Interface Ecology Lab Technical Report 07-01 [DRAFT 2]
Inter-Language Translation Framework for Information Semantics

Type System, In-Context 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 burden of preprocessors that is 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 meta-language, through which details of translation are specified directly in the context of procedural language source code. Translation spaces group together sets of package to class name bindings, and also serve to organize 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 underway. Distributed Computing Services, an Interaction Logging System for user studies, and other reusable component bundles have been built using the translation framework.

Keywords *XML, metalanguage, information semantics, type systems, distributed computing, data binding, 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 [18] 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.

The semantic web, digital libraries, personal repositories such as iTunes, e-commerce, and other application areas utilize XML to store and transmit complex information semantics. XML structures may involve deeply nested trees, with many fields at each level. Nested fields may consist of individual slots with different names, or collections of elements with the same name. 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. Because the framework provides easy-to-program fine control for how serialization is performed, unlike other XML binding software (e.g., [8], [10]), the development of parsers for standard dialects of complex XML, such as RSS and Dublin Core, is simply accomplished through a set of object and meta-language declarations. *Distributed Computing Services* (DCS) enable the rapid construction of messages that specify operations across machines, the remote invocation of these messages, and the receipt of results. The *Interaction Logging Subsystem* records application utilization for user studies. A generalized servlet-based application suite provides more extensive user study facilities. Interactive applications include visual information collecting and a multi-player game.

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 a type system, an in-context meta-language, and intuitive respellings for representing information and media semantics, in order to maximize convenience and expressive power for programmers. The result 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.

We begin by introducing principles embodied in the inter-language type system. We motivate the value of using an in-context meta-language to enable programmers to specify details of how language constructs should be translated to XML with maximum expressive power and minimum work. Maintainability is promoted. We then review the topological semantics of XML, in contrast with those in a procedural programming language (Java) and relate these to information semantics constructs by emphasizing the difference between scalar and element values. Section 4 highlights the utility of our research by introducing an example that will be developed throughout the paper. Section 5 defines constructs of the inter-language type system: intuitive inter-language respelling, rules for type conversion, and `TranslationSpace`s, which group sets of annotated procedural language object definitions together. One `TranslationSpace` can inherit from others, enabling componentization. Section 6 explains the in-context meta-language, and provides details of how translation is performed. The next section describes just-in-time optimizations that make the system efficient and effective in high-performance contexts. Then, we present results in the form of reusable component subsystems for building distributed computing applications and user studies, which are constructed with the translation framework. We also describe specific example applications. Subsequently, we present relevant prior work, and improved performance results. The conclusion and future work emphasize benefits for developing expressive, powerful, and maintainable information semantics.

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) or schema. We completely avoid intermediate interface definition language declarations, like those of CORBA IDL [9], 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, Java declarations possess insufficient expressive power to be used as the sole basis of XML objects. For the sake of readability, or to match standards, there is often a need 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 clearer when the fields that will be translated are declared explicitly. Furthermore, there is a need to explicitly indicate whether a scalar field should be represented as an attribute or a leaf node (to deal with XML "in the wild"). Determining whether a scalar field is text or CDATA has also proved to be an issue.

The solution that increases expressive power, while continuing to maintain all specifications of translation in line with procedural type definitions is an *in-context 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, how.

3. Topology of 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.

```
<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. Abbreviated RSS 2.0 XML Example for The *New York Times* Home Page. In a complete example, many more `<item>` entries would be specified (from http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml).

Entity declarations in XML are gathered together as a *document* [17]. 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. In Figure 1, "`<rss …>`" is an element, while "`version="2.0"`" is an attribute of the `rss` element. The `rss` element also contains the nested element `channel`, which has several nested elements of its own.

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*

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

TranslationSpace RSS_TRANSLATIONS =
       TranslationSpace.get("package.ecologylab.xml.library.rss");
```

Figure 3. A simple Translation Space is defined.

```
ParsedURL NY_TIMES_HOME =
       ParsedURL.getAbsolute("http://www.nytimes.com/services/xml/rss/nyt/HomePage.xml");
```

Figure 3. Defining a web address with the convenience class `ParsedURL`.

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

Figure 4. `Rss`, a subclass of `ElementState`.

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

Figure 5. `Channel`, a subclass of `ArrayListState`, illustrates leaf nodes, a `Collection`, and the `@xml_inherit` directive.

```
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 6. `Item`, a subclass of `ElementState`, represents most of the data from the feed.

```
Rss nyTimesRss = (Rss) ElementState.translateFromXML(RSS_TRANSLATIONS, NY_TIMES_HOME);
```

Figure 7. Inter-language translation framework is invoked to generate procedural language objects.

```
nyTimesRss.savePrettyXML("/temp/nyTimes.xml");
```

Figure 8. Inter-language translation framework is invoked to generate, format, and save XML.

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 [14] available in Java [12] 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 (discussed further, below), 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 schema of entities 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. Note that while the simplified example supports one dialect of RSS, the library of declarations for standard and widely used XML-based languages, which is provided as open source with the framework [5], flexibly supports the panoply of mutually incompatible RSS dialects. The example makes use of a provided helper class, `ParsedURL`, which provides a number of conveniences over the regular URL class.

5. Inter-language 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 scalar translators, such as those for colors and dates, 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.

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 `boolean`. Other 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 `ScalarType`, and register it in an appropriate `TranslationSpace`. The programmer need only override

a few methods in a subclass of `ScalarType`, and provide a default value. The default value for a type is never translated, to make generated XML shorter.

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. Attributes and leaf nodes constitute scalar values. Nested elements are defined using additional subclasses of `ElementState`. Sets of elements of the same tag and type are also supported. Each field to be translated must be declared with an appropriate meta-language annotation.

Translation is accomplished through recursive descent, that is, by depth-first traversal of the element trees. When starting with XML, this is accomplished by traversing the DOM, starting with the root element, using the standard API [17], and generating procedural language object declarations and values for elements and scalars. These values are assigned to appropriate instance variables. This translation from XML to procedural language objects is accomplished by calling the static method `ElementState.translateFromXML(TranslationSpace, ParsedURL)` (see example, Figure 7).

When starting from trees of objects in the procedural language, we start with the root `ElementState`, and use reflection to traverse its fields, and those of nested `ElementState`s. Again, this is a recursive descent depth first tree traversal. At the bottom of the tree, scalar values are generated using the marshalling of the type system. XML is generated to represent everything. The Java `StringBuilder` class is used to avoid heap thrashing and maximize efficiency. Translation from the procedural language to XML is accomplished either with `translateToXML()`, or `savePrettyXML()` (Figure 8).

Figures 2-8 provide a simple example, with explanations, of how to operate on XML from the *New York Times* home page RSS feed example shown in Figure 1. Note that between the last two steps, it would be straightforward to modify news stories, using set and get of fields on `Item`. New `Item`s could also be constructed and filled in this way. Then `channel.add(Item)` would be invoked to add it to a feed. Similarly, `channel.remove(int)` could be called to remove items from the feed, prior to generating XML and saving. It is also straightforward to use these tools to build a tree from scratch, procedurally, and then generate XML.

5.4 Translation Spaces

Sets of type bindings are grouped together into scopes known as `TranslationSpace`s. A translation space defines which `ScalarType`s 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.

Each `TranslationSpace`s is declared with a name and a default package name. Through the name, a global set of name to `TranslationSpace` bindings is provided, so that a static method call enables either construction or retrieval of an instance of a particular `TranslationSpace`. `TranslationSpace`s form nested scopes of bindings. In addition to name and default package name, each `TranslationSpace` declaration can include a single or an array of `TranslationSpace` declarations to inherit from, and an array of Class objects to include. Thus, `TranslationSpace`s can inherit from other `TranslationSpace`s. Multiple-inheritance is supported. This enables modular organization of sets of bindings into `TranslationSpace` components, which can be mixed for re-use, as appropriate. While a default package name 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.

Each Class object, by definition, includes package and class name properties. `TranslationSpace`s can also include declarations for custom scalar type translators, beyond those provided in the distribution. Details are found in the Javadocs [5].

6. In-Context Meta-Language and Translation

Meta-language declarations are made directly in-context with procedural language object declarations. These meta-language declarations are 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 just-in-time translation must recursively collect fields from each

parent class in the chain (see Section 7.). Figure 5 contains an example. `@xml_inherit` is required here, because `ArrayListState` also includes fields that will be translated.

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. Figure 4 shows an example of `@xml_attribute`, while Figures 5 and 6 contain examples of `@xml_leaf`. For leaf nodes only, it is also possible to specify that the value should be represented as CDATA [17], rather than as a plain text node, is specified through an argument, literally in the form: `@xml_leaf(CDATA)`. CDATA declarations enable markup in string literals to avoid further parsing and processing.

Several directives are used to specify the translation of first class nested elements. `@xml_nested` specifies a single nested element. An example shown is the `Channel` field, in Figure 4. `@xml_collection` and `@xml_map` specify sets of elements, in either linear or hashed forms.

6.1 Contextual Translation by Type and by Field Name

We define how inter-language translation of elements and scalar types works. To understand the translation process, it is important to understand 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 in the procedural language*. 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.

6.2 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 (Figure 1), the root element will be of type `Rss`. Figure 4 shows a simplified definition of this class. In this example, nested entities include a scalar value, and a single nested element.

6.3 Translating a Scalar Value Slot

The programmer declares fields in the subclass representing an element, corresponding to each scalar value. Scalar values may be represented either as attributes, like `version` in `Rss`, 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.

6.4 Translating a Nested Element Slot

Often, one declares a single XML element nested inside a parent element. In a DTD or schema, this corresponds to an ENTITY declaration, either unmodified or with the "?" regular expression modifier [17]. 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 defined in Figure 5 is referred to, as a single nested element field, in Figure 4.

6.5 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 [17]. 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 generic parameterized types to enable specification of child elements by tag name, and mapping of types.

6.5.1 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 5. 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`.

6.5.2 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 mechanism is used unconditionally by `ArrayListState`.

6.5.3 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`.

For example, an alternative declaration of the `Channel` element from Figure 5 would be:

```
public @xml_inherit class Channel extends
ElementState
{
   @xml_leaf   String    title;
   @xml_leaf   String    description;
   @xml_collection("item")
               ArrayList<Item> items;
}
```

6.6 Maps: Alternative Set Representation

Sometimes, it is desirable to store set elements using hashing, instead of with 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<T>`, 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`.

We provide an example, again using RSS. Say that we want to store the entries in a hash table, without concurrency control, using the URLs of the article hyperlinks as the keys. First, we must re-define `Item` to implement `Mappable<ParsedURL>`. Next, we redefine `Channel` to store the `<item>`s in a `HashMap<ParsedURL, Item>`. Translation is still invoked as in Figure 7.

```
public class Item extends ElementState
implements Mappable<ParsedURL>
{
   @xml_leaf   String    title;
   @xml_leaf   String    description;
   @xml_leaf   ParsedURL link;
   @xml_leaf   String    author;
   @xml_leaf   Date      pubDate;
   public ParsedURL key()
   {
      return link;
   }
}
public @xml_inherit class Channel extends
ElementState
{
   @xml_leaf   String    title;
   @xml_leaf   String    description;
   @xml_map("item")
            HashMap<ParsedURL, Item> items;
}
```

6.7 Customizing Collections

Often, during recursive descent of a tree, one may wish to do special processing of an element that is constructed, beyond simply adding it to the set that represents that level of the tree. Subclasses simply override the method

```
void createChildHook(ElementState child)
```

to do so.

6.8 Custom Behaviors for Nested Elements

Finer-grained customization is also supported. 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.

7. Just-in-Time Performance Optimizations

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 inter-language translation 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 `ParseTableEntry`s, 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.

8. Results: Components and Applications

The inter-language translation framework serves as a foundation for components, services, subsystems, and applications. Libraries of declarations process popular dialects of information semantics. Services and subsystems provide sets of components that, like the framework itself, are general and re-usable. One of these, *Distributed Computing Services*, enables inter-application message passing and processing of information semantics. Another, the *Interaction Logging Subsystem*, provides flexible logging of user-program interactions for user studies. The `ecologylab.studies` suite builds on logging to provide a powerful set of general-purpose facilities for any interactive application developer conducting user studies. Other applications make use of the inter-language translation framework 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*.

8.1 Libraries of Declarations

A library of inter-language translation framework 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.

8.2 Services and Subsystems

8.2.1 Distributed Computing Services

Distributed Computing Services (DCS) extend the inter-language translation framework to make it 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. `Message` has two subtypes: `RequestMessage` and `ResponseMessage`. Both types are extended by custom subclasses that specify the information that needs to be transmitted, the operations to be performed remotely, and the returned results. These base classes are extended from 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. `ResponseMessage`s have a similar structure to indicate what code will run on the client.

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 `String`s, 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. This means that it is possible to provide different implementations of `performService()` for processing the same messages in different application contexts. This is accomplished by defining different versions of message classes with the same name, but in different packages. A common base class would provide the consistent inter-language data field definitions.

To maximize efficiency and reduce network latency, the DCS utilizes Java NIO [14] over TCP/IP for network communication. Both the client and the server use dual threads, one to handle all network I/O and the other to process incoming messages and send responses.

8.2.2 AJAX-like Framework

An AJAX-like framework has been built on top of the DCS. This enables DHTML code in web pages to communicate dynamically with DCS servers. The framework transmits message XML using either HTTP GET or HTTP POST. In one scenario, the application that functions as a server is a Java Web Start application running on the user's machine, that is, the same machine as the web browser. We call this *client-side middleware*.

8.2.3 Interaction Logging Subsystem

Interaction logging is a technique for tracking users' behaviors while using an application [11]. 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 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) enables local and remote logging of significant user operations in any interactive program. ILS semantics are 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 logging of operations as they occur, 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 inter-language translation framework constructs. 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 [14], 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.

8.2.4 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 the inter-language translation framework.

Figure 9. combinFormation composition spaces are stored using the framework.

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 [13] 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 [11].

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 inter-language translation framework declarations, 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 format files, for import into tools such as Matlab and Microsoft Excel.

8.3 Specific Applications

8.3.1 combinFormation

The inter-language translation framework 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 9). Through mixed-initiatives, agents work together with the user to develop and represent collections. This was shown to support information discovery. The framework 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.

8.3.2 Rogue Signals / PhysiRogue

Rogue Signals is a serious multiplayer game for teaching team cognition [15], which utilizes the framework for data storage and transmission. Specifically, base game `Entity`s, 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 10) at a given instant.

General classes based upon the framework 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. `Message`s 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 real-time performance.

PhysiRogue is an extension of *Rogue Signals* that uses psychophysiological sensing to enhance game play [16]. 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.

9. 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

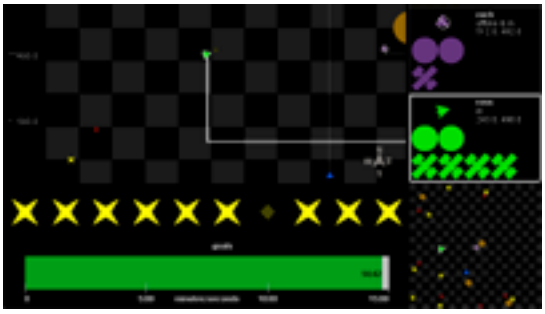

Figure 10. Networked game play in *Rogue Signals* is conducted with the framework and DCS.

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.

9.1 JAXB

JAXB is the Java Architecture for XML Binding [10]. 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. Like JAXB, `ecologylab.xml` can be considered a framework for XML data binding. In contrast, in our framework the primary specification of XML structures is accomplished through the direct definition of procedural objects, with in-context meta-language annotations. Intermediate languages and translators are avoided.

9.2 XStream

Like `ecologylab.xml`, XStream [19] 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. Further, the `ecologylab.xml` in-context meta-language improves maintainability by storing the details of serialization in-line with object field declarations.

9.3 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.

9.4 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.

10. 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.

11. Conclusion

This research develops an inter-language translation framework for information semantics that builds intuitive mappings between XML entities and procedural program objects, while maximizing expressive power and minimizing the burden on the programmer. Through specification via an in-context meta-language, inter-language respelling, and nestable translation spaces, the need for intermediate, development-intensive translation steps is eliminated. Fine control of the details of translations is enabled. The developer is freed from the problems normally associated with object serialization, such as loss of control and difficult software maintenance. Robust support for information objects found in the wild is easily constructed. Extensive customizability of serialization for applications promotes highly readable XML.

The in-context meta-language makes visible the mappings between XML data and procedural object while providing control and promoting maintenance. Powerful and easy-to-use mechanisms for supporting nested sub-collections are available, including the ability to use typed generic `Collections` and `Maps`. 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, enabling the evolution of semantics and 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 processes 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 growing library of such dialects is supplied, already including RSS, Dublin Core, iTunes, Yahoo Media, and Yahoo Web Services.

Furthermore, optimizations make translation to and from XML extremely fast, promoting the development of high performance applications that utilize XML intensively. 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 practically ignore any issues related to network communication, and log recording and playback, so that they could focus on the building a useful and meaningful application.

12. Future Work

We are still working to imbue the inter-language translation framework with expressive power. We have demonstrated initial support for generics, but will also support full-fledged generic declarations, so that XML instances can directly specify type parameters as generic classes are instantiated. Further, we appreciate the support in Vinci for multiple languages. Yet, instead of using a neutral intermediate language, we prefer to continue with procedural languages such as Java as the primary language in which information semantics are specified, in order to be serialized, stored, and message-passed. We anticipate developing two kinds of new inter-language translators: to other mother languages, and to secondary languages. Java is a "mother language" for the inter-language translation framework, that is, one in which information semantics can be defined directly with equivalence to XML. Other languages that support reflection, such as C#, can also function in this capacity. Inter-language information semantics 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 information semantics, and bi-directional serialization from a single master set of object definitions

13. 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] Obasanjo, D., Extreme XML: XML Serialization in the .NET Framework, Microsoft Developers' Network, http://msdn2.microsoft.com/en-us/library/ms950721.aspx, last viewed 3/18/07.

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

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

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

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

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

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

[15] 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*.

[16] 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. *Foundations of Augmented Cognition*. 2nd ed. 2006. 134-139.

[17] W3C, Document Object Model (DOM) Level 1 Specification, http://www.w3.org/TR/REC-DOM-Level-1/.

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

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

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