraries work around this problem by treating unknown derived types as instances of their most basic base type, as this is the most that can reliably inferred about the derived type. In the above example, the apigen-generated libraries would treat the element of type T_b as one of type T, since that is all that can be reliably inferred about T_b.

4.2 Type Derivation by Restriction

Unlike most object-oriented programming languages, XML schemas allow type derivation by restriction. That is, a derived type can alter the data model of its base type. There are limits on what kinds of restrictions are allowed. Elements from the base type may not be removed entirely in the derived type, but their cardinality can change (an exception to the no-removals rule occurs when the cardinality of an optional element changes from at-most-one to at-most-zero). Apigen-generated libraries assume constant cardinality of elements from base type to derived type because elements with cardinality 0-1 are manipulated by functions that expect a single value, whereas elements with cardinality >1 are manipulated using Java collections. As such, apigen does not currently support type derivation by restriction, though this could be added with moderate effort if the need arises.

4.3 XML Schema Lacks Multiple Inheritance

XML schemas permit only single inheritance among XML schema types. This causes problems in the definition of a modularly-extensible language like xArch. Two conceptually orthogonal extensions to the same element cannot coexist in the same instance document unless one extension is made to be dependent on another. Henry Thompson, one of the editors of the XML schema standard, has stated that the XML schema working group acknowledges this problem and is considering adding multiple inheritance support in the future [7]. Admittedly, 'mixins,' in the form of substitution groups, are allowed in XML schemas, which can simulate some aspects of multiple inheritance; however, these are not a substitute for true multiple inheritance. To work around this problem in developing xArch, we introduced artificial dependencies among some of our extensions. As such, some extensions are syntactically dependent on other extensions despite the fact that they are semantically unrelated. We intend to remove these artificial dependencies when and if multiple inheritance is supported in XML schemas.

Ironically, while the lack of support for multiple inheritance in XML schemas complicated our schema development, it greatly simplified our development of apigen. Apigen generates Java libraries, and Java also lacks support for multiple inheritance. As such, there was a natural mapping between the type inheritance in our XML schemas and the object-oriented inheritance in the apigen-generated Java libraries.

The future change from a single-inheritance model to a multiple-inheritance model could be problematic. In a language like C++ that supports multiple inheritance, a natural mapping between XML type multiple inheritance and object-oriented multiple inheritance is relatively easy to maintain. However, in a language like Java, with its support only for single inheritance, the problem is compounded. We have anticipated this problem in our development of apigen. This is part of the reason why apigen generates both a Java implementation *and* interface for each XML schema type. Java allows an object to implement multiple interfaces. We believe it will be possible for a future version of apigen to generate objects that implement multiple interfaces as needed. Admittedly, this has its own drawbacks, but may be the only feasible solution in a programming language without support for multiple inheritance.

4.4 Immature XML Tools

At the time apigen was developed, many XML tool vendors and developers were struggling to keep up with the rapid evolution of various XML standards and technologies. In fact, our development of apigen was only undertaken after a search for off-the-shelf XML schema-aware data binding generators failed to produce any promising results. Even Apache Xerces [1], the parser on which apigen and the apigen-generated libraries are built, has many small peculiarities and bugs (mostly namespace-related) that can inhibit the correct production and processing of XML documents.

While this is a mostly accidental problem that will almost

assuredly be resolved over time, it reveals a good reason to use a tool like apigen to generate XML data bindings. Workarounds for peculiarities and bugs in DOM implementations, parsers and other tools are implemented directly in apigen and the apigen-generated libraries. When a bug in the underlying tool is fixed, apigen's workarounds can be removed and its Java libraries regenerated without changing the libraries' interfaces. Because apigen-generated libraries hide most of the intricacies of XML from tool developers, tools can be made more reliable and more focused on the task at hand.

4.5 Limitations of Apigen

While apigen-generated libraries can hide the nuances of XML from tool developers, the structure of the object model in the libraries still mirrors the structure of an XML instance document. Namely, the objects are arranged hierarchically. Hierarchical constructions are optimal for hierarchical data, but complex relationships between elements cannot be easily modeled this way. We encountered this problem in building xArch because many views of software architectures resemble a general directed graph rather than a tree. To get around this problem in our schema development, we used XLinks [6] extensively to establish links from one part of the XML-based hierarchy to another.

This problem cannot be solved by an automated tool like apigen, because solving it requires semantic knowledge of the model being stored in XML. There is no reasonable way to store such semantic information in an XML schema or instance document. As such, we plan to develop a set of *convenience libraries* that expose a friendlier, non-hierarchical object model. These convenience libraries will work alongside the apigen-generated libraries, and will call them extensively. Because of the amount of semantic information inherent in these convenience libraries, we believe that they will necessarily be maintained by hand.

5. CONCLUSION

XML schemas provide a significant opportunity to create extensible languages, with the tool support of XML, that are useful in many domains. We used XML schemas in the creation of xArch, a modularly extensible language for representing software architectures. To support the development of tools that manipulate xArch representations, we built apigen, a schema-aware data binding generator that works on xArch schemas, which are representative of a large class of XML schemas.

This paper outlines experiences we had while building apigen. In doing so, we encountered and overcame issues that touched upon the nature of XML schemas, the difficulties of mapping XML schema types to programming languages, and problems with the state-of-the-art in XML tools. These problems ranged from essential (the inability to properly deal with unknown derived types in XML instance documents) to the mundane (bugs in our XML parser and DOM implementation).

The development of apigen has not only highlighted these important issues and some potential solutions, but it has also served as a proof-of-concept. Apigen-generated libraries successfully hide the peculiarities of XML and XML namespaces from the user, allowing tool builders to concentrate only on manipulating the modeled data structure (in xArch's case, a software architecture description).

Apigen-generated libraries are unable to hide the hierarchical structure of XML data from the user, as doing so would require semantic information about the data being modeled. However, we believe libraries can be built on top of apigen-generated libraries that can provide this additional level of abstraction.

6. URL

More information on apigen and xArch can be found at:

http://www.isr.uci.edu/projects/xarchuci/

7. ACKNOWLEDGEMENTS

I would like to thank André van der Hoek at UC Irvine and David Garlan & Bradley Schmerl at Carnegie Mellon University for their collaboration in the development of xArch. Furthermore, I would like to thank André van der Hoek, Yuzo Kanomata, Craig Snider, and Richard N. Taylor for their collaboration, feedback and guidance in developing apigen and the apigen-generated xArch libraries. Finally, I would like to thank the anonymous reviewers, who helped me shape the paper and correct several technical inaccuracies.

This effort was partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

8. REFERENCES

[1] Apache Group. Xerces Java Parser Readme. URL: http://xml.apache.org/xerces-j/index.html.

[2] Altheim, M., Boumphrey, F., Dooley, S., McCarron, S., Schnitzenbaumer, S., and Wugofski, T., eds. Modularization of XHTML. URL: http://www.w3.org/TR/xhtml-modularization/.

[3] Biron, P. and Malhotra, A., eds. XML Schema Part 2: Datatypes. URL: http://www.w3.org/TR/xmlschema-2/.

[4] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E. Extensible Markup Language (XML) 1.0 (Second Edition). URL: http://www.w3.org/TR/2000/REC-xml-20001006.

[5] Dashofy, E., Garlan, D., Schmerl, B. and van der Hoek, A., eds. xArch. URL: http://www.ics.uci.edu/pub/arch/xarch/.

[6] DeRose, S., Maler, E. and Orchard, D., eds. XML Linking Language (XLink) Version 1.0. URL: http://www.w3.org/TR/xlink.

[7] Dodds, L. Reconstructing DTD Best Practice. URL: http://www.xml.com/pub/a/2000/06/xmleurope/schemas.html.

[8] JDom.org. JDOM. URL: http://www.jdom.org/.

[9] Le Hors, A., ed. Document Object Model (DOM) Level 3 Core Specification. URL: http://www.w3.org/TR/2001/WD-DOM-Level-3-Core-20010126/

[10] N. Medidovic, R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*. 26(1):70-93. January, 2000.

[11] Megginson Technologies. SAX 2.0: The Simple API for XML. URL: http://www.megginson.com/SAX/.

[12] D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.

[13] Sall, K. and St. Laurent, S. DTDs vs. XML Schemas for Data-centric JavaTM Technology-based Applications. URL: http://www.cen.com/ng-html/xml/schema/.

[14] Thompson, H., Beech, D., Maloney, M. and Mendelsohn, N., eds. XML Schema Part 1: Structures. URL: http://www.w3.org/TR/xmlschema-1/.