Interface Ecology Lab Technical Report 07-01 [DRAFT]

Inter-Language Translation Framework for Distributed Information Semantics

Type System, Contextualized Meta-Language, and Intuitive Respelling

Andruid Kerne, Zachary O. Toups, Blake Dworaczyk, Madhur Khandelwal

Interface Ecology Lab / Center for Study of Digital Libraries / Computer Science Department
Texas A&M University
College Station, TX 77845
{andruid.filter, toupsz, blaked, madhurk}@gmail.com

Abstract

Information is data that communicates, representing stuff of significance to human beings. Information semantics are formal representations that define structures of information. We develop an inter-language translation framework for information semantics. By using programming language mechanisms for translation across languages, including a formal type system and a meta-language, the framework provides expressive power. Just-in-time optimizations enable runtime efficiency without the use of the preprocessors that are typical of most XML data binding frameworks. Intuitive respelling of names works with typed procedural language definition of scalar values and elements to provide direct semantics for types. These are augmented by a contextualized meta-language, through which details of translation are specified in the midst of procedural language source code. Translation spaces group together sets of package to class name bindings, and also serve to assemble some optimizing data structures. Translation spaces can be composed and inherited, functioning as nested scopes, each of which corresponds to a set of components. Initial support for generics is provided, and further support is planned. A Distributed Computing Service *and other reusable component layers have been built using the translation framework.*

Keywords *XML, data binding, metalanguage, type systems, distributed computing, services oriented architecture*

1. Introduction

Information is data that informs, that communicates, representing stuff of significance to human beings. A medium is a sensory representation of information, a representation designed to convey. The concepts of information and media represent different sides of the same coin, and this coin is the essence of communication that connects human beings and bridges the digital and physical worlds. Semantics are formal representations of structures that define entities and connect them with relationships. The more that the digital permeates human life, the more important it becomes to use information and media semantics to represent human experiences. These semantics grow complex, with deep nesting and inter-referentiality. XML, the Extensible Markup Language is a standard form for representing information and media semantics. Procedural programming languages are the most powerful means for operating on these semantics to accomplish real world tasks. The goal of the present research is provide programmers with an expressive framework for defining rich information and media semantics, serializing the resulting complex nested semantic structures that are operated on in procedural programming environments to and from XML, persistently storing and retrieving such representations, and using distributed services to pass them as semantic remote procedure call messages between processes and hosts, via TCP/IP.

In programming languages, essential forms are types and variables; in XML, we have element tags and attributes. So when translating objects from a procedural programming language to XML, a key question is, “How do we map procedural language types and instance variables to XML element and attribute names?” The set of bindings that provides the basis for these mappings is equivalent to the bindings between variable names, types, and values, within the procedural programming language, itself. Thus, it makes sense to apply programming language techniques to defining and translating these semantics. The present research develops intuitive mappings, a type system, and a meta-language for information and media semantics, in order to maximize convenience and expressive power for programmers. The form of this is an inter-language translation environment that strictly binds a procedural language with XML. While the present research utilizes Java as the procedural programming language, on account of its widespread availability across platforms, and its support for reflection, this research can be extended to support other languages.

The semantic web, digital libraries, personal repositories such as iTunes, e-commerce, and other application areas utilize XML to store and transmit complex data. XML structures may be highly nested trees, with many fields at each level. While XML parsers are readily available, the document objects that result from initial parsing lack type information, requiring a second, hand-coded phase of translation to usable typed objects in procedural languages. Thus, a burden remains for programmers to develop code that traverses the type-deficient DOM, extracts informative fields, and builds typed objects. This research frees software developers from this burden, by developing an intuitive, easy-to-use framework that automatically translates the type-deficient DOM into a strongly typed tree of procedural language objects.

The `ecologylab.xml` inter-language translation framework for distributed information and media semantics is being made available as open source for the benefit of the community [5]. It is already in use in a wide range of applications. The development of parsers for standard XML dialects such as RSS and Dublin Core has proved trivial. Distributed Computing Services enable the rapid construction of messages that specify operations across machines, the remote invocation of these messages, and the receipt of results. Another re-usable component is the Interaction Logging Subsystem. A generalized servlet-based application suite runs user studies. Interactive applications include visual information collecting and a multi-player game.

~~We begin by introducing the `ElementState` base class, which is the heart of the translation framework. Next, we provide an overview of our use of reflection and intuitive mappings between names in XML and Java. This is followed by a detailed description of how an XML DOM tree is automatically translated into a tree of Java objects. We continue to develop the RSS example of Figure 1. The next subsection addresses our method of defining which Java fields are translated to XML, and which are skipped. It is also possible for to skip translating some XML elements and attributes to Java. Then, we describe our extensible type system for performing type conversions between Strings and scalar values. `TranslationSpaces` group together sets of mappings between packages and class names, as well as providing a means for providing extra redundant mappings between an XML element name and a Java class name. Finally, we introduce a number of structures for performance optimization that enable the translation processes to be conducted very rapidly. The optimization structures are built at runtime, just in time. The result is that the burden on the programmer is minimized, without performance penalties.~~

2. Principles and Motivations

The first principle of the `ecologylab.xml` framework is that information semantics are declared directly through Java object declarations. A set of Java class definitions function as the definition of an equivalent XML Document Type Definition (DTD). We completely avoid intermediate interface definition language declarations, like those of CORBA IDL [8], with their requirement of multiple definitions of the same data structures, and associated burden of maintenance.

Our original idea was that Java class declarations would be sufficient to specify XML declarations and the process of translation. The benefit of this approach is that there is one set of definitions to maintain, and the definitions can be specified in the same place as operations on the data. The framework uses reflection and intuitive respelling of names across languages to automatically translate XML to and from Java. Reflection enables procedural traversal of and assignment to the fields of Java data structures at runtime. A set of rules defines mappings between XML tag and attribute names, and Java types and field names. The rules are simple and intuitive, making it easy for programmers to specify them. The result is that no intermediate interface definition language is required, so the programmer does not need to run an intermediate translator. The programmer simply defines Java classes that can immediately be operated on within a Java program. The framework uses these class definitions to parse and generate XML, resulting in reusable objects.

However, expressive power was found to be insufficient for using Java declarations as the sole basis of XML objects. The first additional need is to specify which fields are to be translated, and which are to be skipped. While others have used the `transient` keyword for this, we believe that semantics are more clear when the fields that will be translated are declared explicitly. For another example, consider that there was not an easy way to specify whether scalar values should be represented in XML as attributes or as leaf nodes. Similarly, leaf nodes could be represented as text or as CDATA.

The solution that increases expressive power, while continuing to maintain all specifications of translation in the context of procedural type definitions is a *meta-language*. Java's annotations mechanism [3] enables us to integrate meta-language constructs directly with Java object and field declarations. The annotation mechanism is a generalization of that provided by javadoc, and the motivation is similar: integrating the two kinds of declarations facilitates software maintenance. Java classes and instance variables are declared with ecologylab.xml meta-language annotations, which specify which instance variable slots should be translated, and, as appropriate,

```
<rss version="2.0">
- <channel>
  <title>NYT > Home Page</title>
  <description>New York Times > Breaking News, World News & Multimedia</description>
  <copyright>Copyright 2006 The New York Times Company</copyright>
  <lastBuildDate>Mon, 17 Jul 2006 20:05:00 EDT</lastBuildDate>
  <item>
  <title>2 Leaders Want Peacekeepers to Be Sent to Lebanon</title>
  <link>http://www.nytimes.com/2006/07/17/world/middleeast/17cnd-
mideast.html?ex=1310788800&en=0a50575c7971dd8b&ei=5088&partner=rssnyt&emc=rss</link>
  <description>Tony Blair and Kofi Annan called today for an international force to quell
the fighting between Israel and the Hezbollah militia.</description>
  <author>STEVEN ERLANGER and JAD MOUAWAD</author>
  <pubDate>Mon, 17 Jul 2006 00:00:00 EDT</pubDate>
  <guid      isPermaLink="false">http://www.nytimes.com/2006/07/17/world/middleeast/17cnd-
mideast.html</guid>
  </item>
  <item>
  …
  </item>
  …
  </channel>
</rss>
```

Figure 1: RSS 2.0 XML Example for The New York Times Home Page.

how.

3. Semantic Entities: Scalar and Nested Element Types

We begin with a topological description of the semantic entities of each of the source and target languages, Java and XML. We use these as the basis for a specification of requirements for the framework's translation mechanisms. Then, we develop an example, and describe the mechanisms in detail.

Consideration of how to define mappings between Java and XML begins with a topology of the representation of types and values in mathematics, and consideration of the semantics of each language. Mathematically, the specification of a single value is called a *scalar*, while a linear (one-dimensional) set of values is a vector. This distinction is similar to, yet different from the distinction between primitive and reference types. The basis for the differences is that we are interested in how the informational entities function in the world, rather than how they are stored in the computer.

Java objects directly enable the specification of a vector-like set of slots [3]. The slots in Java objects can be declared as either primitives or reference types. While nested reference types may be utilized to represent more complex, higher dimensional structures, primitives are always scalar values. Some reference types, such as Strings, Colors, and Dates, function as scalar values, while for others, the structure of nested objects is essential.

Entity declarations in XML are gathered together as a *document* [16]. The operable tree representation of a document is known as its document object model, or DOM. The significant entities in XML are elements and attributes. Element is the basic entity. The name of an element is also known as its *tag*. Elements typically function in a manner analogous to objects in Java. Each element may have any number of children, which take form as *attributes* or sub-elements. Attributes function as slots that can only represent scalar values.

The set of structural forms represented in the DOM, including elements, attributes, and text are generally known as *nodes*. The DOM begins with a *root element*. Each element other than the root has a single parent element. Trees may terminate with *text* or *CDATA nodes*, which, like attributes, contain scalar values. When a branch of an XML tree terminates with an XML element that has a single text element child, holding a scalar value, we call this a *leaf node*. Other sub-elements are first class nested elements, which can represent complex objects. The inter-language translation framework considers leaf nodes and attributes as equivalent alternative means for representing scalar values, and true nested elements as the basis for representing hierarchical relationships. This distinction between nested elements and scalar values is semantically fundamental.

The standard XML DOM API available in Java [11] and other languages enables us to obtain the root element of a document, and the attributes and sub-elements of each element, their names and values. These mechanisms enable us to traverse the DOM tree. Analogously, the Java Reflection API enables programs to discover all the Fields within a Class object, including their names and types [3]. Applying a reflection Field object to an instance of the class enables us to obtain or set the field's value. Our strategy for translating from XML to Java involves walking the XML DOM and finding, creating, and operating on associated fields and values in Java objects. Likewise, when translating from Java to XML, we walk the tree of Java `ElementState` objects, and create associated XML elements and fields.

4. The RSS Example

We will develop an example, using XML, in the popular RSS dialect, which is used for content publishing and syndication. The purpose of the example is to show how one goes about declaring `ecologylab.xml` objects in correspondence with XML of a particular format. What the framework provides is not just serialization, but expressive control of which fields are serialized, and how. A programmer would take very similar steps to represent any other XML object using the framework. The example RSS feed XML of Figure 1 was published by the New York Times to represent the lead stories of its home page.

5. Type System

Inter-language translation is accomplished by recursive descent traversal of a tree of declarations in one language, in order to generate a tree of declarations in the other language. A fully extensible type system on the procedural language side provides a formal set of relationships that specify the mechanics of this inter-language translation. Intuitive re-spelling formalizes the translation of names between languages. Scalar type translators are developed to marshal XML DOM strings into scalar values, and vice versa. Subclasses of `ElementState` are developed to represent XML root and nested elements as appropriately typed Java reference objects. Slots within these reference objects may, in turn, represent scalar values or nested elements. Some translators are provided with the framework. The framework provides simple means for application developers to develop other translators, as s/he sees fit. Translation spaces group sets of type name to type translator bindings into re-usable scope building blocks.

```
package ecologylab.xml.library.rss;

public class Rss extends ElementState
{
   @xml_attribute float    version;
   @xml_nested    Channel  channel;
}
```

Figure 2: Rss, a subclass of ElementState.

5.1 Intuitive Inter-language Respelling

By convention, XML element and attribute names are spelled in lower case with underscore delimiting word breaks. Java class names are capitalized, and use internal capitals to delimit word breaks. Java instance variable and method names begin with lower case, and also use internal capitals to delimit word breaks. The inter-language translation framework formalizes these relationships by providing automatic respelling. For example, according to convention, a two word element in XML would be spelled as `<composition_space>`, while the corresponding Java class would be spelled as `CompositionSpace`, or an instance variable name as `compositionSpace`.

5.2 Translating Scalar Value Types

Scalar values are represented directly in XML as attributes or leaf nodes. In the XML these are Strings. In a procedural Java program, we wish to operate on them as typed values. In order to remove the burden of performing type conversions from the programmer, when using the framework, these fields are represented directly by instance variables of the appropriate type. These include the regular Java primitive types, such as `int`, `float`, `double`, and

```
package ecologylab.xml.library.rss;

public @xml_inherit class Channel extends ArrayListState<Item>
{
   @xml_leaf  String       title;
   @xml_leaf  String       description;
}
```

Figure 3: Channel, a subclass of ArrayListState, illustrates leaf nodes and a Collection.

```
package ecologylab.xml.library.rss;

public class Item extends ElementState
{
   @xml_leaf  String             title;
   @xml_leaf  String             description;
   @xml_leaf  ParsedURL          link;
   @xml_leaf  String             author;
   @xml_leaf  Date               pubDate;
}
```

Figure 4: Item, a subclass of ElementState, represents most of the data from the feed.

`boolean`. Other already supported types include reference types, such as `String`, `URL`, `Color`, and `Date`. Two mechanisms are provided for marshalling `String` values into instance variables of these types, and assigning them to the appropriate field.

Scalar value translation is structured so as to minimize the burden for the information semantics programmer. Scalar type translators automatically marshal strings into typed values, and vice versa. Unlike with Java Beans, there is no need to explicitly provide set and get methods for declared fields. Explicit `set()` methods may optionally be provided when custom behavior is desired during marshalling.

We have developed an extensible type system, in the package `ecologylab.xml.types.scalar`. A subclass, `ScalarType`, is extended for each primitive or supported reference-based scalar type. Any programmer wishing to implement new custom types can simply provide a new subclass of `Type`, and register it in an appropriate **`TranslationSpace`**s .

The methods of `ScalarType` for subclasses to override are `getInstance(String)`, `isReference()`, `setField(Object, Field, String)`, and `toString()`. To make generated XML shorter, fields containing the default value for a type are not translated into XML at all.

5.3 Translating Elements to Typed Objects

The heart of the translation framework is the `ElementState` base class, defined in the package `ecologylab.xml`. To represent each non-leaf node XML element in Java, the programmer defines a subclass of `ElementState`. Within this subclass, each attribute and nested sub-element is declared as a field. Nested elements are defined using additional subclasses of `ElementState`. Attributes and leaf node constitute scalar values. Each field to be translated must be declared with an appropriate meta-language annotation.

5.4 Translation Spaces

Sets of type bindings are grouped together into scopes known as **`TranslationSpace`**s. A translation space defines which Scalar Types will be used to marshal string values, and which ElementState types will be used to represent elements. A **`TranslationSpace`** registry provides binding of **`TranslationSpace`**s to names, in a re-usable manner. For element types, the question generally answered by **`TranslationSpace`**s is, "In which package will a class of each appropriate name be found?" Each **`TranslationSpace`**s has a default package, used for locating the class. Classes can also be specified explicitly. This is faster at runtime, but more work for programmers. It is necessary in cases when the set of element name to class bindings spans beyond a single package.

`TranslationSpace`s can inherit from other **`TranslationSpace`**s. They form nested scopes of bindings. This enables modular organization of sets of bindings into **`TranslationSpace`** components, which can be mixed for re-use, as appropriate. While package names can be specified for rapid development, specifying explicit sets of Class objects enables more efficient processing, because it bypasses the slow searches for class objects typical of Java's Class.forName() operation. Though just-in-time optimizations ensure that those searches only happen once per program invocation, and only as needed, they could slow down program initialization.

Details on the API for working with translation spaces.

6. Meta-Language

The meta-language consists of declarations, layered over the procedural language in order to enable fine-grained specification of the mechanics of translation. The meta-language specifies which super classes are involved in translation, as well as which fields are involved and how.

Requirement of the `@xml_inherit` directive limits processing of super class object chains. It is required in cases in which the parent class contains values that are intended to be included in the translation. The directive serves to optimize runtime performance, because the optimizer must recursively collect fields from each parent class in the chain.

The specification of single scalar valued slots for translation is accomplished with `@xml_attribute` or `@xml_leaf`, depending on whether attribute or leaf node representation is preferred in XML. With , the representation of the value as CDATA [16], rather than as a plain text node, is specified through an argument in the form of `@xml_leaf(CDATA)`.

Several directives are used to specify the translation of first class nested elements. `@xml_nested` specifies a single nested element. `@xml_collection` and `@xml_map` specify sets of elements, in either linear or hashed forms.

7. Contextual Translation by Type and by Field Name

We define procedural semantics for translation of elements and scalar types. To understand the translation process, please be aware that depending on the topological semantic context of an entity, the XML entity name may be mapped to an instance variable name or a class name. The root element is translated to a class name. Once this class has been identified, it can provide a context so that the XML entity associated with each single scalar or nested element slot is translated by field name. In turn, some collection and Map objects must be translated by class name, while others can be translated by field. The following subsections describe this process in detail.

7.1 Translating the Root Element

The root element of an XML document defines the start of the translation process. Until an appropriate class is identified for translating the root, there is no context for translation. Thus, the framework begins by using the `TranslationSpace` to find an `ElementState` subclass in the procedural language that corresponds to the root tag name. In the case of our example (Figures 1 and 3), the root element will be of type *`Rss`*. Figure 2 shows a simplified definition of this class. In this example, nested entities include a scalar value, and a single nested element.

7.2 Translating a Scalar Value Slot

We declare fields in our subclass, corresponding to each scalar value. Scalar values may be represented either as attributes, like version in *`RssState`*, or as leaf nodes, like the `<title>` element in `Channel` (see Figure 1). Either way, the name of the Java field corresponds to the name of the XML entity. The type is declared in the Java field declaration. Automatic marshalling, that is, type conversion to and from strings to these types, is performed as per Section 5.2.

Each scalar-valued entity that should be represented in XML as an attribute must be declared with the `@xml_attribute` annotation modifier. Alternatively, to serialize a scalar as a leaf node, declare the field with the `@xml_leaf` modifier.

7.3 Translating a Nested Element Slot

Often, one declares a single XML element nested inside a parent element, like the `Channel` element in Figure 3. In a DTD or schema, this corresponds to an ENTITY declaration, either unmodified or with the “?” regular expression modifier [16]. The framework does not enforce membership requirements; it simply provides slots for value translation. A single nested element is declared with the `@xml_nested` modifier. Continuing the example, the `Channel` element is defined in Figure 3.

7.4 Collections: Translating Nested Sets

The framework features a highly extensible mechanism for embedding sets of nested sub-elements within an element. Collections are procedural language data structures used to represent the sets. This corresponds to a DTD ENTITY declaration with either the * or + regular expression modifier [16]. We consider three specification mechanisms for nested sets: simple homogeneous nested sets with constituents of a single type, a mechanism that uses method override to map nested set elements into particular collection data structures by type, and a mechanism that connects a meta-language argument with a generic parameterized types to enable specification of child elements by tag name, and mapping of types.

7.4.1 ArrayListState: Homogeneous Nested Sets

The usual structure of a collection in XML, as with the `<item>` elements in Figure 1, is that multiple child elements of a parent (in this case, the `Channel` element), have the same tag name and the same structure. The framework provides `ArrayListState`, a subclass of `ElementState` that is a wrapper of `ArrayList`, to handle these simple cases. We see the example declaration that uses this in Figure 4. As the example shows, `ArrayListState` is defined as a generic, which can be parameterized with the class that corresponds to the child elements, for type safety. This is all that must be specified for inter-language translation in these common contexts. A limitation on this structure, and the generalization provided in the next section, is that the children which are grouped into the set must be first class elements, and not scalar values. For situations in which uniqueness of child elements must be maintained, `HashSetState` is provided, and for cases in which the set must exclude concurrent operations, we provide `VectorState`.

7.4.2 Heterogeneous Nested Sets: Mapping by Type

During translation from XML, the tag for each sub-element is translated into a field name via intuitive inter-language respelling (Section 5.1). If this does not match a field in the object, then the tag is translated into a class name. The active `TranslationSpace` is used to seek a `Class` object (Section 5.4). If this is successful, `getCollection()` is called on the `ElementState` object, with this `Class` object as the argument. The framework looks for an appropriate `Collection` field that is declared with the `@xml_collection` modifier. `Collection` is an interface in Java, which is implemented by classes such as `ArrayList`, `Stack`, `Queue`, and `HashSet`. This interrogation is accomplished via a method which returns null in the base `ElementState` class, but which can be overridden in subclasses that utilize collections:

```
protected Collection getCollection(Class
thatClass)
```

The subclass of `ElementState` may return a non-null `Collection` object reference. If it does, the translation method will recursively descend to form the nested object according to its `Class`, and then invoke the `add()` method to add it to the `Collection`. This is the mechanism used by `ArrayListState`.

7.4.3 Heterogeneous Nested Sets: Mapping by Tag

A more general mechanism enables mapping child XML sub-elements into procedural language collection data structures by tag. This means that there can be multiple tags associated with the same class at the same level in the XML, yet in the Java, these entities will be grouped into homogeneous collection data structures, for operational convenience. This is accomplished by combining more complex meta-language with parameterized generic declarations. The meta-language syntax is `@xml_collection(tag)`. These kinds of collections declarations can operate directly on scalar types, to support sets of leaf nodes, as well as on first class nested elements. An example declaration looks like this:

```
@xml_collection("tag_name")
ArrayList<String>      tagNameSet;
```

It is easy enough to specify the tag as an argument to the annotation. This construct solves the key problem for this kind of expression: how to know what type to use for these sub-elements. Parameterization of the generic collection declaration provides a specification of this type information. Using reflection, the framework is able to dynamically acquire the type token parameter at runtime, and thus to assign a type to sub-elements that are set members. Multiple declarations of this kind can be specified in the same object to enable heterogeneous collection of elements with different tags, without grouping them together into a homogeneous parent `ArrayListState`.

7.5 Maps: Alternative Set Representation

Sometimes, it is desirable to store set elements using hashing, instead of a linear structure. In many such cases, the key for the hash operation comes from a scalar value in the element, itself. Our solution for supporting these cases starts with a simple interface, `Mappable`, which simply provides a method for obtaining the hash key.

```
package ecologylab.xml.types.element;

public interface Mappable<T>
{
   public T key();
}
```

Since recursive descent processing will form the child element before trying to insert it into the set, we can now provide all of the same functionalities as those for linear sets, for hash tables. These are accomplished by annotating a declaration of a field of type `Map` with

```
@xml_map
```

```
or
```

```
@xml_map("tag_name")
```

Thus, as with collections, we can support three types of declarations: simple homogeneous nested maps with constituents of a single type (via `HashMapState`), a mechanism that uses method override to insert nested set elements into particular map data structures by type (via `getMap()`), and a mechanism that connects a meta-language argument with a generic parameterized types to enable specification of child elements by tag name, and mapping of types.

Again, the last form requires a parameterized declaration of the field's type, such as

```
@xml_map("tag_name")
HashMap<Foo, Mappable>   tagNameMap;
```

It is important to note that first type parameter in the HashMap declaration corresponds to the type variable used in defining the `Mappable`. Thus, in practice, it can be any type. The second parameter must be the type that implements `Mappable`.

<<Develop a more real example that uses @xml_map(tag) with a generic declaration.>>

7.6 Custom Behaviors for Nested Elements

If `getCollection()` returns null, there is one final possible mechanism for translation. This mechanism enables developers to provide custom processing for nested elements, based on their tag name. If an XML element has not been matched via any of the above parsing mechanisms, it will be passed to the method

```
protected void addNestedElement(
ElementState elementState)
```

The default implementation provides a warning that the element is being ignored, but only the first time it is invoked for a particular object. This method can be overridden to provide custom behaviors. An example of such a behavior is to add an entry into a hash table in an application specific way, such as to create an entry for a URL the first time it is observed.

8. Invoking the Translator

Translation is accomplished by calling the static `translateFromXML()` method in `ElementState`. The arguments to this method are an `InputStream` or equivalent (e.g., URL, File, String), and a `TranslationSpace` (see Section 5.4). The method performs a recursive descent of the supplied XML DOM, performing translations as per the intuitive rules specified in the following subsections. Translation in the other direction is accomplished by calling the `translateToXML()` method on a tree of Java objects. The mechanisms are similar to, yet simpler than those described in detail here. The reason for this is that while names needed to be converted, the Java objects include type information directly. Examples of these calls in Figure 5 finish the RSS translation example.

9. Translation Algorithm

Recursive descent of object trees in either direction.

10. Just In Time Performance Optimizations

<< can re-write this section top-down to prioritize and inventory all current optimization structures >>

As mentioned above, beyond the mapping of XML element and attribute names to Java classes and field names, translation from XML to Java requires knowing what packages contain each Java class. Each `TranslationSpace` includes of a set of entries for optimization, each of which includes class name, tag, and package. When a `TranslationSpace` is created, the parameters include a name, a default package name, and a set of `TagMapEntry`s, each of which includes a Java `Class` object. Class name and XML tag are derived, and stored as part of each entry. These are also used as used as keys in separate `HashMap`s, for fast lookup during translation. Extra mappings can also be specified between tag names and class names. This supports versioning, in cases when one wants to rename a `Class` and its tag, and continue to support old XML with backwards compatibility.

If no entry is found initially, the `TranslationSpace` will look for a class using its default package name, the first time it sees an entry that should correspond to a certain tag name. If this also fails, a special `TagMapEntry` is generated and registered indicating that the tag is unsupported. Thus, the mapping of XML tags to and from class objects can be executed with optimal performance.

Our design goal is to optimize the use of programmer and CPU time. To realize this, the `TagMapEntry` is one of several data structures for performance optimization that are compiled just in time during translation processes. This means that the first time a tag or Java class is processed that it will take extra time, but that all future invocations run as fast as possible. The just in time, or lazy evaluation, derivation of these structures enables the `ecologylab.xml` framework to provide optimal performance without any preprocessor stage that would be a burden to the programmer. An example of this is the `Optimizations` data structure, which is compiled for each `ElementState` subclass that is encountered during translation from XML.

Each `Optimization` consists of a set of `ParseTableEntrys`, one for each field. Each `ParseTableEntry` includes all values needed for fast translation, including the Java reflection `Field` object, the `Class` object referred to, the XML tag, the custom `set` method if there is one, and an integer type indicator. These types include regular attribute, ignored attribute, leaf node, scalar value, regular nested element (specified by field name), collection element, other nested element, or ignored element.

11. Implications and Applications

We are already using `ecologylab.xml` in a slew of component based applications. A libraries of declarations processes many types of popular information semantics. Services and subsystems provide sets of components that, like the framework itself, are general and re-usable. Examples of these include *Distributed Computing Services*, and an *Interaction Logging Subsystem*. Another application, the `ecologylab.studies` suite, has been developed to be of general use for any interactive application developer who conducts user studies. Other applications make use of `ecologylab.xml` to simplify the representation of complex computing semantics for persistent storage and networked communication in ways that are application-specific. These include *combinFormation* and *Rogue Signals*.

11.1 Libraries of Declarations

A library of `ecologylab.xml` declarations for popular semantic web and digital library data sources is provided with the framework. For example we support all dialects of RSS with a single `TranslationSpace` of `ElementState` declarations. These dialects are considered to be mutually incompatible. Additional dialects support for which his already provided include, Dublin Core, Endnote, FeedBurner, Yahoo Web Services, Yahoo Media, iTunes, and the International Children's' Digital Library. By necessity, some of these draw on our support for XML namespaces, the details of which are beyond the scope of this paper.

11.2 Services and Subsystems

11.2.1 Distributed Computing Services

Distributed Computing Services (DCS) extend `ecologylab.xml` to make it very easy for programmers to specify messages, send them, perform operations, and receive results -- across applications. The applications can be running on the same or on different, networked computers. *DCS* is built on a base `Message` class. `Messages` come in two further subtypes: `RequestMessage` and `ResponseMessage`. Both `Messages` are extended by custom subclasses that specify the information that needs to be transmitted to specify an option and return results. These base classes are easily extended with the `ecologylab.xml` machinery in order to send and receive message data structures of arbitrary complexity. Each `RequestMessage` must implement a `performService()`method that actually provides the service, and produces the appropriate `ResponseMessage` result.

When a DCS server is initially instantiated, it is passed an object of type `ObjectRegistry`. This is a scratch pad, in which the keys are Strings, and the values are any reference object. The `ObjectRegistry` may need to provide particular objects, in order for the message processing of `performService()` to function. This enables us to re-use message definitions in different applications, while providing an operating context for message processing consisting of just the objects needed for each application's particular operations.

Each server must also be passed a `TranslationSpace` at initialization time. This controls which message definitions will be used to process requests. It is possible to provide different sets of implementations of the same messages in different application contexts.

Several inter-operable versions of the client and server components have been developed, depending on required deployment capabilities. Communication is based on TCP/IP. The server can be run with a single control thread plus an additional thread per connection, using standard Java I/O, building on the well-known and powerful Java `SeverSocket` abstraction [11]. For higher performance, the there are NIO [13] versions of the server that use a single, two, or *n* threads. The appropriate version for performance-intensive distributed applications may depend on the nature of each application's network utilization as well as the specifics of the hardware on which the server will be running; for example, the *n*-threaded version may work best on a multi-processor machine, while the 2-threaded version may be better with a single core. Since all servers implement the same methods and interoperate with clients in the same way, the exact implementation may be specified at runtime. Clients can be instantiated using standard I/O or NIO base classes, depending on application performance requirements.

11.2.2 Interaction Logging Subsystem (ILS)

Interaction logging is a technique for tracking users' behaviors while using an application [10]. The application is "instrumented" with logging calls throughout. These correspond to significant events, such as the user's operation of a particular control, or, in the case of context-aware and other mixed-initiative systems [4], the system's

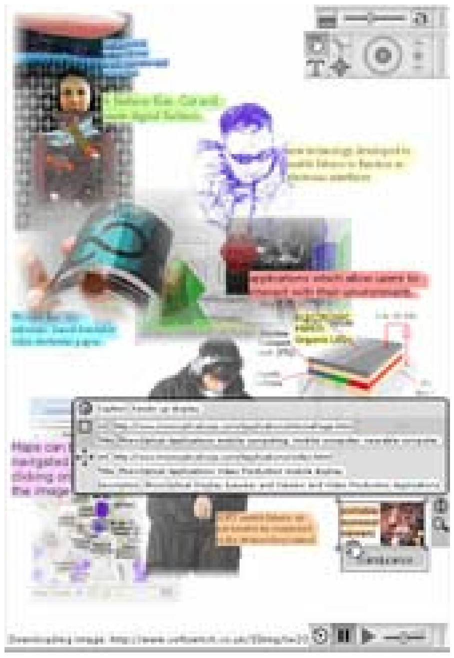

Figure 4: combinFormation composition spaces are stored using `ecologylab.xml`.

automatic initiative of actions. Logs can be used to play back simulations of the user experience, as well as for undo/redo functionality. They can be analyzed to see which features of a program actually get used, and how.

The *Interaction Logging Subsystem* (*ILS*) provides semantics based on a class called `MixedInitiativeOp`, which provides a thin layer of appropriate standard functionality over the `ElementState` base class. The only serialized field in this class is a timestamp. Methods include `performAction()`, which is used for tracking the time when the event occurred, and an `isHuman()` method, which notes whether the action was initiated by a human or by the system. Each application can add whatever particular fields are appropriate to its own operations. In addition to the operations, the application can define a `Prologue` and `Epilogue`, which are written to the log at the beginning and end of the session, respectively. These context-specific state specifications can be formed to provide appropriate semantics by using any `ecologylab.xml` facilities. The same class definitions that are used to generate logs are re-used to read and analyze them.

Three mutually exclusive modes define how the log data is written: standard, memory-mapped, and networked. In standard I/O mode, the data is queued and periodically written to the hard drive by a low-priority thread when processor time becomes available. In memory-mapped mode, the data is written to a memory-mapped file using NIO [13], allowing the virtual memory subsystem to handle when it is written to disk. Again, a separate low priority thread is used for these writes, in order to ensure that log

I/O never introduces latency into user interaction.

Networked ILS provides a logging service to remote clients. A dedicated logging server utilizes DCS to accept client connections. It can be configured to use encrypted password authentication. To initialize the message-passing conversation through a network socket, a client sends a `String` that corresponds to the application name, and a unique identifier (UID) that corresponds to the user. The ILS server uses these to decide the directory and file name to use to write the log data for the session. The client sends batches of log event messages. Eventually, the `Epilogue` accompanies the close message. This version of ILS is integrated with the `ecologylab.studies` suite.

11.2.3 The ecologylab.studies Suite

Developers of interactive (HCI) systems must regularly conduct user studies, in order to validate how these systems perform in the context of human use. This process is intensified on projects like *combinFormation* [6], in which development is sustained over many years. This requires regularly conducting user studies, in order to validate the efficacy and effects of each system, and of components within it. `ecologylab.studies` is a servlet-based suite of components for conducting user studies. Each study is specified using an XML file. Likewise, results for each subject that performs a study are saved as an XML file. These files are specified, written, read, and analyzed using `ecologylab.xml`.

The `study.xml` configuration file contains specifications for user tasks, as well as pre- and post-questionnaires. User tasks can be configured to use Java Web Start [12] to launch an application. The configuration file also defines an abstraction called a `question_path`, which enables automatic counterbalancing of experimental conditions. Counterbalancing is a standard experimental method for determining whether and how experimental factors, including order, affect results.

When a study is conducted, the servlets are used to collect data on experimental subjects, resulting in the generation of study data XML files. In addition, the ILS is integrated, so that a consistent UID that represents each experimental subject is used to write log files as well as study data. Afterwards, or iteratively as a study is conducted, it is necessary to analyze the data that is gathered about subjects' performance. Using the `ecologylab.xml` declaration, it was easy to develop tools that re-use the `study.xml`, study data, and log files. These tools perform a join on these files to generate a single integrated data set. They generate comma separated value (CSV) format files, for import into tools such as Matlab and Microsoft Excel.

11.3 Application Specific

11.3.1 combinFormation

`ecologylab.xml` was first used as a means for storing compositions in *combinFormation*. *combinFormation* uses visual composition of image and text surrogates to represent collections via content-based visualization [6] (see Figure 4). Through mixed-initiatives, agents work together with the user to develop and represent collections. This was shown to support information discovery. `ecologylab.xml` has subsequently been used to provide optimized history processing and logging in this application. The framework has facilitated the iterative growth of the semantics with the application over the past four years.

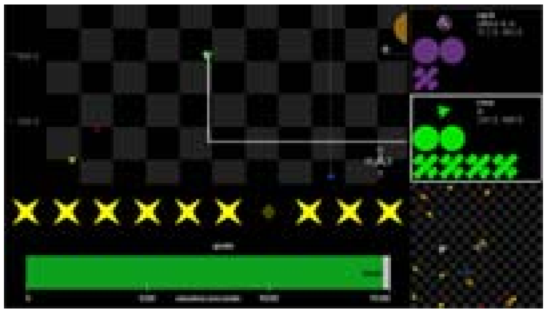

Figure 5: Networked game play in *Rogue Signals* is conducted with `ecologylab.xml`.

11.3.2 Rogue Signals / PhysiRogue

Rogue Signals is a serious multiplayer game for studying team cognition [14], which utilizes `ecologylab.xml` for data storage and transmission. Specifically, base game `Entitys`, representing players, goals, and enemies, extend `ElementState`. These are gathered in collections to build up `GameData` objects, which encapsulate all information about the game state (visual representation in Figure 5) at a given instant.

General classes based upon ecologylab.xml handle data logging and network transmission. For networking, it was necessary to implement a custom server and client, both extended from the DCS, to provide custom packet prioritization. `Messages` encapsulating game information provide the necessary behaviors for play. Finally, for logging, it was necessary only to extend `MixedInitiativeOp` with the `GameData` class to enable automatic logging of events utilizing memory-mapped I/O to ensure high performance.

PhysiRogue is an extension of *Rogue Signals* that uses psychophysiological sensing to enhance game play [15]. It expands *Rogue Signals* to utilize additional data recorded by physiological sensors. Creating *PhysiRogue* was easy: simply extending the relevant game objects to use the extra data. Since the objects already extend `ElementState`, XML translation of the new data for logging and network transmission is handled automatically.

12. Prior Work

This research is inspired by the metaobject protocol, which added powerful introspection features to the Common Lisp Object System [7]. More recently, considerable work has addressed the serialization of Java classes in XML. Several well-known libraries for XML data binding address features, speed, or ease of use. We examine some of the most popular and efficient ones: JAXB, XStream and Castor. There are others. Some require the use of intermediate translators. Most require more intermediate code. None that we are aware of provide nested scopes for composing sets of tag to class translations. As far as we have seen, none provide the expressive power, depth of semantics, the performance, or, most importantly, the ease of use for programmers that ecologylab.xml does. We also investigate another framework, Vinci, which is similar to `ecologylab.xml` but not based on XML.

12.1 JAXB

JAXB is the Java Architecture for XML Binding [9]. XML schemas are the necessary starting point for the operation of an intermediate translator, which compiles each schema into a set of Java class definitions. These, in turn are integrated into the developer's Java program. In contrast to JAXB, `ecologylab.xml` is a framework for XML data binding, in which the primary specification of XML structures is accomplished through the definition of Java objects using the `ElementState` base class. Intermediate languages and translators are avoided.

12.2 XStream

Like `ecologylab.xml`, XStream [17] is a tool that focuses on ease of use by programmers for XML serialization. Thus, XStream is also fairly simple and straightforward to begin using. Unlike most other tools, XStream also has the ability to maintain duplicate references through the use of IDs or direct XPATH references. This allows for complete graph serialization. Unfortunately these simplifications do not translate to performance. The Bindmark project for benchmarking Java XML tools shows that XStream performs only at half the speed of JAXB for large XML data sets [2].

Both XStream and `ecologylab.xml` have very low initial setup costs because they do not require any intermediate class creation or generation. However, `ecologylab.xml` provides serialization of complex types without the need for creating additional converters, which are required in XStream. Further, XStream maintains XML tag to Java class mappings globally; while `ecologylab.xml` uses `TranslationSpaces` that encapsulate these mappings for clearer semantics and better re-use.

12.3 Castor

Castor is a fully featured data-binding framework for XML serialization, Java to SQL and Java to LDAP persistence, as well as the ability to generate classes and/or serializations from bean introspection. In contrast with `ecologylab.xml`, Castor is an extremely complex project that requires a lot of initial work to get running, including the operation of intermediate translators. In fact, Castor is so complex to use that the open-source community of Castor users has created a GUI, called OX2Mapper, to help simplify the mapping of classes to other structures such as XML, RDBMS, and LDAP.

12.4 Vinci

Vinci is a language-neutral mechanism for providing distributed computing services [1]. Vinci declarations are written in an easily understood, type-less XML-like language. Facilities are provided to translate Vinci data into optimized forms for transmission across sockets. A runtime services layer provides sophisticated load balancing of Vinci services across multiple processors.

In comparison with `ecologylab.xml`, the language-neutral aspect of Vinci is an advantage. However, the price is declarations in an intermediate language, which must be compiled into procedural code in languages such as Java. In cases in which Java is the only language being used, this adds an extra burden on programmers in the development process. The recompilation will be necessary each time the message declarations are changed. Also, our type system enables automatic marshalling of Strings to typed values. Another advantage of `ecologylab.xml` in comparison with Vinci is the ability to use it to easily write processors for standard dialects of XML, such as RSS, or for application specific dialects. Finally, it should be noted that Vinci is proprietary and not open source.

12.5 Media Semantics: MPEG 11

Cite: That Obscure Object of Multimedia Desire

13. Results: Performance

We investigated the performance of .`ecologylab.xml` on the task of translating a very large XML document to a typed tree of equivalent Java objects. Each task was performed ten times and the times were averaged. We separated the task into two phases: translating from XML, and translating back from the Java objects to an XML. We compared the performance of `ecologylab.xml` with that of JAXB, because among the Java frameworks we have mentioned, JAXB has the best performance. It should be noted, again, that JAXB pre-compiles Java objects through an intermediate translation process that places an additional burden on programmers in the develop-run-debug iterative development cycle. `ecologylab.xml` was found to be 33% faster than JAXB on roundtrip translation of objects from XML to procedural Java, and back.

This means that `ecologylab.xml` is faster than the other translation frameworks we have mentioned. JAXB is 49% faster than XStream on the roundtrip operation [2]. Thus `ecologylab.xml` is 66% faster than XStream. The comparison with Castor is similar as JAXB was 46% faster than Castor on large XML files.

14. Conclusion

`ecologylab.xml` is notable for its support for rapid development and extensibility. No intermediate translators are required to transform an interface definition language into Java. The mapping and marshalling of types is straightforward and extensible. Powerful and easy-to-use mechanisms for supporting nested sub-collections are available. Intuitive mappings between names promote readability with a minimum of programmer effort. New fields can be added iteratively, as programs develop. `TranslationSpaces` facilitate modular re-use of sets of package to class and tag name mappings. Multiple mappings for a single class can be specified, allowing semantics to change while maintaining backwards compatibility.

While the framework supports rapid development of new, encapsulated XML dialects and web services, it also enables integration with existing dialects and services. Distributed Computing Services enable rapid use of the framework across executables and machines. At the same time, the semantics are flexible enough to conveniently support translation of existing standard XML dialects. For the programmer's convenience, a small library of such dialects is already supplied, including RSS, Dublin Core, ITunes, Yahoo Media, and Yahoo Web Services.

In practice, utilizing `ecologylab.xml` in applications is easy and fast, enabling rapid, iterative development. New attributes and nested elements can easily be added, and `TranslationSpaces` enable the backwards-compatible introduction of new class and tag names. For example, in *Rogue Signals* and *PhysiRogue*, the programmers were able to ignore any issues related to network communication and log recording and playback so that they could focus on the building the system efficiently.

The current research addresses ease of use by programmers without neglecting performance. With regard to performance, translation from Java to XML is already at an acceptable level. Translation from XML is presently a bit slow, due to our use of the DOM parser. We are addressing this by developing a SAX-based [11] version of the translation framework. We expect that performance on this version will compare favorably to all alternatives. We will allow selection of SAX or DOM to be performed at runtime, through a switch similar in nature to the choice of an optimizing C compiler. While SAX will be faster, the DOM provides better validation of XML and error messages during a typical development process. Also, it will enable compatibility with existing programs that use the DOM dynamically, instead of solely as a means for static translation of XML.

15. Future Work

Much remains to be done on this project. We appreciate the support in Vinci for multiple languages. Yet, instead of using a neutral intermediate language, we prefer to continue with Java a primary language for the definition of data to be serialized, stored, and message-passed. We anticipate developing two kinds of new language translators. Java is a "mother language" for `ecologylab.xml`, that is, one in which objects can be defined directly with equivalence to XML. Other languages that support reflection can also function in this capacity. Inter-language translators from any mother language to any other language, as well as XML, can be built. We have begun work on a Java based translator to JavaScript. JavaScript will function only as a secondary language, a target language. This will enable the automatic generation of JavaScript code for equivalent objects, and bi-directional serialization. In turn, we will integrate this with `HttpURLRequest`, to integrate `ecologylab.xml` and its DCS with AJAX [18]. This is the current horizon for future work on this project.

16. References

[1] Agrawal, R., Bayardo, R.J., Gruhl, D., Papadimitriou, S., Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications, *Proc WWW 2001*.

[2] Bindmark Project for benchmarking XML tools. https://bindmark.dev.java.net/

[3] Gosling, J., Joy, B., Steele, G., Bracha, G., *The Java Language Specification, Third Edition,* Boston: Addison Wesley, 2005.

[4] Horvitz, E., Principles of Mixed-Initiative User Interfaces, *Proc CHI 1999*, 159-166.

[5] Interface Ecology Lab, ecologylab.xml, http://ecologylab.cs.tamu.edu/xml/

[6] Kerne, A., Koh, E., Dworaczyk, J., Mistrot, M., 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.

[7] Kiczales, G., des Rivières, J., Bobrow, D.G., *The Art of the Metaobject Protocol*, Cambridge: MIT Press, 1991.

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

[9] Project Glass Fish, JAXB, https://jaxb.dev.java.net/

[10] Preece, J., Rogers, Y., Sharp, H., *Interaction Design: Beyond Human-Computer Interaction,* Wiley, 2002.

[11] Sun Microsystems, Java 2 Platform Standard Edition 5.0 API Specification, http://java.sun.com/j2se/1.5.0/docs/api/

[12] Sun Microsystems, Java Web Start, http://java.sun.com/j2se/1.5.0/docs/guide/javaws/

[13] Sun Microsystems, New I/O APIs, http://java.sun.com/j2se/1.4.2/docs/guide/nio/

[14] Toups, Z. O., Kerne, A., Caruso, D., Devoy, E., Graeber, R., Overby, K. *Rogue Signals*: A location aware game for studying social effects of information bottlenecks, *Proc. Ubicomp 2000 Extended*.

[15] Toups, Z. O., Graeber, R., Kerne, A., Tassinary, L., Berry, S., Overby, K., Johnson, M. A design for using physiological signals to affect team game play. *Proc. Augmented Cognition International 2006*, in press.

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

[17] Walnes, J., XStream, http://xstream.codehaus.org/

[18] Wikipedia, Ajax programming, http://en.wikipedia.org/wiki/Ajax_%28programming%29