

# Taming XML: Objects First, Then Markup

Matt Bone, Peter F. Nabicht, Konstantin Laufer and George K. Thiruvathukal  
Emerging Technologies Laboratory  
Department of Computer Science  
Loyola University Chicago  
Chicago, IL 60640  
{mbone,nabicht,gkt,lauffer}@etl.luc.edu

**Abstract**—Processing markup in object-oriented languages often requires the programmer to focus on the objects generating the markup rather than the more pertinent domain objects. The BetterXML framework aims to improve this situation by allowing the programmer to develop a domain-specific object model as usual and later bind this model to preexisting or newly generated markup. To this end, the framework provides two types of object trees, *XElement* and *NaturalXML*, for representing XML documents. *XElement* goes beyond DOM-like automatic parsing of XML by supporting the custom mapping of elements to domain objects; *NaturalXML* allows the mapping of existing domain objects to XML elements using class metadata. Both types of object trees can be inflated and deflated by means of a common intermediate representation in the form of an event stream. Finally, the framework includes the *XML Intermediate Representation (XIR)*, a lossless record-oriented representation of XML documents for efficient streaming and other types of data exchange.

## I. MOTIVATION AND APPROACH

When processing markup in an object-oriented language, we often confront a large amount of accidental complexity. Instead of dealing with our domain objects directly, we are forced to consider event streams or DOM trees. The BetterXML framework aims to mitigate the situation, allowing for the processing of markup without losing focus on the underlying domain model. The design goals of BetterXML are ease of use, simplicity, flexibility, and grounding in design patterns.

The BetterXML framework starts from the assumption that markup is inherently tree-structured. Though this starting point may differ from the original intent of markup as a stream of characters interrupted by the occasional tag, the XML specification does not allow for documents without tags and requires exactly one root element [1]. Thus the specification itself is a recognition of this tree structure. Following this assumption, BetterXML represents markup in the form of an object tree, and the framework contributes two types of these trees, *XElement* and *NaturalXML*, as well as an internal event stream representation and a record-oriented external representation.

*XElement*: While both types of object trees retain a one-to-one relationship between elements and classes, *XElement* is the more DOM-like of the two. In *XElement*, all classes extend a common base class. This base class contains a single heterogeneous list of all children and a map of attributes.

*NaturalXML*: Classes in *NaturalXML*, on the other hand, need not extend a common base class. Instead the mapping between elements, attributes, and child elements occurs in class metadata, and each class is responsible for providing its own data structures for children and attributes.

*Event streams*: Both tree formats in the framework know nothing of the underlying markup they represent; instead, the trees are inflated from and serialized to an event stream. This stream, though not meant for end users, is the lingua franca of the system, serving as an intermediary that decouples the object trees from the various kinds of markup the framework supports.

*XIR*: Though the framework was originally intended for XML processing, it is compatible with other forms of markup including its own representation, the XML Intermediate Representation (XIR), a record-oriented regular language that is easy to parse and for which writing a parser is trivial. XIR also serves as an external representation of event streams.

In this paper we will discuss BetterXML as it relates to the reference Java implementation. The ideas of BetterXML, while closely tied to the object-oriented paradigm, are independent of specific object-oriented languages, and Python and C#.NET ports of the framework are in progress.

### A. A Brief Example

Throughout this paper we will use a simple applicative calculator to illustrate various aspects of the BetterXML framework. The calculator will support integer addition and subtraction, and we will represent these operations in XML. For example, we express the computation  $21 + 20 + (5 - 4)$  as:

```
<group>
  <add>
    <value value="21"/>
    <value value="20"/>
    <subtract>
      <value value="5"/>
      <value value="4"/>
    </subtract>
  </add>
</group>
```

The computation, of course, evaluates to 42. If we were to represent this as an expression tree in an object-oriented language, we would most likely create a distinct class for each operation. Each of these classes might share a common

method, `evaluate()`. Values would evaluate to themselves, and a more complex operation like addition would call the `evaluate()` method on all children and return their sum.

This example is typical of an introductory object-oriented programming or data structures course and is used to illuminate the ideas of polymorphism and recursive evaluation. In such a setting, the example is valued for its conceptual clarity. In the sections below we will show how the BetterXML framework can represent this problem in XML while using the aforementioned object model with little or no modification.

## II. OBJECT TREES

In this section, we discuss the two object tree formats provided by BetterXML in more detail.

### A. XElement

Of the two tree formats in BetterXML, XElement is the more DOM-like. As in DOM, XElement classes must extend a common base class, and this base class contains the data structures and methods for manipulating children, attributes, and character data. DOM, however, is not designed for the programmer to create element-specific or other custom subclasses. By contrast, XElement allows the programmer to map specific XML elements to one particular subclass of the common base class, thereby creating a domain-specific object tree.

For instance, in the calculator example we map the `add`, `subtract`, `group`, and `value` elements to the `Add`, `Subtract`, `Group`, and `Value` classes respectively:

```
ToXElementContentHandler handler =
    new ToXElementContentHandler();
handler.registerElementClass(Add.class, "add");
handler.registerElementClass(Subtract.class,
    "subtract");
handler.registerElementClass(Group.class, "group");
handler.registerElementClass(Value.class, "value");
```

Here, `handler` is a reference to the event handler for the BetterXML event stream (see section III), and this handler is used to establish the mappings from element names to classes prior to parsing a document. When the document is parsed and an element is encountered, the mappings are examined. If the element is mapped to a particular class, then that class is instantiated. Otherwise, the base class, XElement, is instantiated.

Since all unmapped elements are represented as instances of XElement, the mappings from element names to classes are optional. Thus, smaller applications that lack a rich domain model can rely directly on instances of XElement. The XElement class itself is straightforward and has methods for extracting the element name, attributes, and children. Some methods of this class are excerpted below. (The XAttributes class represents the attributes of an element as a map.)

```
String getName() //element name
String setName(String name)
XAttributes getAttributes()
List<XElement> getChildrenElements()
List<XElement> getChildrenElements(String elemName)
```

We can see the methods that allow us to get at the children elements in action by examining `evaluate()` in the `Add` class:

```
public class Add extends XElement
implements Expression {
public int evaluate() {
int result=0;
for(XElement elem: super.getChildrenElements()) {
if (elem instanceof Expression) {
result += ((Expression) elem).evaluate();
}
}
return result;
}
}
```

This method iterates through each child element and casts each reference to an `Expression`, an interface that requires the `evaluate()` method and is implemented by all calculator classes. Here we know ahead of time that we have mapped all elements in the XML document to classes implementing this interface, so the type cast will not fail. Nevertheless, this approach can lead to problems if we do not know all of the elements in the XML document when we perform the compile-time mapping. If this is the case, there must be explicit checks or appropriate exception handling if the parsing is to run to completion.

The actual parsing of the XML document and retrieval of the object tree is facilitated by a utility class:

```
Reader reader = new FileReader("expressions.xml");
XDocument document =
    ParserUtil.getXElementFromXml(reader, handler);
Expression expr =
    (Expression) document.getRootElement();
```

This `getXElementFromXml` method accepts the previously mentioned event handler (which contains mapping information), and a Java Reader (implementations of which can read from files, strings, etc). The returned XDocument reference wraps the object tree, and the root can be requested and type-cast if necessary.

### B. NaturalXML

Unlike DOM or XElement, NaturalXML does not require objects to extend a common base class. Instead, objects are written as usual, with their own class hierarchy if necessary, and metadata embedded within the class describes the relationship to the markup. In the reference Java implementation, this metadata is implemented with field and class annotations.

Returning to the calculator example, the `Add` class in NaturalXML is expressed as follows:

```
@Element("add")
public class Add extends ContainsExpressionList {
public int evaluate() {
int sum = 0;
for(Expression expr: super.expressions) {
sum += expr.evaluate();
}
return sum;
}
}
```

Recalling that there is a one-to-one relationship between classes and elements, we see that that the `@Element` annotation on the class itself specifies this mapping. Thus when a document is parsed the `Add` class will be instantiated whenever an

add element is encountered. The logic in the `evaluate` method is the same as in the `XElement` version, but there are no type casts. This is because the `expressions` instance variable is declared in the superclass with extra type information (via generics) that makes the casts unnecessary. This super class is not part of the framework. Rather, it contains the code common to all operations in the calculator example. Looking at the abstract super class:

```
public abstract class ContainsExpressionList
    implements Expression {

    @Children({Group.class, Add.class,
              Subtract.class, Value.class})
    protected List<Expression> expressions =
        new ArrayList<Expression>();

    public List<Expression> getExpressions() {
        return expressions;
    }

    public void addExpression(Expression expr) {
        expressions.add(expr);
    }
}
```

We can see this extra type information and the annotation on the `expressions` field. Elements mapped to the classes listed in the `@Children` annotation are all stored in the list, and the type of the list specifies that each of these classes must also be an `Expression`. In the current version of NaturalXML this check is delayed until run time, but annotations can be examined at compile time, and this is one possible area of future improvement. Still, this approach is not only safer but also more concise.

1) *Annotations*: Six annotations form the basis of the NaturalXML system, providing the mechanism for binding elements to classes and fields to attributes, child elements, and character data. Each annotation is discussed below in detail:

```
@Element("element_name")
```

This is the only class-level annotation in the framework, and it defines the one-to-one mapping between classes and element names. This element name is the annotation's only parameter and is required.

```
@Children({SubElement1.class, SubElement2.class})
```

This field-level annotation describes an instance variable that is usually a collection of references to child element data (i.e. instances of the classes in the annotation's parameter list). Each class in the annotation's parameter list must contain an `@Element` annotation and is thus mapped to some markup element. The object hierarchy mimics the element hierarchy; when the elements mapped to the classes in the parameter list are encountered (as sub-elements of the element to which the enclosing class is mapped) they are instantiated and populated. The mutator for the field is then called with this inflated sub-element object as a parameter. The type of the field being described is not important (but usually a subclass of `java.util.Collection`). NaturalXML uses the name of the field to locate the appropriate accessor and mutator methods via reflection. The name of the accessor follows the Java Beans

specification [2]. For example, with a field named `children`, NaturalXML looks for an accessor named `getChildren()`. The name of the mutator is the singular form of the field name prepended with "add"; with this same field, NaturalXML looks for the mutator `addChild(...)`. A port of the Ruby on Rails Inflector [3] provides the singularization capability.

```
@Singleton
```

This field-level annotation may occur only on elements already annotated with `@Children`. It signifies that one and only one child is expected and should annotate a reference to that child as opposed to a collection (as in a standard `@Children` annotation). Both the accessor and mutator (i.e. "get" and "set" methods) are then discovered as per the Java Beans specification [2].

```
@Attribute("attribute_name")
```

This field-level annotation describes instance variables of type `String`. This field contains the value of the attribute specified in the parameter. Both the accessor and mutator are then discovered as per the Java Beans specification [2].

```
@CDATA
```

This field-level annotation describes an instance variables of type `List<CDATAWrap>` and contains the character data of the element mapped to the enclosing class. Elements with character data may provide only one field with this annotation. The `CDATAWrap` class is a string wrapper provided for convenience with methods such as reducing a list of `CDATAWrap` instances to one string. It also provides a hook for future expansion.

```
@Namespace("http://namespace.uri")
```

This annotation is applied at the class or field level to specify a namespace for elements or attributes. It cannot stand alone.

2) *Usage*: The NaturalXML object tree is ideally suited to situations where the underlying data model *or* the underlying markup may change frequently. This may occur, for example, in exploratory programming, prototyping, or the creation of ad-hoc data formats. Similarly, NaturalXML is useful when an existing data model needs to be expressed in markup. As shown in the calculator example, the changes are simple, requiring little more than the insertion of the appropriate annotations into an existing class hierarchy.

Notice, however, that some of the burden does fall on the programmer in NaturalXML; every piece of the markup must be represented in some data structure lest it not be preserved. In situations where the programmer is only interested in manipulating a small subset of the data stored in the markup, this can be tedious and `XElement` would be the preferable tree format.

3) *Limitations*: A limitation of NaturalXML in its current form is its inability to retain ordering information among interspersed elements that are mapped to different data structures. As an example, consider the XML fragment:

```
<root>
  <a num="1"/>
  <b num="2"/>
  <a num="3"/>
</root>
```

Assume we map the `root` element to the `Root` class, the `b` element to the `B` class, and the `a` element to the `A` class. If in the `Root` class we store instances of `A` and `B` in distinct lists:

```
@Element("root")
public class Root {
    @Children(A.class)
    List<A> aList;

    @Children(B.class)
    List<B> bList;
}
```

Then the ordering of the `a` elements relative to the `b` elements will not be preserved. That is, all the `a` elements will be printed before the `b` elements. If we were to output the NaturalXML tree to XML we would get:

```
<root>
  <a num="1"/>
  <a num="3"/>
  <b num="2"/>
</root>
```

While all element, attribute, and character data information is preserved, the ordering information of elements mapped to different data structures is not. While this limitation is usually not an issue for the ad-hoc data formats to which NaturalXML is ideally suited, the limitation does make it difficult to represent documents like HTML where such ordering information is often necessary and one would like to store instances of the objects mapped to these elements in disparate data structures. If this is a requirement, it may be best to use XElement, but we plan to alleviate this problem in the near future by using an order-preserving map implementation for the children.

### III. EVENT STREAMS

Both of the object trees discussed in Section II are unaware of the underlying markup. Rather, NaturalXML and XElement trees are created from a stream of events. Likewise, each is serialized to the same stream of events. This approach decouples the tree formats from the various input and output formats, much like intermediate representations decouple the front end of a compiler from the back end. As in the compiler world where  $n$  front ends and  $m$  back ends lead to  $n * m$  possible configurations, the same holds in the BetterXML framework for tree and markup formats.

Internally, the event stream is a sequence of method calls and can be externalized using the XML Intermediate Representation described in Section IV. On the input side, each tree format provides an implementation of the the `BetterXmlContentHandler` interface (all methods are **public**):

```
void startDocument()
void endDocument()
void startPrefixMapping(String prefix, String uri)
void endPrefixMapping(String prefix)
void startElement(String uri, String name,
    String qname, int attrCount)
void endElement(String uri, String name,
    String qname)
void attribute(String uri, String name,
    String qname, String value)
void characters(int length, String cdata)
void whitespace(int length, String cdata)
```

```
void skippedEntity(String name)
void processingInstruction(String name,
    String target)
```

For example, the `NaturalXmlContentHandler` is an implementation of this interface and creates a NaturalXML object tree. When the `startElement()` method is called, an instance of the appropriate class (i.e. the class mapped to that element name) is created and pushed onto a stack. Future method calls adding children, attributes, or character data are then able to find the parent class by looking at the top of the stack. After processing, the handler provides a method to retrieve the root of the NaturalXML object tree, which sits at the bottom of the stack.

Both tree formats are serialized to this same event stream. In XElement, the logic is embedded within the base class via the `acceptContentHandler()` method. This implementation of the visitor pattern [4] lets the tree accept and serialize itself to any implementation of the aforementioned `BetterXmlContentHandler` interface. NaturalXML trees are serialized by traversing the tree externally with simple recursive method that takes the tree and a handler as a parameter. This approach is necessary as users of NaturalXML should not be expected to embed traversal logic in their objects.

In practice, the `ToXMLContentHandler`, which generates an XML document from an event stream, is the most common event handler for serialization. However, we can hand any handler to the traversal logic, and the flexibility allows us to generate other markup such as XIR or do more unusual things such as generating another object tree or a SAX event stream.

### IV. XIR

The XML Intermediate Representation (XIR) provides a structured intermediate form of an XML document's content for efficient streaming and other types of data exchange. XIR also serves as an externalized representation of a document's BetterXML event stream discussed in Section III.

While XIR aims to be somewhat human readable, the important distinction is that the format is not tree-structured but rather a record-oriented regular language. Therefore, even though the resulting linear representation appears less concise than the corresponding XML tree, XIR is fast to process and interpret (similar to the byte-code concept found in Java and elsewhere). The simplicity of the format also makes the implementation of parsers trivial; it requires nothing more than the ability to read colon-delimited text and the encoding and decoding of Base64 data. Thus the format can be easily ported to any language or platform.

As an example, consider the XML fragment below:

```
<value value="5">some cdata</value>
```

This fragment is encoded in XIR as:

```
xir.type:verbatim=element
ns:verbatim=
xir.subtype:verbatim=begin
qname:verbatim=value
name:verbatim=value
attributes:verbatim=1
```

```
xir.type:verbatim=attribute
ns:verbatim=
xir.subtype:verbatim=none
qname:verbatim=value
name:verbatim=value
value:verbatim=5
```

```
xir.type:verbatim=characters
cdata:base64=c29tZSBjZGF0YQ==
xir.subtype:verbatim=none
length:verbatim=10
```

```
xir.type:verbatim=element
ns:verbatim=
xir.subtype:verbatim=end
qname:verbatim=value
name:verbatim=value
```

The start of the element, its attribute, the character data, and the end of the element are represented as distinct records. Some records (i.e. those representing elements or prefix mappings) have a beginning and end so as to nest other records inside their scope. Others (i.e. attributes, whitespace and cdata) are “singletons” with no nested scope. This behavior is specified in the `xir.subtype` field.

Notice also that each field contains a “verbatim” or “base64” identifier which specifies whether or not the value of the field is encoded as plain text or in Base64. In general, only character data is encoded in Base64; this allows for compactness and maintains the record-oriented format and ease of parsing by suppressing (but losslessly retaining) newlines and whitespace. Ordering of fields within a record is not specified nor should it be relied upon as the fields are usually represented in memory by an unordered data structure such as a hash table.

## V. RELATED WORK

There is a wide range of related work, of which we attempt to describe the most relevant and closely related.

Our work most closely overlaps with work on *data binding*. In particular, we focus on binding XML data to application-specific data structures. In object-oriented programming, such data structures are built using (hierarchically arranged) classes. There are a number of efforts to address data binding as defined in this way, and we briefly summarize a few of these.

One of the popular approaches is the JDOM (Java Document Object Model) [5], which is effectively an implementation of the standard World Wide Web (W3) Consortium DOM implementation. While written in Java, the JDOM does not support direct binding of elements and attributes to classes and instance variables as done in the BetterXML framework. XElement supports the *essence* of W3’s DOM (while choosing not to be fully compatible) and in its default implementation works exactly like JDOM. Just as JDOM builds an object tree of JDOM Element instances, so, too, will XElement build a tree of XElement instances, should the programmer choose not to custom-map element names to classes.

JAXB [6] is a Java framework that goes from XML schemas to class hierarchies. BetterXML does not start at the level of schemas, and we presently do not support code generation

from a schema. We take the view that code generation systems are only effective if the underlying framework allows the generated code to be modified and can pick up any subsequent changes that are made to the schema. The JAXB team must be well-aware of the problem as their latest effort includes a mechanism based on reflection and annotations—ideas that have been in our implementation since [7]. Nevertheless, we share a number of goals with the JAXB effort and will be using the framework itself to develop a code generator that goes from RELAX NG [8] schemas to NaturalXML.

JAXB’s reflection API [9] is actually borrowed (almost wholesale) from the Microsoft .NET framework [10], which makes direct use of annotations `XmlAttribute` and `XmlElement` to identify elements and attributes. As shown above, NaturalXML provides full support for “all things XML” and allows programmers to map namespaces and address how character data should be mapped to and from XML. NaturalXML also works clearly with classes related by inheritance. Despite its simplicity, the expression language from our example in Section I is typical for the widespread use of recursion in real XML languages and the need to map to object oriented class hierarchies. Because most data-binding frameworks fail to deliver support for this most fundamental and common usage, many XML programmers resort to writing their own parser *handlers* using SAX to avoid the headaches. We believe our framework is among the first to support recursive mapping properly.

Our work on an XML Intermediate Representation (XIR) is partially covered by the SGML/ESIS (Element Structure, Information Set) work of James Clark [11], which aims to have a non-SGML formulation of any SGML document. SGML is a predecessor to XML that fundamentally differed in its lack of support for namespaces and its more ambiguous structure. It was also hard to parse correctly because closing an element was optional (much the same as in HTML version 4 and earlier). The Pyxie framework [12] was an attempt to resurrect many of ESIS ideas in XML but does not appear to be an actively maintained effort. In addition, XIR goes well beyond the ideas by supporting all XML features and ensuring that they are encoded losslessly. Our ability to encode record fields in verbatim or Base64 mode is a competitive advantage and allows us to guarantee that character data and international text are encoded in a truly printable and portable format. All told, we consider ESIS and Pyxie as important inspiration; our work is a refinement that we believe makes it possible to embed XML in any programming environment, especially when the full power of an XML parser is neither desired nor needed.

It must be said that SAX (Simple API for XML Processing) is a significant inspiration for our work. SAX by its nature focuses on the essence of XML parsing and fulfills many of the same goals as a front end does for a language compiler. SAX influenced our ideas on XIR, which can be thought of as a record-oriented format that supports the SAX event stream. The only difference is that (unlike SAX) XIR allows the events to be replayed multiple times if necessary. Furthermore,

because XIR can itself be generated, we envision a day when many applications could be written with XML *in mind* but not necessarily *in sight*. While our benchmarking efforts and optimization are still under way, XIR requires fewer resources to process than most XML parsers, and the stateless design makes it an ideal match for distributed systems, which are increasingly relying upon XML/RPC schemes. While beyond the scope of the present work, XIR can stand alone as a separate serialization/deserialization system to build a robust and efficient RPC system with all of the benefits of full XML/RPC minus the significant overhead of building temporary data structures.

Another library for serialization of Java objects to XML and back designed with simplicity and ease of use in mind is XStream [13]. XStream uses reflection to map Java classes to XML elements automatically; in addition, it supports aliases and other custom mappings, as well as annotations. Nevertheless, the crucial difference between serialization libraries such as XStream and more general data binding frameworks, including BetterXML, is the ability of the latter to start with existing XML documents as opposed to a Java class hierarchy.

We have presented an earlier version of this work in a non peer-reviewed department column on scientific programming [7], where we presented an overview of the use of reflection and POJOs (plain old Java objects). Our work here is a synthesis of many frameworks and is part of an evolving vision to build one of the most intuitive and efficient XML processing systems.

Finally, we know that the world is beginning to question how and when XML should be used. YAML [14] is an approach that builds on the notion of *property files* and is as simple to parse as XIR. We believe this model is mostly equivalent to XML but lacks support for *schemas*, which we believe will ultimately limit its success in serious application development. JavaScript Object Notation (JSON) [15] is also gaining popularity in the new world of web development (affectionately known as Web 2.0). It is used primarily in handling `XmlHttpRequest()` calls but is used widely for initializing state for plugins and other JavaScript data structures. While we believe JSON to be a good approach for this purpose, the inability to do meaningful type checking and schema validation will ultimately limit its use outside of web applications. We mention these alternatives to make it clear that there are alternatives to XML that, while tempting, are not as robust. We are hoping our work will allow developers to reconsider XML. When done right, XML can be lightweight and straightforward as these competing approaches.

## VI. ANALYSIS AND CONCLUSION

The BetterXML framework allows programmers to focus on their domain objects, and the decoupling of input and output formats from the object tree interface via an internal event stream provides room for future growth if necessary. Though not suited for all uses, NaturalXML is ideal for ad-hoc data formats, and XElement performs well whenever a DOM-like tree is desired. In addition, XIR provides a record-oriented

representation of XML documents for efficient streaming and other forms of data exchange. We hope this work represents a useful step forward in the processing of markup.

The Java reference implementation of BetterXML, including the examples shown in this paper, is available as an open-source project through Google Code at <http://code.google.com/p/betterxml>

## REFERENCES

- [1] "Extensible Markup Language (XML) 1.0 (Fourth Edition)," 2006. [Online]. Available: <http://www.w3.org/TR/2006/REC-xml-20060816/>
- [2] G. Hamilton, Ed., *JavaBeans*. Sun Microsystems, 1997. [Online]. Available: <http://java.sun.com/products/javabeans/docs/spec.html>
- [3] "Module: Inflector," 2008. [Online]. Available: <http://api.rubyonrails.org/classes/Inflector.html>
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [5] "JDOM." [Online]. Available: <http://www.jdom.org/>
- [6] E. Ort and B. Mehta, "Java Architecture for XML Binding," March 2003. [Online]. Available: <http://java.sun.com/developer/technicalArticles/WebServices/jaxb/>
- [7] G. K. Thiruvathukal and K. Laufer, "Natural XML for Data Binding, Processing, and Persistence," *Computing in Science and Engg.*, vol. 6, no. 2, pp. 86–92, 2004.
- [8] J. Clark and M. Murata, "RELAX NG Specification," 2001. [Online]. Available: <http://relaxng.org/spec-20011203.html>
- [9] "JAXB 2.0 Annotation Parsing Library." [Online]. Available: <https://jaxb2-reflection.dev.java.net/>
- [10] "XML Developer Center." [Online]. Available: <http://msdn2.microsoft.com/en-us/xml/default.aspx>
- [11] "ISO 8879," International Organization for Standardization, Tech. Rep., 1986.
- [12] S. McGrath, "Pyxie," *O'Reilly XML.com*, March 2000. [Online]. Available: <http://www.xml.com/pub/a/2000/03/15/feature/>
- [13] "XStream," 2008. [Online]. Available: <http://xstream.codehaus.org/>
- [14] O. Ben-Kiki, C. Evans, and I. dot Net, "YAML Aint Markup Language (YAML) Version 1.1," *yaml.org*, Tech. Rep., 2005. [Online]. Available: <http://www.yaml.org/spec/1.1/>
- [15] D. Crockford, "The Application/JSON Media Type for JavaScript Object Notation (JSON)," 2006. [Online]. Available: <http://www.ietf.org/rfc/rfc4627.txt?number=4627>