Enhancing Reuse via Inheritance in XML

Giovanni A. Modica
Department of Computer Science
Mississippi State University
gmodica@cs.msstate.edu

Hasan M. Jamil
Department of Computer Science
Mississippi State University
jamil@cs.msstate.edu

ABSTRACT

While the idea of extending XML to include object oriented features has been gaining popularity in general, the potential of inheritance in document design and ontology management has not been well recognized in contemporary research. In this paper we demonstrate that XML with *dynamic* inheritance aids better document designs and decreased management overheads and support increased autonomy. We present an object oriented extension to the language of XML to include dynamic inheritance and describe a middle layer that implements our system.

1. INTRODUCTION

The reliance on XML as the *defacto* standard for data representation and electronic data interchange has motivated both academic research endeavors and industrial developments. These capabilities have even been exploited in areas such as ontologies and phyloinformatics in addition to their use in document databases. XML has also played major roles in creating new languages. Its popularity has led to an increasing number of standards that have been created using XML as the meta language specification. While the representation and electronic interchange capabilities of XML have been used and investigated quite well, the "re-use" aspect of XML remains rather neglected. Traditionally, *inheritance* has been the key to re-use and express specificity in specialization/generalization hierarchies. While there have been some efforts in incorporating the so called object oriented features in XML, nothing substantial has been reported in XML standards.

In this paper, we propose an extended XML, called the *XML*$^{++}$ to include a subset of object-oriented (OO) features such as dynamic inheritance, encapsulation, and methods. Our contributions parallel other existing extensions to XML such as SOX [3] but as opposed to SOX, we extend the XML data model itself while extensions such as SOX are orthogonal to XML. By that we mean that these extensions exploit object oriented concepts for schema validation only but fail to extend to document levels. As a consequence, they have little to do with document design and structuring. Our extensions will allow re-using existing documents in other remote documents by treating them as class templates. In this approach, documents will inherit features from other documents autonomously and without intrusion.

2. RELATED RESEARCH

Some recent research have addressed the issue of object-orientation in XML documents in the form of proposals and notes to the W3 Consortium. We will discuss only two such major proposals: XML Schema [4] and SOX [3] for the sake of brevity. We have adapted some of these concepts in XML^{++} with substantial modifications that strengthens the concepts in XML Schema and SOX.

WWW2002, May 7–11, 2002, Honolulu, Hawaii, USA.
ACM 1-58113-449-5/02/0005.

XML Schema [4] is an approved recommendation from the W3 Consortium. Designed to overcome the limitations of DTDs, schemas are used to define the structure, contents and semantics of XML documents. XML Schema is divided in two parts: *XML Schema Structure* to specify the structure and constraints of XML documents, and *XML Schema Datatypes* used to represent and validate complex data types. While XML Schema features some OO capabilities it cannot be regarded as fully object-oriented as it lacks many essential features including methods, polymorphism, encapsulation, etc.

SOX, Schema for Object-Oriented XML [3], is a meta-grammar which extends the XML DTDs by supporting OO capabilities as follows: (i) data types, (ii) inheritance, (iii) namespaces, (iv) polymorphism, (v) embedded documentation and (vi) distributed schema management. Although SOX provides a great deal of Object-Oriented capabilities for XML documents, these capabilities are defined from the point of view of XML elements, and not from the point of view of XML documents as a whole. This is where XML^{++} comes to our aid and introduces a new set of features that enrich the SOX language for better document design.

3. XML^{++} OVERVIEW

In this section, we present the XML^{++} language specification. The XML^{++} specification describes XML documents to build XML^{++} documents, and we use the same XML language to build an extension to it. XML^{++} documents are described based on an XML DTD or Schema which constrains the language syntax and the semantics of XML^{++}

Classes: object classes are declared using the `element` tag. XML^{++} object classes can contain attributes (using the `attribute` tag), simple text or XML fragments (using the `elementContent` tag), or subclasses which definition is nested within the parent class.

Inheritance: Inheritance is supported by allowing object classes to extend other classes. The `element` tag uses an optional `extends` attribute which is used to specify the object class for which it inherits the contents. The inheritance is based on template definitions (which are XML^{++} documents) by specifying the template prefix and the superclass name in the template. Superclasses in templates must have a unique name in the template scope, so they can be identified in the subclass document.

It is important to note here that dynamic inheritance is in the heart of our extension to current XML standards. As opposed to static inheritance, a run decision is needed to resolve conflicts, and to inherit properties and methods. Such an approach consequently supports currency, increased autonomy of class design and decreased management overhead, features that are considered essential for distributed XML documents as we highlight in this paper.

Support for methods: XML^{++} elements can support the use of methods in their declarations. The implementation of the method must be externally defined using a pro-

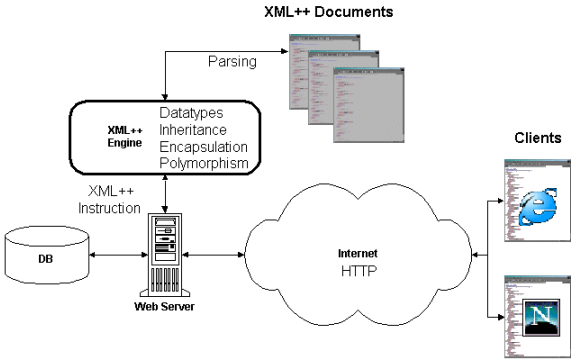

Figure 1: System Framework.

gramming language supported by the XML^{++} engine. Currently, the only programming language supported is the Java programming language, however, different implementations of the engine can add support for other common languages like C++, Perl, etc. XML^{++} engines can be extended using language plug-ins to support different programming languages for method execution. Plug-ins must support some sort of *Introspection*, as defined for the Java language [7]. By using introspection, the executioner module is able to query at runtime for methods interfaces defined inside the class library.

Methods can use two types of parameters: input parameters and return parameters, which are used to specify return values for functions. Parameters are identified by a name and a value. At method invocation, all the input parameters must define either a value or an XPath [1] expression for the attribute value. The use of XPath expressions as parameter values is very useful for dynamic content creation. During declaration time, parameters are required to be listed in the same order as defined in the method definition, in other words, parameter specification is position dependent. On the other hand, during method invocation the use of parameter is position independent, meaning that method parameters can be listed in any order as long as the name of the parameter is used to specify its value.

4. XML^{++} ENGINE

We now present a few highlights of XML^{++} middle-tier engine, its architecture and design. Our implementation follows the multi-tiered MVC (Model-View-Controller) programming model for web applications development advocated in [6]. Although the XML^{++} engine is implemented using Java Enterprise Edition (J2EE) [2] platform specifications, its design and implementation can be applied to similar platforms, like for example the Microsoft .NET architecture. The framework, depicted in figure 1, shows a typical three-tier implementation of a web application. The engine is implemented in the middle-tier as an extension to the web server, which will be configured to forward requests for `.xml` and `.xpp` documents containing XML^{++} instructions to the engine, which after the appropriate parsing and loading of the XML^{++} document templates, will solve all the Object-Oriented instructions of document requested.

The engine was developed using the Java programming language due to its Object-Oriented capabilities and support for introspection [7]. Object-Oriented concepts implemented in XML^{++} are based in this language. All the different components of the engine are developed using Java Servlets and JavaServer Pages in a J2EE application. Each Object-Oriented feature (i.e. data types, inheritance, etc.) is handled by a different module. The XML^{++} parser is responsible of loading the appropriate module depending on the type of Object-Oriented instruction being parsed. Using this modular approach allows for the implementation of new Object-Oriented features in future versions of the engine, without compromising existing ones. The XML^{++} servlet is the controller for the engine. It is registered to process every request with an `.xml` or `.xpp` files.

Before the file is delivered to the client response object, the requested file is retrieved from the engine web context and passed to the parser. If the file defines templates, each template is treated as a new request. Once all the templates are resolved and parsed, the requested XML^{++} document is returned to the client. If an error occurs (like a parsing error) the application forwards the control to a predefined error page, which is responsible for showing an appropriate error message to the client. Templates can be resolved either in the engine context (local to the web server where the engine is installed) or to an external context through the Internet (e.g. templates contained in other XML^{++} engines). In order to parse XML^{++} documents, JDOM and Xerces [5, 6] are used.

5. CONCLUSION

In this paper, we presented the idea of structured document design in an autonomous world using the concept of dynamic inheritance. To this end we have proposed several object-oriented extensions to XML. We have presented some of the XML^{++} specifications to handle object definitions, inheritance, support of methods, and encapsulation. We presented the specification DTD, against which all XML^{++} document should be validated. An overview of the implementation details for the XML^{++} engine, as well as a general framework for the design of the engine, has also been introduced.

Our future work is geared towards the implementation of additional features (encapsulation, polymorphism, etc.) for the XML^{++} specification and for the enhancement of the XML^{++} middle-tier engine. An extended version of this paper with additional details, including illustrative examples and implementation, may be found in the authors' home page at `www.cs.msstate.edu/~gmodica`.

6. REFERENCES

[1] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Simon. Xml path language (xpath) 2.0. http://www.w3.org/TR/xpath20/, December 2001. W3C Working Draft 20.

[2] S. Bodoff, D. Green, E. Jendrock, M. Pawlan, and B. Steams. The j2ee tutorial. Technical report, Sun Microsystems, 2001.

[3] A. Davidson, M. Fuchs, M. Hedin, M. Jain, J. Koistinen, C. Lloyd, M. Maloney, and K. Schwarzhof. Schema for object-oriented xml 2.0. http://www.w3.org/1999/07/NOTE-SOX-19990730, July 1999. W3C Note 30.

[4] D. C. Fallside. Xml schema part 0: Primer. http://www.w3.org/TR/xmlschema-0, May 2001. W3C Recommendation.

[5] J. Hunter and B. McLaughlin. Jdom. http://www.jdom.org, January 2002.

[6] N. Kassem and the Enterprise Team. Designing enterprise applications with java 2 platform, enterprise edition. Technical report, Sun Microsystems, October 2001. Version 1.0.1.

[7] P. Naughton and H. Schildt. *The Complete Java Reference: Java 2.* Osborne/McGraw-Hill, Berkeley, California, 3rd. edition, 1999.