Interface Ecology Lab Technical Report 08-06

Expressive, Efficient, Embedded, and Component-based XML-Java Data Binding Framework

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

Writing efficient, correct programs to process complex XML documents can be time-consuming and difficult. While data binding frameworks automate translating untyped, structured XML data into strongly typed data structures, their overhead, in terms of configuration during development and maintenance, and CPU at runtime, may be prohibitive. We present an open source XML-Java data binding framework to optimize the efficiency and expressive power of programmers and their applications. An annotation metalanguage embedded in Java field declarations enables fine-grained expression of the semantics of bi-directional XML-Java translation. The tight integration of information and program semantics is symbiotic, promoting software maintenance. The programmer can choose the most appropriate and efficient data structures to represent a particular dialect of XML, based on application needs. For example, collections can be automatically bound to hash tables with a single metalanguage declaration. A formal type system of *translation scopes* encapsulates sets of Java classes for binding XML elements. Translation scopes can be organized in correspondence with software components and composed through multiple-inheritance, maximizing reuse. The metalanguage is efficiently interpreted at runtime, avoiding pre-processors, through a parse tree of optimization structures. Performance and usability exceed that of other frameworks.

Categories and Subject Descriptors

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

General Terms

Algorithms, Performance, Design, Human Factors, Languages.

Keywords

XML, Java, translation, binding framework, metalanguage.

1. INTRODUCTION

XML is widely used for representing structured documents and inter-application communication. Writing efficient, correct programs to process and author complex XML documents can be time-consuming and difficult. Subsequent to the operation of an XML parser, values must be converted from strings into typed scalar values, complex nested objects, and collections. Data binding frameworks automate translating untyped, structured XML data to strongly typed data structures. However, the overhead of such frameworks, in terms of configuration during development and maintenance, and CPU at runtime, may be prohibitive. We need efficient data binding frameworks that facilitate software maintenance and are sufficiently expressive to enable programmers to specify powerful and efficient data structures that map automatically to the XML of their task contexts.

We present an open source XML-Java data binding framework, `ecologylab.xml` (http://ecologylab.net/xml), to optimize the efficiency and expressive power of programmers and their applications working with XML documents. A succinct annotation metalanguage embedded in Java field declarations enables fine-grained expression of the semantics of bi-directional XML-Java translation, so that the structure of information (fields), its behaviors (methods), and its documentation (comments) are integrated in a single document. The programmer can choose appropriate and efficient data structures to represent a particular dialect of XML, based on application needs. A formal type system of translation scopes encapsulates sets of Java classes for binding XML elements. Translation scopes can be organized into components and composed through multiple-inheritance, maximizing reuse. The metalanguage is efficiently interpreted at runtime, avoiding pre-processors. Performance and usability exceed that of other frameworks.

The present framework is constructed from an imperative-programming-language-centric view, privileging the expressive power of imperative declarations, enhanced by clear metalanguage annotations that direct XML binding. 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 [10], they do not always exist [13]. A human touch can ensure that the XML-Java mappings match what is desired, and that they are efficient and easy-to-understand.

We begin by presenting an example, which uses the `ecologylab.xml` framework for parsing the popular RSS dialect of XML. Then, we describe the type system of translation scopes, and how they are utilized in XML-to-Java translation. Next, we use a BNF-style grammar to present details of the annotation metalanguage for expressing relationships between Java object and XML element structures. Briefly, we describe optimizations that enable the framework's high performance. Next, we examine other XML-Java binding frameworks, comparing expressiveness, usability, performance, and runtime size. We subsequently describe a series of applications that make use of `ecologylab.xml`, including a Lightweight Semantic Distributed Computing Services (LSDCS) framework. We conclude by discussing advantages of describing information semantics with imperative languages, rather than XML schemas.

```
<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 700MHz spectrum purchase</title>
   <link>http://feeds.arstechnica.com/~r/arstechnica/BAaf/~3/167531099/20071009-att-…</link>
   <guid isPermaLink="false">http://arstechnica.com/news.ars/post/20071009-att-…</guid>
   <pubDate>Tue, 09 Oct 2007 17:29:00 GMT</pubDate>
   <author>----@arstechnica.com (Eric ----)</author>
   <description>In 2001 and 2003, the FCC auctioned off a total of 12MHz of spectrum…</description>
  </item>
  <item>…</item>
  …
 </channel>
</rss>
```

Figure 1. RSS feed example from the Ars Technica news service [2], showing a single story.

2. RSS EXAMPLE

`ecologylab.xml` is used to author custom XML documents from program objects and to process XML from the wild. 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 structures, one can read any matching XML document into an application, manipulate the data contained within it through Java 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 [13]. 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 complete implementation that supports all the varied syntaxes of RSS versions 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 must 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 [19] with metalanguage, embedding specification of translation semantics in the code: *`version`*, a `float`, is declared with the `@xml_attribute` annotation; **`channel`** is annotated with `@xml_nested` in order to specify that this field is used to represent 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` [20]. 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` [21]. 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 ElementState` subclass, to which `Channel` refers, is the most useful part of the feed for programs such as news readers.

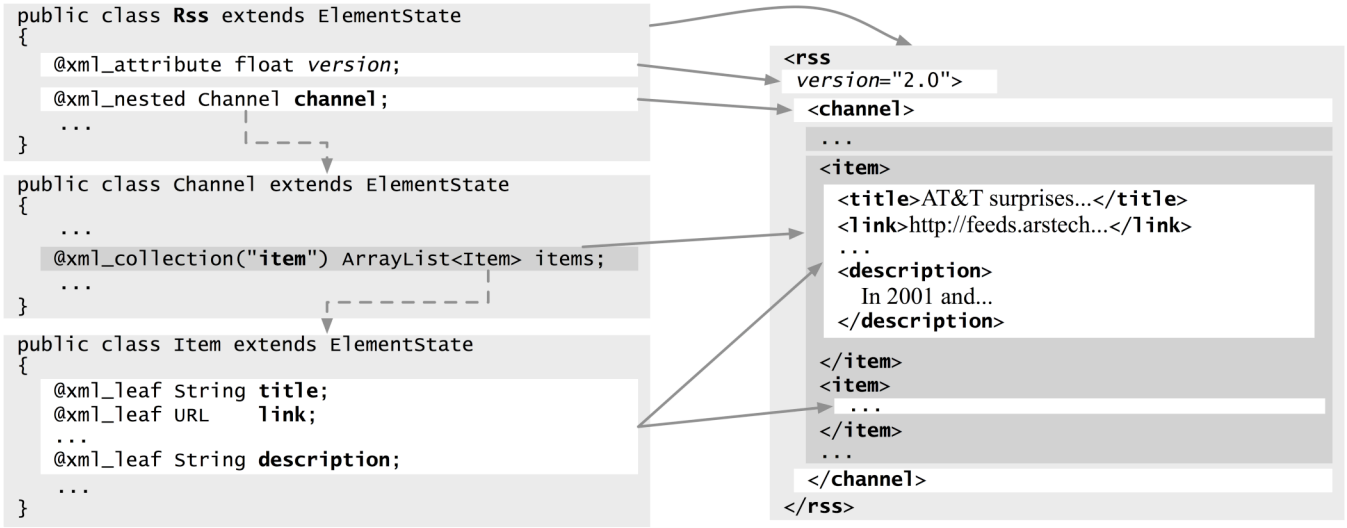

Figure 2. Direct mappings between Java code and example RSS XML.

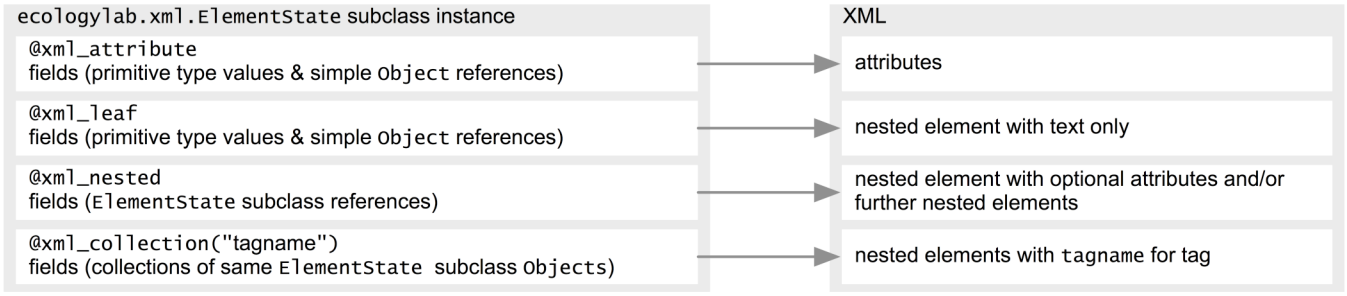

Figure 3. Field annotation declarations in Java classes are mapped to XML node structures.

Each `item` refers to a news 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` (Section 3) 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 to the console, we call:

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

3. TRANSLATION SCOPES TYPE SYSTEM

We introduce a lexically scoped [16] type system for binding XML tags to Java classes. Lexical scopes enable bundling sets of tag-class bindings that correspond to a software component. Developers can compose these sets of bindings to build complex programs and services [26] from modular components. Each of these sets of bindings is a *translation scope*. The type system allows composition of any number of translation scopes, supporting multiple inheritance. The data structure is optimized for lookups either by tag or by class name. A global scope maps the translation spaces, themselves, to names, for efficiency and convenience.

Translation from Java to XML is straightforward because, since Java is strongly typed, the types of XML elements and attributes are unambiguously determined from class definitions. However, to translate from an XML document into the corresponding tree of Java objects without a schema, we must map XML nodes (elements and attributes) to Java types.

A static accessor/factory method is used for forming translation scopes, instead of the constructor, so that the returned value can be registered for reuse:

```
TranslationScope.get(String name, Class[] translations,
  String defaultPackageName, TranslationScope…
  inherited)
```

Repeated calls with the same `name` will not construct a new `TranslationScope`, but instead retrieve the previously constructed instance. This allows programmers to avoid concerns about inefficiencies associated with potential re-instantiations, without having to coordinate initialization sequencing control flows across modules. The suggested practice is to create a static class for each software module, with a static method encapsulating the `get(…)` call for a particular set of translations.

The `translations` parameter defines a set of `Class` literals with which to form the translation scope. Specifying `Class` literals promotes software maintenance because references to them are maintained automatically through refactoring, as in Eclipse [4].

The `defaultPackageName` parameter instructs the framework where to look for an appropriately named class if one is not otherwise located.

The `inherited` parameter enables each translation scope to inherit from one or more others. A set of translations can be specified corresponding to a particular software component, such as those for RSS in the example ("`rss_scope`", Section 2), while another set of translations can be defined corresponding to a messaging protocol that embeds RSS. Multiple inheritance is supported. If the sets of mappings overlap, where there is a conflict with two identical unqualified class names but different class definitions, a previous mapping will be shadowed.

Shadowing in translation scope inheritance is useful, for example, when public and private versions of an API must be supported. The public version defines `ElementState` subclasses with all the data slots of the API, but no implementation functionality. The private version specifies a set of subclasses vis-à-vis the public ones. The subclasses will have the same unqualified names and tag names, but live in different packages. Good programming practice suggests that the public and private classes be defined in separate Eclipse projects to enforce data separation. In this way, the private classes are defined with whatever complex functionality and referentiality are necessary. The public class definitions are defined with minimal code that specifies data structure semantics without exposing software internals.

4. DATA BINDING ALGORITHM

Figure 4 presents an overview of the data binding algorithm used for translating from XML to Java. Tag names must be resolved into class *and* field names. Instances are created and field values set. Elements are either *scalar* valued, as for attributes and leaf nodes, or *non-scalar* valued, comprising sets of root element or sub-element fields, each with a name and a type. When a tag is resolved into the appropriate class or field, this, in turn, determines its type, and how the string representation will be unmarshalled. The `Field` reflection object [25] is used to assign this value into the instantiated tree of strongly typed objects, unless the programmer chooses to specify a `set…(…)` method, in

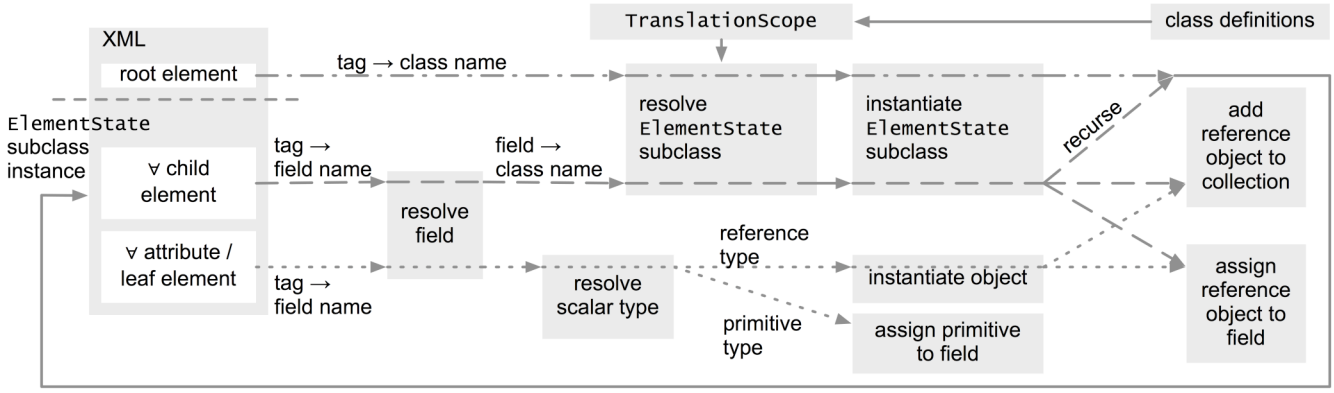

Figure 4. Translate from XML algorithm overview: binding data from an XML document to a strongly typed tree of Java objects.

which case the method overrides automatic unmarshalling and field assignment. We address the translation of non-scalar element types first, and then fill in the details for unmarshalling XML nodes that resolve to scalar types.

In the framework, the elements that are non-scalar will be mapped to subclasses of `ecologylab.xml.ElementState`. Scalar leaf nodes will be resolved to an appropriate scalar primitive or reference type. Attributes will always be resolved to scalars. These choices of representation are specified with the metalanguage (Section 5). Translation begins with the root element, which must be bound to an instance of the appropriate non-scalar `ElementState` subclass. Thereafter, during document tree descent, an appropriate field must be resolved for each node, in the scope of the current `ElementState` subclass instance (Figure 4). In many cases, XML tag names are bound to field names, but in some others, they are bound to class names. They also may be bound to a collection. These choices are determined by a combination of context, and the specific metalanguage construct specified by the programmer (see Section 5).

4.1 Non-Scalar Types

Each non-scalar valued element in an XML document tree will be resolved to a corresponding `ElementState` subclass, or ignored, per the programmer's specification. The process of translation begins with the root element. The Java object bound to the root element is not contextualized during translation as a field in another object, so the unqualified class name is derived from the tag name, with automatic conversion from underscore-delimited names to camel case (**`rss`** root element → **`Rss`** class instance). In the case of complex nested elements that correspond to individual fields, the process is similar, but the tag name is mapped to a field name, in the nested object currently in scope (**`channel`** element → **`channel`** field in an **`Rss`** instance).

The metalanguage can be used to make declarations that alter the tag-class mapping, using the `@xml_tag` metalanguage declaration (Section 5.3). This declares an alternative tag name to use as the binding for a class or field. For collection fields declared with `@xml_collection` or `@xml_map` (Section 5.2), with their one-to-many mappings, the tag name for elements that will be added to the collection, as with `@xml_tag`, can be declared as a metalanguage parameter, or, by default, the camel-case-converted class name can be used. As a third alternative, in these cases, when polymorphism is desired, the `@xml_classes` metalanguage declaration can be used to specify an appropriate set of subclasses. Again, the class declaration serves as the basis for the tag name (**`item`** element, child of **`channel`** → entry in **`items`** collection).

Which `ElementState` subclass will be used to instantiate objects that represent XML elements? To enable polymorphism, as with the class name derived from the root element's tag, the field objects are used to extract an unqualified class name; that is, one that lacks package specificity. Once this name is derived, it is used as the key for lookup in the translation scope (see above). If that lookup fails, then the field's fully qualified class is used.

4.2 Scalar Types

An extensible scalar type system supports unmarshalling and marshalling. For each scalar-valued field, the node name (attribute or leaf element) is used, again with automatic conversion to camel case, to form the field name. The class of the field is obtained from the Java Reflection `Field` object. The type system uses this class to derive a binding from the node to an appropriate scalar type, enabling handling primitives and simple reference types. For each such type, the `ScalarType` subclass is extended, defining methods for unmarshalling from `Strings`, marshalling to `Strings`, and the assignment of field values. The field value assignment method enables assignment to primitive typed fields without the overhead and inconvenience of boxing [6].

The programmer can use this mechanism to support any scalar-valued field type. The framework comes with `ScalarType` subclass definitions for all of Java's primitive types, as well as popular scalar reference types, such as `String`, `StringBuilder` (for efficiency), `File`, `Color` (marshaled with the HTML '#' notation), `Date`, and `URL` (or, alternatively, our convenient wrapper class, `ParsedURL`, which provides features such as lower case, domain identification, and suffix extraction by lazy evaluation).

4.3 Ignored Nodes

The framework enables elements and attributes to be ignored during translation. Nodes are ignored when, after camel case / `@xml_tag` conversion, there is no corresponding field with matching name and appropriate annotation. Thus, if a program does not need to assign the values of certain XML nodes to fields in Java objects, no resources are allocated for parsing and storing them, enabling more efficient translation. The first time an ignored node is encountered, a console warning will be issued, to aid debugging. Repeated encounters are silently ignored.

5. ANNOTATION METALANGUAGE

Metalanguage declarations work with translation scopes to specify how a tree of loosely typed XML nodes is bound to a strongly typed tree of Java objects, as well as how such object trees are marshaled to XML. Metalanguage takes the form of Java

annotations [19] that are directly processed at runtime, enabling the `ecologylab.xml` framework to determine how to translate classes into XML elements and fields into XML attributes and elements. No preprocessor is involved. The structural co-incidence of metalanguage declarations with field declarations facilitates readability and program maintenance.

In tandem with translation scopes (Section 3), metalanguage declarations specify unambiguous mappings for bidirectional translation between Java and XML. They serve both as program semantics and as documentation.

We present a grammar of the `ecologylab.xml` framework's metalanguage. This grammar (Figure 5) is an augmented subset of the BNF-style grammar for Java presented in *The Java Language Specification* [6], which formally specifies how the annotation metalanguage is used in the context of the Java language.

In this section, we cover the different types of annotations, indicating their role in the grammar's productions and explaining the mappings they define between Java code and corresponding XML. We first describe individual field annotations (*XMLIndividualAnnotation*), which map fields within a Java class to attributes and nested elements of an XML document in a one-to-one relationship. We then describe collection annotation declarations (*XMLCollectionAnnotation*), each of which specifies a one-to-many relationship between a Java field and nested elements. Then, we address constructs that enable further customization and explain how to limit the scope of translation.

5.1 Individual Field Annotations – *XMLIndividualAnnotation*

The *XMLIndividualAnnotation* production functions as a constituent of a field declaration (*FieldDecl* production, Figure 5), serving to declare a class field with a one-to-one relation to its data. The *Type* of the field will be declared as a non-scalar, multi-field object that extends `ElementState`, such as the **`Rss`** class (above), or as a scalar, such as an `int`, `String`, or `URL`.

Nested non-scalar elements (Section 4.1) within an XML document can be composed of attributes, leaf nodes, and, recursively, of other nested non-scalar elements. Similarly, Java objects may have any number of fields, including scalars and other non-scalar reference types. The `@xml_nested` metalanguage terminal symbol of *XMLIndividualAnnotation* declares a one-to-one mapping between a non-scalar nested XML element and a strongly typed Java class. A field annotated with `@xml_nested` must be declared with a *Type* that is 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). However, for polymorphism, `@xml_classes` can be used to bind one of various polymorphic instances of a super type to a single field. This declaration takes an array of `Class` literals as its argument, and then uses these classes as the basis for the corresponding tag names.

XML attributes and leaf nodes contain only scalar data, as more complex types would require their own set of delimiters within the data (obviating the purpose of XML) [27]. Java class fields that represent scalar data are thus annotated as `@xml_attribute` or `@xml_leaf`, indicating that, in the XML, the field's data is an attribute of a parent element, or a leaf node. Of course, the *Type* for such a declaration must correspond to a scalar. A subclass of `ecologylab.xml.types.scalar.ScalarType` must be previously

[x] denotes zero or one occurrences of x.
{x} denotes zero or more occurrences of x.
x | y means one of either x or y.

ClassBody:
{ {*ClassBodyDeclaration*} }

ClassBodyDeclaration:
;
| [`static`] *Block*
| {***OtherModifier***} *MemberDecl*

OtherModifier:
OtherAnnotation | `public` | `protected` | `private` | `static` | `abstract` | `final` | `native` | `synchronized` | `transient` | `volatile` | `strictfp`

MemberDecl:
| *GenericMethodOrConstructorDecl*
| ***MethodDecl***
| ***FieldDecl***
| `void` *Identifier MethodDeclaratorRest*
| *Identifier ConstructorDeclaratorRest*
| *ClassOrInterfaceDeclaration*

ClassOrInterfaceDeclaration:
ModifiersOpt (*ClassDeclaration* | *InterfaceDeclaration*)

ClassDeclaration:
`class` **`[@xml_inherit]`** **`[@xml_tag("`*`TagName`*`")]`** **[*XMLOtherTags*]** *Identifier* [`extends` *Type*] [`implements` *TypeList*] *ClassBody*

Type:
Identifier [*TypeArguments*]{ . *Identifier* [*TypeArguments*]} {[ ]}
| *BasicType*

TypeArguments:
< *TypeArgument* {, *TypeArgument*} >

TypeArgument:
Type
| ? [(`extends` | `super`) *Type*]

FieldDecl:
`[@xml_tag("`*`TagName`*`")]` **[*XMLOtherTags*]** **[*XMLClasses*]**
XMLIndividualAnnotation *Type Identifier MethodOrFieldRest*
| ***XMLCollectionAnnotation*** *Identifier MethodOrFieldRest*

XMLIndividualAnnotation:
`@xml_nested`
| **`@xml_attribute`**
| **`@xml_leaf([CDATA])`**
| **`@xml_text([CDATA])`**

XMLCollectionAnnotation:
`@xml_collection(["`*`TagName`*`"])` **[*XMLClasses*]** *Identifier TypeArguments*
| **`@xml_map(["`*`TagName`*`"])`** **[*XMLClasses*]** *Identifier TypeArguments*

XMLClasses:
`@xml_classes(`*`{ClassName`*`.class{, `*`ClassName`*`.class}}``)`

XMLOtherTags:
`@xml_other_tags(`*`{"TagName"{, "TagName"}}`*`)`

TagName:
any allowed XML tag name

ClassName:
name of any existing class

Figure 5. Grammar for `ecologylab.xml` annotation metalanguage. Productions that differ from the original Java specification are highlighted and in bold.

declared, such as `IntType` for `int` (see Section 4.2 and the API documentation [8]). `@xml_text` is similar to `@xml_leaf`, but for cases in which a scalar-valued text node child of an element is combined, in the XML, with scalar values.

XML leaf nodes (but not attributes) may also contain unescaped character data (CDATA), strings that may contain markup characters ("<", ">", etc.), and should be ignored by the parser [27]. CDATA is declared with the optional `CDATA` literal argument to the `@xml_leaf` and `@xml_text` metalanguage constructs.

5.2 Collections – *XMLCollectionAnnotation*

It is common for XML documents to contain parent nodes with one-to-many relationships with a variable number of children, each of which is of a common type. Java Collections likewise contain zero or more nested objects of the same type. RSS feeds, for example, may contain multiple `item` elements inside the `channel` element (Section 2) [13]. The *XMLCollectionAnnotation* production specifies such a one-to-many relationship between fields and their data. These constructs must be used with the Java Collections Framework, a set of classes that handle variable-sized groups of objects of the same class [20].

Within the *XMLCollectionAnnotation* production, the `@xml_collection` metalanguage declaration specifies a one-to-many mapping of child objects to a parent, with sequential access to collection members. The *Type* declaration for such a field can either be for a class that implements the `Collection<? extends ElementState>`[1] interface, or for `Collection<? extends ScalarType>`. In this latter case, the elements of the collection will be represented in the XML as leaf nodes.

`Collection`s declared with `@xml_collection` have two possible mechanisms for bi-directional translation, maximizing the flexibility of the construct. Instantiated generic type variables play a key role in both. In the first mechanism, a developer-specified tag name is used for translating to XML from Java, in place of automatic camel case conversion. For translating to Java from XML, `ecologylab.xml` uses the declared parameterized type for the `Collection`. To use this mechanism, the developer indicates a tag name through the *TagName* constituent of the `@xml_collection` expression. It is necessary to parameterize the generic `Collection`, so that its parameterized type is a subclass of `ElementState`. `ecologylab.xml` will use this parameter to instantiate the children of the collection.

With the other `@xml_collection` mechanism, the *TagName* parameter is omitted. Instead, the tag name of each child element in the XML is directly bound to an unqualified Java class name through camel case conversion, and then resolved through the translation scope (Section 2 and Figure 4). This creates a different flexibility, as now child elements may be polymorphic subclasses of the generic type parameter of the `Collection`. For example, if we had a class `Question`, a subclass of `ElementState`, and a class `EssayQuestion`, a subclass of `Question`, both of these could be stored in a field whose type is `Collection<Question>`. To accomplish such polymorphism, the `@xml_classes` directive is used to specify the alternative subclasses. For the example, the appropriate declaration is `@xml_classes({Question.class, EssayQuestion.class})`.

Because it is often necessary for collections to be quickly and randomly accessed, based on the data they contain, rather than an ordinal index, we provide support for automatically generating hashed data structures from sets of XML elements. `Map` is an interface in `java.util` for specifying a collection of values, each of which can be retrieved using a key [20]. The critical observation is that 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.

`@xml_map` is an extremely powerful, easy-to-use extension of the `@xml_collection` production. `@xml_map` can only be used with fields whose types implement the simple `Map<KEY, VALUE extends ElementState & Mappable<KEY>>`[2] interface. Because, in practice, keys are often computed from data in the `Map`'s values, we specify the `Mappable<KEY>` interface, which defines only a `KEY key()` method which the framework uses to establish hashed mapping relationships. `@xml_map` fields are translated to XML in the same way as `@xml_collection`, but when translated to Java, they are automatically loaded into the appropriate `Map` object with key-value relationships established. Thus, the single line declaration of the `Map` with the `@xml_map` metalanguage, and the simple implementation of the `key()` method are all the programmer must specify in order to automatically build a data structure such as a `HashMap` [20] from a set of XML elements, with no computational overhead of first loading the constituents into an intermediate collection, and no intermediate coding.

5.3 Custom Tags

The `@xml_tag("`*TagName*`")` construct (*FieldDecl*, and *ClassDeclaration* productions) allows the programmer to explicitly declare the tag name for a given field or class. This flexible mechanism provides maximum control over the output XML, and enables reading XML from the wild. `@xml_tag` overrides `ecologylab.xml`'s automatic camel case conversion of names. It handles names normally outside the scope of translation because the characters used are not allowed in Java fields, such as those including a dash ("-") or colon (":"), or names that collide with Java reserved words (such as "`abstract`"). When `@xml_tag` is used for a field declaration, as in the *FieldDecl* production, it remaps the XML node name for the field. When used as part of a class declaration, as in the *ClassDeclaration* production, `@xml_tag` overrides the name conversion for objects of the given class, when they are translated based on class name (as with the root element of a document, or the second `@xml_collection` mechanism).

`@xml_tag` also allows the programmer to compress custom XML as desired. Field names can be of arbitrary length (and readability), while the resulting XML (which may not be meant for human eyes) can be as small as possible. Compressing reduces verbosity, which can be a shortcoming of XML [14].

Support for compatibility with old versions of an XML dialect is provided by the `@xml_other_tags` directive. This directive takes one or more strings as its argument. It can be applied to either a class or field declaration. The result creates extra one-way

[1] `<? extends ElementState>` is the parameterized type of a generic object [21]. Here, we indicate a collection of objects which are subclasses of `ElementState`. It is unknown at compile time exactly which subclass these objects will be (hence, "`?…`").

[2] To be clear, `KEY` is the parameterized type of the keys of the `Map`, while `ElementState & Mappable<KEY>` is the parameterized type of the values to which the keys will map. Thus, a `Map` consists of a set of key-value pairs, whose types are specified above.

mappings from an XML tag to a class or field. These mappings are second class, in that they will only be used in translating *from* XML *to* Java. In the other direction, the primary tag, as specified with other metalanguage, will be used. Like `@xml_tag`, the interpretation of `@xml_other_tags` must modify entries in referencing `TranslationScopes`.

5.4 Limiting the Scope of Translation

Translation to and from XML is not computationally free, so it is useful for the programmer to be able to specify limits on its scope. The `@xml_inherit` construct (*ClassDeclaration* production), by its presence in a class's declaration, indicates that the fields of the class's superclass *should be translated*. This construct enables programmers to streamline the execution of their code. Superclasses that contain needed functionality but no marshalled fields can be omitted from translation.

6. JUST-IN-TIME OPTIMIZATIONS

The framework spares the programmer the inconvenient need to invoke a pre-processor as part of the code / debug / deploy cycle of program development. Yet runtime performance has not been sacrificed. Behind the scenes, the framework builds a parse tree of objects to optimize translation performance. The optimizations are created just-in-time, as needed through lazy evaluation, the first time that a Java class is involved in translation to- or from-XML. They are indexed, with hashing, enabling retrieval in near-constant time. Each class creates a set of node-to-Java optimizations for each tag that will be encountered in the scope of its translation from XML, indexing them by tag name. A similar set of field-to-XML optimizations is created for translation to XML. Each of these optimizations records essential information, such as the translation type of the element or field, the associated Java Reflection `Field` object, the `ElementState` subclass for nested objects, and the `ScalarType` subclass for scalar objects. Of particular note in this strategy is the observation that `Field` reflection objects, each of which denotes an instance variable declaration, are reusable across different instances of the same class. Thus, the cost of obtaining `Field` objects once, on demand by lazy evaluation, and then caching them, is minimal.

7. PERFORMANCE AND PRIOR WORK

We compare the performance and expressive power of `ecologylab.xml` with the best other frameworks. `ecologylab.xml` avoids external intermediate definition language declarations, like those of JiBX [18] and CORBA IDL [15]. We measured performance using the Bindmark XML binding benchmark, which compares Java-XML binding frameworks in marshalling and unmarshalling, with a range of XML file sizes [3] (Figure 6) `ecologylab.xml` outperforms every framework. The benchmark for `ecologylab.xml` is unbiased, containing no code specific to Bindmark. The performance of one framework, JiBX [18] seems comparable, but not only does JiBX require extra work by the programmer, it also includes initial invocation of a binding compiler, the runtime of which is mysteriously excluded from measurement in the Bindmark distribution.

The JiBX framework [18] offers XML-Java translation through the use of binding documents and byte-code modification. JiBX developers must create *XML binding definitions*, one or more XML documents that verbosely map classes and their fields to XML data structures (see Figure 7). Although this document enables flexible XML-Java mapping, it is not at all automatically maintained, leaving that job to the programmer. The binding definitions text must be updated whenever the structure of a class or the XML specification changes. This process is cumbersome for programmers, because it requires maintaining an extra set of parallel definitions in a complex XML file, separate from code.

The JiBX binding compiler is run offline, augmenting Java byte code with code for handling mapping, resulting in increased performance. This extra build stage must be managed by the developer. Alternatively, JiBX can bind at runtime, but this requires that the programmer explicitly invoke the binding compiler, which requires overhead at the start of the application. `ecologylab.xml` does not need configuration files or a binding compiler, because all mappings are embedded in the source. Only by ignoring the overhead of the binding compiler is JiBX performance measured as comparable to `ecologylab.xml` (adding the binding compile step would require modifying Bindmark significantly). The operations of the JiBX binding compiler are analogous to the building the parse tree of just-in-time optimizations in `ecologylab.xml`, but more invasive. Yet, we include the JIT optimizations time in our benchmark measurements.

The Java Architecture for XML Binding (JAXB 2.0) [9] supports a wide variety of XML-Java bindings and provides an optional annotation metalanguage similar to that of `ecologylab.xml` (see Figure 7). However, unlike `ecologylab.xml`, JAXB does not support automatic binding of a collection to a `Map`, but only to a linear data structure. Further, using the metalanguage can be cumbersome, due to how JAXB utilizes JavaBeans [22]. Mapped fields can *either* be annotated in place or the `get`/`set` methods themselves must be annotated, instead of the fields, placing the annotation in an obscure location. In the former case, JAXB creates the `get` and `set` methods itself and will fail if such methods are provided by the developer, meaning that `get`/`set` methods must have unintuitive names that differ from their fields,

framework /	marshalling time (nanoseconds)			unmarshalling time (nanoseconds)			runtime
(# runs)	small (1000)	medium (100)	large (10)	small (1000)	medium (100)	large (10)	size (K)
ecologylab.xml	39,004	1,528,674	7,151,383	289,966	5,815,660	35,446,572	176
JiBX*	35,943	1,719,212	15,627,181	70,420	5,683,769	39,249,876	141
JAXB 2.0	65,749	2,223,438	10,567,599	362,868	14,889,646	87,146,075	3,800
XStream	413,465	18,373,301	173,198,604	719,290	28,818,238	211,358,650	368
Castor	1,183,407	11,831,280	55,922,610	1,813,563	22,926,830	162,330,464	3,000
XML file size (K)				1.42	123	1,003	
XML file lines				55	1,817	8,567	
XML file depth				2	20	45	

Figure 6. Performance in Bindmark [3] and runtime size statistics show strong performance of `ecologylab.xml`. Bottom rows indicate specifications of the XML document used for unmarshalling; marshalling uses a set of Java objects that match this XML.
*JiBX uses a binding compile step to optimize Java byte code for performance; this step is excluded from the benchmark.

unlike, for example, those produced automatically by Eclipse [4]. In contrast, when a scalar-valued field is annotated with `ecologylab.xml`, `get`/`set` methods are seamlessly optional.

Instead of using the metalanguage, an advantage of JAXB is that it can create class definitions from an XML schema, through an offline binding compile step. However, such classes are instantiated through factory methods, preventing subclassing, and its expressive power. Despite JAXB's inconveniences, it is considered one of the easiest frameworks to use [3]. `ecologylab.xml` is faster in every phase of the Bindmark suite, with an overall performance roughly twice as fast. In runtime size, which impacts distributing applications through the internet, `ecologylab.xml` is more than an order of magnitude smaller.

JAXB and other frameworks utilize `javax.xml.bind.annotation` to define their annotations [9]. We considered using the annotations specified in this package, but decided that they could be confusing to the developer, because there are almost 30 annotations, with excessively complex syntax and semantics. These annotations include wrappers, inline binary data, and special schema annotations. Providing a tighter set of annotations facilitates learning the metalanguage and understanding code written in it (see example, Figure 7). This aids software usability.

XStream [28] is a binding framework that automatically generates XML based on class definitions. The user may specify an optional configuration file to disambiguate tag-to-class mappings, or can use a rudimentary metalanguage. XStream's metalanguage lacks the expressive power of `ecologylab.xml`, which is nonetheless roughly an order of magnitude more efficient.

Finally, Castor [5] provides persistence through SQL databases or XML files. In order to perform Java-XML binding, Castor *requires* that the programmer author `get` and `set` for each field to be translated, or declare them `public`. Castor attempts to map between Java and XML automatically, but sometimes requires the programmer to maintain an additional configuration file to disambiguate mappings. `ecologylab.xml` is roughly an order of magnitude faster and smaller than Castor.

8. APPLICATIONS

`ecologylab.xml` has been used as a foundation for numerous research projects and educational assignments. Graduate computer science students who have used the framework in coursework have commented on the framework's power and ease of use. Many software projects have been developed with the framework. These include libraries that support dialects of XML, component-based reusable frameworks, and domain-specific applications.

8.1 Libraries

The framework is currently distributed with `ecologylab.xml` object declarations for a variety of popular XML dialects. These declarations are all one needs to parse instances of XML encountered in the wild, and to generate such XML. They can be extended with application-specific logic. The dialects already supported include Dublin Core, Endnote, International Children's Digital Library, iTunes podcasts [1], Java Network Launching Protocol (JNLP) [23], Online Processor Markup Language (OPML), RSS (all versions) [13], Google Earth Keyhole Markup Language (KML), Yahoo Media, and Yahoo Web Services. We will continue to add support for additional dialects to the distribution, as per the needs and contributions of the community.

8.2 Studies Framework: `ecologylab.studies`

User feedback is critical in software development and scientific experimentation. To facilitate feedback elicitation, we develop the Studies Framework, `ecologylab.studies`, a component-based Java web application for building online user studies that administer complex questionnaires, and launch and gather data directly from instrumented Java applications.

`ecologylab.xml` is the basis of Studies Framework components, including the specification of questions, paths through the questions, the dynamic generation of application preferences based on study questions, the dynamic generation of JNLP files [23], and the storage of user response data.

Experimenters can rapidly develop and deploy a user study by authoring an XML study specification file. This XML file specifies questions and their form (essay, multiple choice, etc.). It includes question paths, which allow the application to automatically counterbalance conditions for the study. Question types are easily extendable by creating new `ecologylab.studies.state.questions.Question` subclasses and adding the appropriate XML to a study file. Some question types launch applications, some ask multiple choice questions, and others ask open questions.

One of the most powerful abilities of `ecologylab.studies` is the ability to launch custom Java applications. Because JNLP files are a dialect of XML [23], experimenters provide a base JNLP file and can specify logic that customizes the JNLP file automatically on a per-participant basis. These deployed applications can also

```
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;
  …
```

```
<binding>
  <mapping name="rss" class="Rss">
     <value name="version"
       field="version"
       value-style="attribute" />

    <structure name="channel"
       field="channel">
      <collection field="items"
       item-type="Item" />
    </structure>
  </mapping>

  <mapping name="item" class="Item">
    <value name="title" field="title"/>
    <value name="link"  field="link"/>
    …
  </mapping>
  …
</binding>
```

Figure 7. Comparison of RSS code in `ecologylab.xml` (left), JAXB 2.0 (center), and JiBX (right). The JiBX code is an XML configuration file, which would need to be maintained alongside Java code similar to the other two. Note that, because the *fields* have been annotated, the JAXB example *cannot* have `get`/`set` methods for its data fields.

receive preferences that are based on the user, as well as his/her responses to study questions. The applications are easily instrumented using the Interaction Logging Services (see below), so that a participant's actions in the application are recorded, in conjunction with their direct question responses.

8.3 Semantic Distributed Computing Services

The Lightweight Semantic Distributed Computing Services (LSDCS) integrate information semantics with remote operations on data by using `ecologylab.xml` as a substrate to connect internal data structures with XML serialization structures [26]. In the spirit of object-oriented programming, service method declarations are also coincident. Programmers customize message subclasses by providing annotated information fields and overriding methods that will be automatically called to perform execution on the information remotely. Class definitions thus define the XML structure of messages, and their behavior. Multi-threading and the NIO framework [24] make the LSDCS efficient and enable high performance. LSDCS servers are resilient and robust, with resumable client/server sessions and security features. The basis in `ecologylab.xml` makes the LSDCS easy to extend.

Messages consist of two types: `RequestMessage`, which will be sent from a client to a server and provides the `performService(…)` method; and `ResponseMessage`, which goes from server to client and provides the `processResponse(…)` method. We first describe the interoperation between a Java LSDCS client and Java LSDCS server, then describe how the LSDCS server handles other sources of XML messages.

In a Java LSDCS client, an instance of a subclass of `RequestMessage` (a subclass of `ElementState`) is translated to XML and sent over the network to a Java LSDCS server. On the server, a `TranslationScope` defines how the incoming XML is translated into a local copy of the `RequestMessage` subclass instance. Its `performService(…)` method is called automatically. `performService(…)` is executed within the client session scope, which defines a set of server state variables that can be read and modified through their own interfaces. The scope also provides a space in which the client may store information to maintain context between messages and restore this context after an unexpected disconnect. The result of this call is an instance of a subclass of `ResponseMessage` (another subclass of `ElementState`), which the service programmer has designed.

The `ResponseMessage` subclass instance is then translated to XML and sent back to the LSDCS client. The client translates the message from XML into a `ResponseMessage` instance, using its `TranslationScope`, and then automatically executes the message's `processResponse(…)` method in the client's scope.

The LSDCS is flexible, as `TranslationScope`s define how XML translates into imperative Java objects. A programmer can author a message class to capture XML from a non-LSDCS source, such as a form submitted in a web browser, and use it to perform custom computation remotely. This mechanism is used for providing combinFormation (Section 8.5) with user-generated seeds from a web interface.

8.4 Interaction Logging Services

Interaction logging is an important method for collecting quantitative data for the evaluation of interactive systems [17]. The Interaction Logging Services (ILS) build upon the LSDCS [26], enabling application programmers to easily instrument software to automatically gather information about state and user actions. The ILS enables recording through multiple modalities: local disk, local disk with memory-mapping (Java NIO [24]), and/or remote server via LSDCS. Log messages, which are application-specific, are not known by the LSDCS logging service. Thus, the service can handle input from multiple heterogeneous clients without configuration. Log analysis tools extract data from log XML to popular spreadsheet and statistical analysis packages, using comma-separated values (CSV).

8.5 combinFormation

combinFormation is a mixed-initiative exploratory search and creativity support tool that integrates processes of searching, browsing, collecting, mixing, organizing, and thinking about information [7][11][12]. By mixed-initiative, we mean that generative agents work in partnership with a human participant to collect relevant information, and represent the collection in the form a visual composition that changes over time. Images and text engage complementary cognitive subsystems. Each collection of information resources is represented as a connected whole. This promotes information discovery, the emergence of new ideas in the context of information.

combinFormation (cF) uses LCDCS (Section 8.3) to operate an information collection visualization service. This allows external clients to supply search seeds that tell combinFormation's agents where to collect information. Seeds can specify search queries to particular search engines, web documents (HTML, RSS, PDF), and the names of pre-curated seed sets, such as 'news' and 'art museums'. In a typical deployment, the user specifies these seeds through a web page. cF is launched as a Java Web Start application. A JavaScript client sends the user's search seeds to the cF service using the LSDCS using a form of AJAX. combinFormation also uses `ecologylab.xml` for persistent storage of collection metadocuments. The application has been validated through field studies [11], and laboratory studies [12] in which data has been collected using the Studies Framework and ILS.

8.6 Games for Team Coordination

The Teaching Team Coordination through Location-aware Games (TTeCLoG) project produces multiplayer games for exploring the impact of games on team skills and communication in stressful environments. `ecologylab.xml` is used for network communication, logging and replay, and customized launches during user studies. To support realtime collaborative distributed game play, each of four game clients (1.60GHz Intel Pentium M 725) sends the server (2.4 GHz Intel Core 2 Duo) a message and receives a response every 100 milliseconds. Each server response is ~4.5 kilobytes long, containing over 100 XML entities.

TTeCLoG generates comprehensive logs of game play events with the ILS, supporting both quantitative and qualitative analysis. A playback application allows researchers to watch replays synchronized with audio recordings, enabling investigation of how players coordinate action. Analysis tools extract important events, such as the number of radio activations by each player, to CSV files, for import into statistical analysis applications. The Studies Framework launches the game after acquiring data from participants, using `ecologylab.xml` to produce a custom JNLP file for each player based on their study data.

9. CONCLUSION

XML is a *lingua franca* for information semantic services in digital libraries, news feeds, the semantic web, multimedia, e-commerce, and other applications. The lack of typing provided

directly by XML creates challenges for information semantics programming. Strongly typed representations of complex data structures provide better support than the untyped XML DOM for programming in the large, the long term development and maintenance of complex programs.

The present research develops an approach to the representation of information semantics that is grounded in the process of writing programs in a strongly typed language. The linguistic symbiosis of embedding metalanguage in imperative language source enables tight integration of information design and program functionality, facilitating software development and maintenance. Component-based translation scopes help programmers maintain sets of the annotated object definitions. Translation scopes are defined in source code, with class literals, so they can be manipulated through IDE operations such as re-factoring, contributing further to software usability.

We are bucking the trend of runtime platform agnostic information specifications. A reason for defining semantics in an imperative language, such as Java[3], is that it enables the developer to express them in ways that are optimally convenient and efficient, facilitating program development and maintenance. Automatic generation of `ecologylab.xml` declarations from schemas sounds good in theory, and will be addressed in future work. Yet in practice it is less than a panacea. As with RSS, there is not always a schema. When schemas exist, they can be useful for building applications that interoperate. Yet schemas are, in key ways, less expressive than the semantics of the metalanguage in conjunction with Java itself. Many different possible imperative language declarations can correspond to a particular XML schema. For example, the `ecologylab.xml` metalanguage directives `@xml_collection` and `@xml_map` offer interesting options for information semantics expression. The two constructs can be used to represent exactly the same data, corresponding to the same XML schema definition. Yet, in situations where a collection is best represented as a hashed data structure, such as in typical cases where one field serves as unique identifier, or key, `@xml_map` is better. When such a collection is large, the computational cost of reading the data twice: once into an automatically generated linear collection form, and again into a hashed data structure may be prohibitive. Further, this will require extra work on the part of the programmer, extra lines of code to write and maintain. A fundamental goal of the present framework is to eliminate both of these inefficiencies. As a result of the parse tree of just-in-time optimizations, data binding is efficiently accomplished. Expressivity and usability that promote software engineering are combined with high performance. The framework has proved to function as a strong basis for developing applications and layered semantic services.

10. ACKNOWLEDGEMENTS

Support provided by NSF through IIS-0633906 and IIS-0742947. Special thanks to Bjarne Stroustrup for his invaluable feedback.

11. REFERENCES

[1] Apple – iPod + iTunes. http://www.apple.com/itunes/.

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

[3] Bindmark Project for benchmarking XML tools. https://bindmark.dev.java.net/ Access 10/30/07..

[4] Eclipse Foundation. Eclipse.org home. http://www.eclipse.org.

[5] ExoLab Group. Castor Project. http://www.castor.org. 2005.

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

[7] Interface Ecology Lab, combinFormation, http://ecologylab.net/combinformation.

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

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

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

[11] Kerne, A., Koh, E., Dworaczyk, B., Mistrot, M.J., Choi, H., Smith, S.M., Graeber, R, Caruso, D., Webb, A., Hill, R., Albea, J., combinFormation: A Mixed-Initiative System for Representing Collections as Compositions of Image and Text Surrogates, *Proc JCDL 2006*, 11-20.

[12] Koh, E., Kerne, A., Damaraju, S., Webb, A., Sturdivant, D., Generating Views of the Buzz: Browsing Popular Media and Authoring using Mixed-Initiative Composition, *Proc ACM Multimedia 2007*, 228-237.

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

[14] Harold, E. R. *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, TrAX*, Addison-Wesley Prof, 2002.

[15] Object Management Group, Catalog of OMG CORBA/IIOP Specifications, http://www.omg.org/technology/documents/corba_spec_catalog.htm. Access 2/15/07.

[16] Sethi, R., *Programming Languages: Concepts & Constructs*, New York: Addison Wesley, 1996.

[17] Sharp, H., Rodgers, Y., Preece, J., *Interaction Design: Beyond Human-Computer Interaction*, New York: Wiley, 2006.

[18] Sosnoski, D. M. JiBX: Binding XML to Java Code. http://jibx.sourceforge.net/. 2007. Access 10/30/07.

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

[20] Sun Microsystems. Collections Framework Overview. http://java.sun.com/j2se/1.5.0/docs/guide/collections/overview.html. 2004. Access 10/9/07.

[21] Sun Microsystems. Generics. http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html. 2004. Access 9/23/07.

[22] Sun Microsystems. JavaBeans. http://java.sun.com/products/javabeans/. 2007. Access 10/30/07.

[23] Sun Microsystems. Java Web Start. http://java.sun.com/j2se/1.5.0/docs/guide/javaws/. 2004. Access 10/13/07.

[24] Sun Microsystems. JDK 5.0 New I/O-related APIs & Dev Guides. http://java.sun.com/j2se/1.5.0/docs/guide/nio/. 2002.

[25] Sun Microsystems. Reflection. http://java.sun.com/j2se/1.5.0/docs/guide/reflection/index.html. 2002.

[26] Toups, Z.O., Kerne, A., Webb, A. Composing service components with lexical scoping for little semantic webs: An expressive framework. Submitted to *Intl Semantic Web Conference 2008*.

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

[28] XStream, About XStream. http://xstream.codehaus.org. 2007.

[3] Future work will extend the framework to other programming languages.