A Concise XML Binding Framework Facilitates Practical Object-Oriented Document Engineering

Andruid Kerne, Zachary O. Toups, Blake Dworaczyk, Madhur Khandelwal
Interface Ecology Lab | Computer Science Department | Texas A&M University
{andruid, toupsz, blake}@ecologylab.net, madhurk@gmail.com

ABSTRACT
Semantic web researchers tend to assume that XML Schema and OWL-S are the correct means for representing the types, structure, and semantics of XML data used for documents and interchange between programs and services. These technologies separate information representation from implementation. The separation may seem like a benefit, because it is platform-agnostic. The problem is that the separation interferes with writing correct programs for practical document engineering, because it violates a primary principle of object-oriented programming: integration of data structures and algorithms. We develop an XML binding framework that connects Java object declarations with serialized XML representation. A basis of the framework is a metalanguage, embedded in Java object and field declarations, designed to be particularly concise, to facilitate the authoring and maintenance of programs that generate and manipulate XML documents. The framework serves as the foundation for a layered software architecture that includes *meta*-metadata descriptions for multimedia information extraction, modeling, and visualization; Lightweight Semantic Distributed Computing Services; interaction logging services; and a user studies framework.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – data types and structures, frameworks.

General Terms
Design, Human Factors, Languages.

Keywords
XML, Java, object-oriented programming, translation, binding framework, metalanguage.

1. INTRODUCTION
We need to discover alternative means for programmers to represent the semantics of serialized structured information. XML provides an excellent basis, but does not, in itself, provide the strongly typed data structures that are best for supporting programming in the large. According to its specification, XML Schema was designed to, "define and describe a class of XML documents by using schema components to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content and attributes and their values" [12]. The Schema language is developed from an information-centric perspective. This is a worthy approach. The problem is that in its design XML Schema does not seem to focus on the practical needs of software developers building and deploying applications.

The purpose of the open source `ecologylab.xml` information binding framework (http://ecologylab.net/xml), is to provide an environment optimized for practical XML document engineering. In our design and implementation, we have taken the perspective of the Java programmer's needs, because that is who we are. We have taken the object-oriented approach of using Java class declarations, augmented by an embedded annotation metalanguage, as the basis for defining XML document structures. We have focused the design of the metalanguage, so that declarations and resulting XML code are concise. This framework is being developed as the foundation of a software architecture of connected layers for practical semantics. The next layer, of Lightweight Semantic Distributed Computing Services (LSDCS), enables concise and practical transport of semantic declarations between processes, across the network, as well as the distributed performance of operations on such semantic structures [10]. A subsequent layer, of *meta*-metadata, provides a basis in XML for specifying the extraction of strongly-typed information structures from HTML and XML document templates, and means for creating interactive information visualization applications with these structures [2]. While our initial implementations are in Java, emphasizing the immediate practical construction of complex working systems with many users, planned future work will port these layers of semantic infrastructure to support other development platforms.

It makes sense for programmers, rather than schema-driven code generators, to author metalanguage definitions, because the set of possible mappings from a schema to object declarations is one-to-many. The classes that result from automatic translation can sacrifice runtime efficiency and design clarity. Further, because authoring schemas is complex [6], they do not always exist [7]. Human involvement ensures that the XML-Java mappings match what is desired, and that they are efficient and easy-to-understand.

We begin with an example, using the `ecologylab.xml` framework to parse the RSS dialect of XML. Then, we present the annotation metalanguage for expressing relationships between Java objects and XML element structures. Next, we compare expressiveness and usability of `ecologylab.xml` to the JAXB framework. We describe semantic programming framework layers built on top of `ecologylab.xml`. We conclude by discussing advantages of describing information semantics with object-oriented languages.

2. EXAMPLE: READING/WRITING RSS
`ecologylab.xml` is designed to simplify writing code to read and manipulate XML from the wild, and also for authoring custom XML documents to represent program state. We develop an example of the syntax and semantics of translation by taking a sample of wild XML and developing `ecologylab.xml` code to represent corresponding Java data structures. With these data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DocEng'08, September 16–19, 2008, São Paulo, Brazil.
Copyright 2008 ACM 978-1-60558-081-4/08/09...$5.00.

structures, one can read any XML document of the same dialect into an application, manipulate the information contained within through program objects, and translate these it back into XML.

A popular technology news feed (Figure 1) is published with the Really Simple Syndication (RSS) dialect of XML [7]. Note that despite the popularity of RSS, it is not a true standard: official formal schemas do not exist, only human-readable specifications. We present Java classes annotated with metalanguage corresponding to a subset of RSS. A more complete implementation supporting all varied syntaxes of RSS is provided in `ecologylab.xml.library.rss`. Figure 2 presents annotated code with mappings to the example RSS data.

To translate RSS to Java, we define a Java class for each non-scalar element used in RSS XML. Fields in each such class correspond to attributes and nested elements. Metalanguage constructs annotate the declaration of each Java field that is to be translated to and from XML.

We first declare a top-level class for the `rss` root element, `Rss`. The `ecologylab.xml.ElementState` class building block provides methods for XML translation; subclasses function as program objects that map to XML constructs. The **`Rss`** subclass is declared with fields that correspond to the **`rss`** root element's single attribute, *`version`*, and its nested element, **`channel`**. The declarations for these fields are annotated [8] with metalanguage, embedding specification of translation semantics in the code: *`version`*, a `float`, is declared with `@xml_attribute`; **`channel`** is annotated with `@xml_nested` in order to specify that this field is represents a complex, non-scalar type of XML element, declared as another `ElementState` subclass, `Channel`.

Each `channel` may contain an arbitrary number of **`item`** sub-elements, which requires representing a one-to-many relationship. The `@xml_collection` metalanguage construct declares a field that adds nested elements into an object that implements `java.util.Collection`, such as `ArrayList` [9]. The "**`item`**" argument specifies the tag name for these elements in XML. An instantiated generic type variable is used for declaring, in Java, the type of the objects in the `Collection`. The `ecologylab.xml` framework utilizes this type declaration (e.g., `ArrayList<Item>`) as the basis for constructing child objects into which the associated XML is translated.

The `Item` subclass of `ElementState`, to which `Channel` refers, is the most useful part of the feed for programs such as news readers.

```
<rss version="2.0">
 <channel>
  <title>Ars Technica</title>
  <link>http://arstechnica.com/index.ars</link>
  <description></description>
  <item>
   <title>AT&T surprises with beachfront…</title>
   <link>http://feeds.arstechnica.com/~r/arstechnica/BAaf/
       ~3/167531099/20071009-att-…</link>
   <description>In 2001 and…</description>
  </item>
  …
 </channel>
</rss>
```

Figure 1. RSS feed example from the Ars Technica news service [1], showing a single story.

Like *`version`* in **`Rss`**, the **`title`**, **`link`**, and **`description`** fields correspond to scalar type data. However, in XML, they are represented using *elements*, instead of attributes. To represent XML in which an attribute-less element with a single text node child is used to represent a single scalar value, we introduce the `@xml_leaf` metalanguage construct. While the **`title`** and **`description`** fields are of type `String`, the `URL` declaration of the **`link`** field, like the *`version`* attribute above, exemplifies automatic marshalling and unmarshalling, with type conversion, of scalar types.

To perform translation from XML into Java, we must define a `TranslationScope` that specifies which Java classes can function as targets of the translation. Translation scopes constitute a formal type system, encapsulating sets of Java classes for use in unmarshalling XML.

```
Rss ars = (Rss) ElementState.translateFromXML(
   "http://feeds.arstechnica.com/arstechnica/BAaf",
   TranslationScope.get("rss_scope", Rss.class,
     Channel,class, Item.class));
```

This statement produces an `Rss` object populated by the data in the RSS feed of Ars Technica. Conversely, to output RSS from this `Rss` object, for example to the console, we call:

```
ars.translateToXML(System.out);
```

3. ANNOTATION METALANGUAGE

Metalanguage declarations work with translation scopes to specify how a tree of loosely typed XML nodes is translated to and from a strongly typed tree of Java objects and fields. Metalanguage takes the form of Java annotations [8] processed at runtime, enabling the `ecologylab.xml` framework to determine how to translate classes

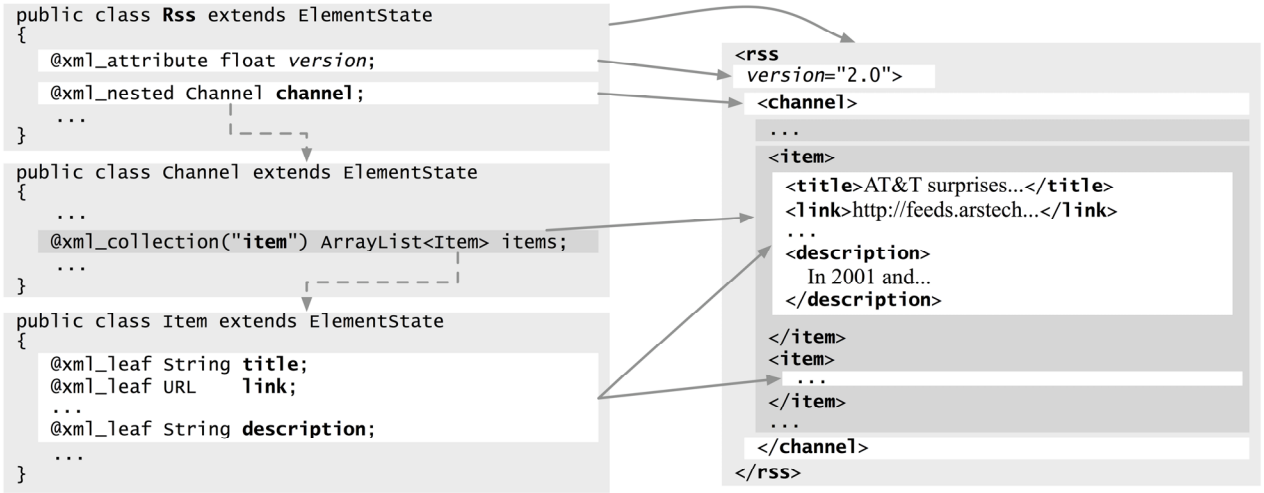

Figure 2. Direct mappings between Java code and example RSS XML.

into XML elements and fields into XML attributes and elements. The structural co-incidence of metalanguage declarations with field declarations facilitates readability and program maintenance. They serve both as program semantics and as documentation, supporting object-oriented software engineering principles.

This section addresses principal `ecologylab.xml` metalanguage constructs for automatically translating XML nodes into Java objects, fields, and values. We begin with mechanisms for representing scalar-valued simple types. Next, we move up in complexity to nested values that take the form of rooted sub-trees corresponding to a single type. Finally, we address collections of typed values, including hash tables. For both single nested values and collections, we introduce support for polymorphism.

Scalar values are stored in XML as attributes (e.g. `version` in the RSS example), or as *leaf nodes* (e.g. `link`), that is, as the single text node child of an element. Scalar values can be directly stored in Java object fields annotated with the `@xml_attribute` or `@xml_leaf` metalanguage constructs. Type conversion, that is un/marshalling, is performed by an extensible scalar type system. The current release provides support for all primitive types, and others, including `String`, `URL`, `Color`, and `Date`.

Nested non-scalar (complex typed) elements of an XML document can be composed of attributes, leaf nodes, text, and, recursively, other nested non-scalar elements. These correspond to Java objects, each with any number of fields, including scalars and other non-scalar reference types. To specify a one-to-one mapping between a non-scalar nested XML element and an instance of a strongly typed Java class, use the `@xml_nested` metalanguage construct. A field annotated with `@xml_nested` must be declared as a subclass of `ElementState`, meaning that it, in turn, has further annotated fields which bind to XML. In general, fields declared with `@xml_nested` bind field name to XML tag (with camel case conversion, unless overridden). To support polymorphism, the `@xml_classes` annotation informs the framework to bind one of various polymorphic instances of a super type to a single field. To guide translation, this declaration takes an array of `Class` literals, each of which can be instantiated when specified in the XML, as its argument, and then uses the names of these classes, instead of the field name, as the basis for the corresponding XML element tag name specifications.

Some XML nodes have one-to-many relationships with a variable number of children, each of a common type. Java collection objects [9], such as `ArrayList`, likewise contain zero or more objects, whose type can be constrained by the value of a generic type variable. RSS feeds, for example, may contain multiple `item` elements inside the `channel` element (Section 2) [7]. The `@xml_collection` metalanguage declaration specifies a one-to-many mapping of child objects to a parent, with sequential access to collection members. When all collection elements are of exactly the same type, a single argument to the construct indicates the tag used for the child elements, while the instantiated generic type variable in the declaration specifies their type. Alternatively, to declare a polymorphic collection of elements of different types and a common super type, `@xml_classes` may be used with `@xml_collection`. In this case, `ecologylab.xml` will translate each sub-element using a tag-class mapping.

Because it is often necessary to quickly and randomly access the contents of collections by a key, rather than an ordinal index, we provide support for automatically generating hashed data structures from sets of XML elements. The `Map` interface, which is implemented, for example, by `HashMap`, declares a collection of values, each of which is retrieved using a key [9]. When transforming XML into Java, one often wants to create such a hashed collection of elements by using one of each element's scalar-valued fields to form its key. Like `@xml_collection`, the `@xml_map` metalanguage annotation specifies a one-to-many relationship, automatically instantiating a `Map` data structure from the XML. The type for the objects declared as values in the corresponding generic declaration for the `Map` must implement the provided `Mappable` interface.

4. COMPARISON TO JAXB

JAXB 2.0 is a popular Java-XML data binding framework that also includes an annotation metalanguage. It is significantly more complex and cumbersome to write when compared to `ecologylab.xml` and produces more verbose XML. JAXB can generate annotated Java classes from XML schemas [5]. Automatic code generation is not a panacea, however, as many dialects of XML, such as RSS, lack schemas, so a human must specify correct translation between XML and program objects. A key step in writing correct and efficient code is the definition of optimal internal data structures to represent collections. Unlike JAXB, `ecologylab.xml` enables directly generating rich data structures such as hash tables, promoting efficiency, software development, and maintenance.

The vocabulary for JAXB is large and complex, including 30 declarations, which are verbose and redundant, requiring the repetition of field and class names (Figure 3). `ecologylab.xml`'s vocabulary is concise, with a total of 10 constructs. This simple set is optimized for programmer convenience, while maximizing expressivity. Field and class names automatically map to element and attribute names, without a need to redundantly specify them. It is possible to customize these mappings. `ecologylab.xml` further supports backward compatibility through the metalanguage, as a set of element names may be mapped, one-way, to a field, so that existing XML with deprecated mappings may be read, but written out in a different, newer, way.

```
public class Rss extends ElementState {
 @xml_attribute          float           version;
 @xml_nested             Channel         channel;
…}
public class Channel extends ElementState {
 @xml_collection("item") ArrayList<Item> items;
…}
public class Item extends ElementState {
 @xml_leaf               String          title;
 @xml_leaf               URL             link;
…}
```

```
@XmlRootElement(name="rss") @XmlType(name="Rss")
public class Rss {
 @XmlAttribute             float          version;
 @XmlElement               Channel        channel;
…}
@XmlType(name="Channel") public class Channel {
 @XmlElement(name="item")  ArrayList<Item> items;
…}
@XmlType(name="Item") public class Item {
 @XmlElement(name="title") String         title;
 @XmlElement(name="link")  URL            link;
…}
```

Figure 3. Contrasting code density of `ecologylab.xml` (left) and JAXB (right) in RSS example (metalanguage constructs in italics). Note the redundancy of the JAXB annotations and their proliferation. By comparison, specifications with `ecologylab.xml` are concise.

```
<questions>
 <short_answer_question …/>
 <essay_question …/>
 <multiple_choice_question …/>
</questions>

@xml_nested @xml_classes({ShortAnswerQuestion.class,
 EssayQuestion.class, MultipleChoiceQuestion.class})
 ArrayListState<Question>   questions;
```

```
<questions>
  <question
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:type="ShortAnswerQuestion" …/>
  <question
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:type="EssayQuestion" …/>
  <question
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance" xsi:type="MultipleChoiceQuestion" …/>
</questions>
@XmlElement @XmlElementWrapper(name="questions")
ArrayList<Question>  questions;
```

Figure 4. Comparison of XML produced by `ecologylab.xml` (left) and JAXB 2.0 (right) for polymorphism. JAXB repeats declarations in each XML element, creating extremely verbose XML. With `ecologylab.xml` the declarations of possible classes are part of the metalanguage, so they do not need to be repeated. This also enables using polymorphism when reading XML from the wild.

In practice, the JAXB metalanguage is limited in its support for writing correct and efficient programs because it does not eable directly creating `Map`-based constructs, where a list of collection elements is automatically loaded into a randomly accessible, hashed data structure. The `@xml_map` directive of `ecologylab.xml` automatically populates such `Map` structures from XML. In JAXB, such constructs must be created by placing a collection of elements into an intermediate data structure, and then creating the `Map` through hand coding. This is inefficient first for the programmer and then for the CPU and memory.

Polymorphic instances in JAXB result in verbose XML. `ecologylab.xml` handles polymorphism through the `@xml_classes` construct (Section 3). JAXB supports polymorphic subclass instances, but only uses a single tag name. To disambiguate the *class*, it must redundantly embed the polymorphic type in each XML element (see Figure 4). This has limited application, as XML from the wild that uses different tags for elements of different types does not contain the extra attributes.

5. LAYERED FRAMEWORKS

`ecologylab.xml` proves to be an excellent foundation layer for the development of higher level semantic frameworks. *Meta*-metadata layers on `ecologylab.xml` to produce a framework for specifying structures of multimedia semantics associated with particular document sources, to support representation, extraction, modeling, visualization, and interaction [2]. Lightweight Semantic Distributed Computing Services (LSDCS) layer over `ecologylab.xml` to provide the transport of semantic data through sockets, and the distributed invocation of operations on the data [10]. LSDCS are more concise and easier to develop and invoke than other approaches, such as SOAP. Interaction Logging Services (ILS) layer on LSDCS to provide facilities for application developers to gather semantic information about state and user actions, across the network. The `ecologylab.studies` framework layers on `ecologylab.xml` and ILS to develop a component-based Java web application for building user studies that administer complex questionnaires and launch and gather data directly from instrumented Java applications. Experimenters prepare studies by authoring an XML document, specifying questions, application preferences that correspond to different experimental conditions, and paths through the questions and conditions for automatic counter-balancing.

6. CONCLUSION

We develop a novel approach to specifying the meaning, usage and relationships of the constituent parts of XML documents: datatypes, elements and their content, attributes, and their values. Where XML Schema separates data definition from implementation, the present approach to XML document definition unifies them through its basis in an object-oriented programming language. While Schema has the virtue of being platform-independent, `ecologylab.xml` is oriented toward meeting the needs of software developers engaged in building practical systems for document engineering. Thus, schema-equivalent specifications are defined through annotated program object declarations. This approach enables more object-oriented software development by increasing the integration of data structures and the algorithms that manipulate them. The metalanguage has been designed to be especially concise, in order to facilitate software and document engineering. The resulting code is easier to write and easier to understand. Future work will demonstrate its runtime efficiency. The use of Java makes the present implementation platform-independent at the level of operating systems. Future work will extend the framework to support other programming languages.

7. REFERENCES

[1] Ars Technica. http://feeds.arstechnica.com/arstechnica/BAaf.

[2] Damaraju, S., Bandaru, B.K., Kerne, A., *Meta*-Metadata: A Layer for Multimedia Metadata Definition, Extraction, and Representation, submitted to SAMT 2008.

[3] Gosling, J., Joy, B., Guy, S., Bracha, G. *The Java Language Specification, 3rd ed.* The Java Series. Prentice Hall, 2005.

[4] Interface Ecology Lab, `ecologylab.xml`. http://ecologylab.net/xml.

[5] jaxb: JAXB Reference Implementation. http://jaxb.dev.java.net.

[6] Kay, M.H. XML five years on: A review of the achievements so far and the challenges ahead. *Proc. DocEng 2003*, 29-31.

[7] RSS Advisory Board. RSS 2.0 Specification (version 2.0.9). http://www.rssboard.org/rss-specification. 2007.

[8] Sun. Annotations. http://java.sun.com/j2se/ 1.5.0/docs/guide/language/annotations.html, 2004.

[9] Sun. Collections Framework. http://java.sun.com/j2se/1.5.0/docs/ guide/collections/overview.html, 2004.

[10] Toups, Z.O., Kerne, A. A Framework for Rapid Development of Composable Little Semantic Web Services. sent to *ISWC* 2008.

[11] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition). http://www.w3.org/TR/REC-xml, 2006.

[12] W3C. XML Schema Part 1: Structures, http://www.w3.org/TR/ 2004/REC-xmlschema-1-20041028/structures.html, 2004