Object Interoperability for Geospatial Applications: A Case Study*

Isabel F. Cruz† and Paul W. Calnan‡
Department of Computer Science
University of Illinois
Chicago, IL 60607-7053, USA

Abstract

In this paper, we analyze a geospatial application for visualizing U.S. election results in order to show the problems that need to be solved in the mapping between different XML representations and their conceptual models. We propose a framework that provides a number of core classes that allow applications to treat XML documents as graphs and to evaluate XPath expressions against such document graphs. We also propose a mechanism that allows information to be exchanged between document types. Our goal is to ultimately attain an overall framework for interoperation where maintenance problems are minimized. To achieve this, we anticipate the need for introducing metadata in the semantic layer that will guide the translation process between document types.

Keywords: Geospatial applications, user interface backend, object interoperability, technologies for interoperability.

1 Introduction

In statewide geospatial applications, hundreds of systems need to be integrated. In these applications, challenges in achieving interoperability are at the semantic level (e.g., different classification schemes) and at the data structure level (e.g., different XML DTDs). Current approaches that deal with this problem require the intervention of a human expert to perform the mapping between

*Research supported in part by the Advanced Research and Development Activity (ARDA) and the National Imagery and Mapping Agency (NIMA) under Award Number NMA201-01-1-2001, and by the National Science Foundation under CAREER Award IRI-9896052 and Award EIA-0091489.
The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Imagery and Mapping Agency or the U.S. Government.

†Corresponding author. Email address: ifc@cs.uic.edu
‡Worcester Polytechnic Institute, ADVIS Research Group.

XML representations or to map between the conceptual models for those representations. Also, changes in the representations will typically entail specific code changes by a human expert. A recently proposed layered model [11] partitions this problem by considering the requirements of different layers: syntax, object, semantic, and application.

This paper discusses our approach to interoperability in the object layer using XML [2], developed as a part of a framework for geospatial data visualization applications. Applications using our framework should be able to seamlessly pull data from multiple XML data sources. Thus, we focus on the backend portions that allow for querying and integrating data from multiple XML sources.

In developing our framework, we have focused on a number of application scenarios. In this paper, we examine our election results scenario. In this scenario, we create an application that will visualize results from the 2000 U.S. Presidential Election, superimposing vote totals and campaign donations over a given region on a map. Users will be able to search for a geographic area in the United States, choose which data to display (e.g., demographic information, votes cast per candidate, campaign donations, etc.), and then view that data on a map of the region. Data can be displayed at multiple granularities, depending on what the user wants to see (e.g., the user can display a map of Massachusetts with county boundaries showing election results for each county and then zoom in on a map of Worcester County showing election results and campaign donations for each municipality). A screen shot from this scenario is shown in Figure 1.

XML was chosen as our data representation language because it is quickly becoming the new standard for information interchange. The main appeal of XML is that it allows users to define their own document types, and then store and exchange documents conforming to that document type. A document type declaration is used to accomplish this. Document type declarations serve to identify the root element of a document, but can also contain a document type definition (DTD) [1]. This provides users a way of declaring the markup, syntax, and structure of a given document type. This also provides the parser a way of determining whether a document is valid (i.e., conforms to a DTD), rather than just well formed (i.e., uses correct XML syntax).

Allowing everyone to create their own document types makes data exchange easier, but it also makes data interoperability more difficult. Different organizations could model the same information with different document types [1]. It is trivial to load and view documents with different document types, but integrating the data between them is potentially quite difficult.

In this paper, we concentrate on providing a general-purpose application framework that allows data to be interchanged between various XML document types. We utilize XPath to concisely describe a path through the document graph and to select a number of nodes whose values can be aggregated. Finally, we introduce the concept of a *geospatial authority*. Authoritative geospatial information about the region being covered by a given application is collected in an XML document. This document serves as the source for geospatial relationships at the core of the application. For instance, in our election scenario,

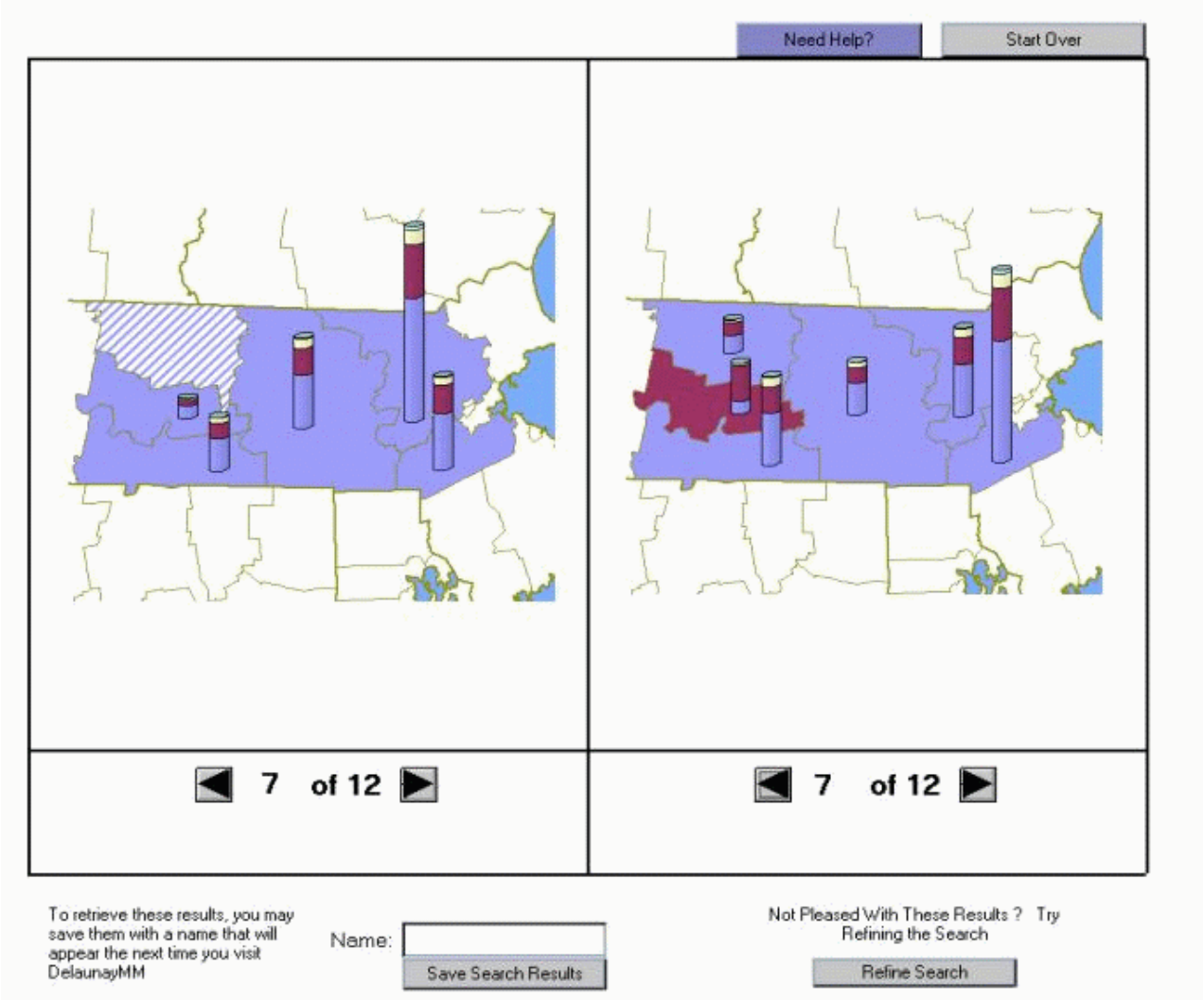

Figure 1: User interface for the election scenario

we maintain a geospatial authority containing the names of all of the states, the counties contained in those states, and the municipalities contained in those counties. This provides us with a means of query refinement by providing context for a given geospatial query (e.g., a search for the string "Worcester" would yield matches in New York, Vermont, Massachusetts, Wisconsin, Missouri, and Pennsylvania—the authority allows us to ask the user for clarification in which "Worcester" is desired). This also provides us with context and a starting point when looking up data in other documents, as seen in the examples given below.

Our framework is implemented in Java 1.3 and uses Apache's Xalan-Java 2 API for XML, XPath, and XSLT processing.

The paper is organized as follows: In Section 2, we examine treating an XML document as a graph, using the Document Object Model (DOM) and XPath as a means of traversing the graph. In Section 3, we present a number of classes that form the core of the backend of our application framework. In Section 4, we introduce our lookup mechanism that allows users to interchange data from one document type to another. Finally, in Section 5, we examine issues that remain open or that need to be addressed to complete our framework.

2 XML As a Graph

XML is actually a collection of W3C recommendations that define the syntax and semantics of XML and its related technologies [1]. The core XML recommendation is the XML Information Set (or Infoset) [5], which models the core abstractions of XML as a set of information items [1]. Information items represent the pieces of an XML document, what is required of them, and how they behave. The items modeled in the Infoset are reflected in the Document Object Model (DOM) [9]. The DOM allows programmers to access XML documents uniformly, regardless of the underlying implementation.

When an XML document is loaded using a DOM compliant parser, a DOM tree is returned. The DOM provides interfaces to the following information items: documents, document fragments, document types, entities, entity references, elements, attributes, processing instructions, comments, text, CDATA sections, and notations. While this is useful, for certain purposes it would be more useful to deal with XML documents as a graph of elements.[1] Each element would have a type corresponding to its attributes. It should be possible to retrieve an element's children, parent, siblings, etc.

Also worth noting is the XPath [4] engine in the Xalan-Java API. XPath provides a simple way of expressing a path through a document tree to select a set of nodes. When a path expression is evaluated, the XPath facilities select a set of nodes relative to a context node. Any information item represented

[1] Documents whose elements contain references to other elements (e.g., IDREFs) can be considered to be a graph, rather than a tree. In the DOM, references are not considered to be edges in the document tree, thus maintaining the tree structure. However, in certain circumstances, it may be necessary to treat references as edges, and thus treat the document as a graph. In any case, an XML document is primarily tree-structured, so we can use terms like sibling, parent, ancestor, and child.

in the DOM (and consequently any information item in any document) can be selected by an XPath expression. Using XPath expressions to traverse an XML document is much more concise and easier to understand than using the DOM. Also, after analyzing a DTD, it is possible to generate XPath expressions for the body of the accessor methods necessary to perform the graph operations described above.

3 Framework Core Classes

Our framework provides a number of core classes, shown below in Figure 2, that allow applications to treat XML documents as graphs, to perform XSL transformations on an XML document, to evaluate XPath expressions against a document, and to perform inter-document lookups.

The base class that we use in the framework is the `XMLObject` class. In our Java implementation, the `XMLObject` class contains a reference to a node in the DOM tree. Through this node reference, it is possible to retrieve any data contained in the node, as well as any nodes linked to that node in the DOM tree. This class also provides an interface to the XPath facilities in the Xalan-Java API. The node referenced by an `XMLObject` object is used as the context node when evaluating XPath expressions. When an XPath expression is evaluated, a set of DOM nodes matching that expression are selected and placed in a DOM `NodeList`. The `NodeList` interface is wrapped by the `XPathResult` class, which allows users to cast the i^{th} node in the list to a `Boolean`, `Double`, `Integer`, `Node`, or `String`. Coupled with prior knowledge of attribute types (determined by a domain expert, or in the future by analyzing an XML Schema for a given document type), this allows typed access to data in the document.

The `XPathResult` class also provides aggregate operations for the set of nodes that match a given XPath expression. Once a set of nodes is selected, it is often necessary to traverse the set, accumulating data. Rather than have the application designer write code to perform these aggregate operations, we provide such code in our framework. The aggregate operations that are provided are listed in Figure 3.

There are three subclasses of the `XMLObject` class: `XMLData`, `XMLSource`, and `LookupResult`. The `XMLData` class represents elements in the DOM tree and is the basis for treating a document as a homogeneous graph. It encapsulates many of the DOM methods that relate only to elements (i.e., retrieving attribute names, retrieving attribute values by attribute name, retrieving children, ancestors, and siblings). It also translates the name and namespace of an element into the type of the graph node. `XMLData` objects are instantiated by the `XMLSource` class. `XMLSource` objects represent XML documents and allow users to load and parse documents, to serialize documents out to a stream, and to retrieve the root node of a document as an `XMLData` object. The `XMLSource` class also provides access to the XSLT [3] facilities (discussed below) provided in the Xalan-Java API. The `LookupResult` class encapsulates the resulting data from an inter-document lookup (discussed below).

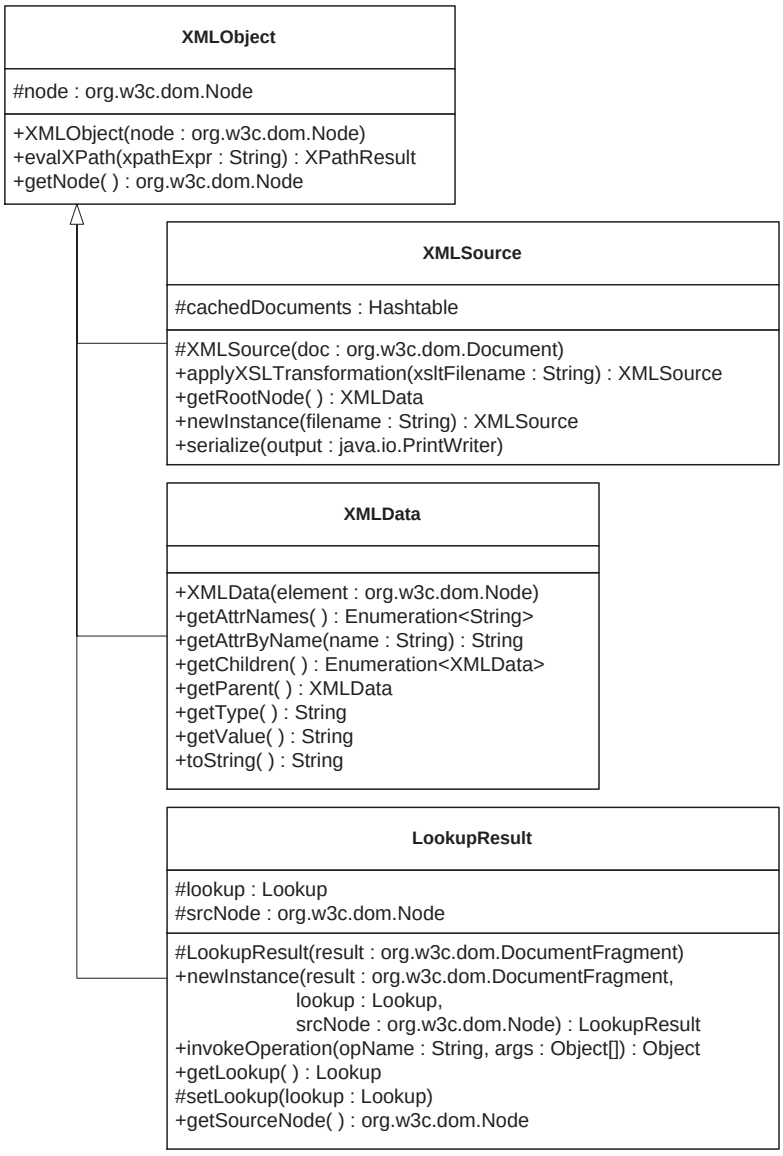

Figure 2: UML class diagram of core framework classes

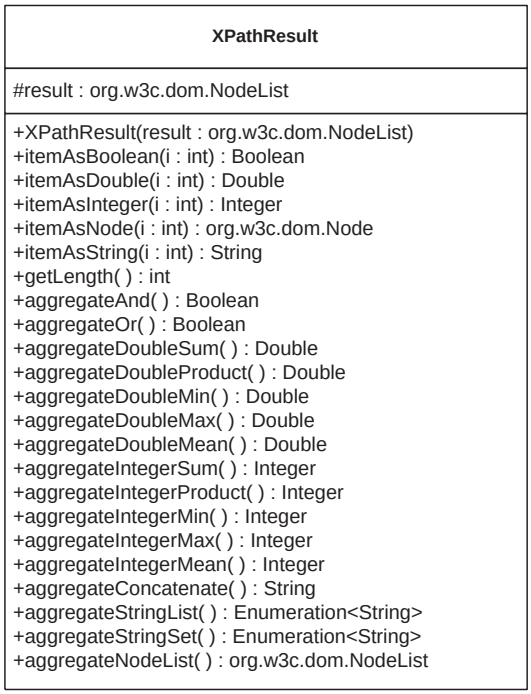

Figure 3: UML class diagram for the `XPathResult` class

3.1 Example using the Framework Core Classes

In our election scenario, we start with an XML document that we call the *geospatial authority*, which contains geographic information about the area being covered by the application (in this case, the United States). For simplicity, we consider an authority document that contains state elements, which in turn contain county elements, which in turn contain municipality elements. The DTD for the geospatial authority is shown below:

```
<!ELEMENT state        (county+)>
<!ELEMENT county       (municipality+)>
<!ELEMENT municipality EMPTY>

<!ATTLIST state        name CDATA #REQUIRED>
<!ATTLIST county       name CDATA #REQUIRED>
<!ATTLIST municipality name CDATA #REQUIRED>
```

This would suggest that subclasses of `XMLData` are needed for states, counties, and municipalities. Each of these classes would have an accessor method that would allow the user to retrieve the name, as well as the region(s) contained by that region (e.g., retrieve all counties for a given state, or retrieve the municipality named "Worcester" from Worcester county in Massachusetts). There are also documents for each state containing election results. Potentially, each state can have a different DTD, and consequently, a different structure for its results document. For simplicity, we will examine the DTDs for election results from only two states, Massachusetts and Maine:

MA_results.dtd

```
<!ELEMENT state        (county+)>
<!ELEMENT county       (municipality+)>
<!ELEMENT municipality (candidate+)>
<!ELEMENT candidate    EMPTY>

<!ATTLIST state        name CDATA #REQUIRED>
<!ATTLIST county       name CDATA #REQUIRED>
<!ATTLIST municipality name CDATA #REQUIRED>
<!ATTLIST candidate    name CDATA #REQUIRED
                       votes CDATA #REQUIRED>
```

ME_results.dtd

```
<!ELEMENT state        (county+)>
<!ELEMENT county       (municipality+)>
<!ELEMENT municipality (ward+)>
<!ELEMENT ward         (precinct+)>
<!ELEMENT precinct     (candidate+)>
<!ELEMENT candidate    EMPTY>
```

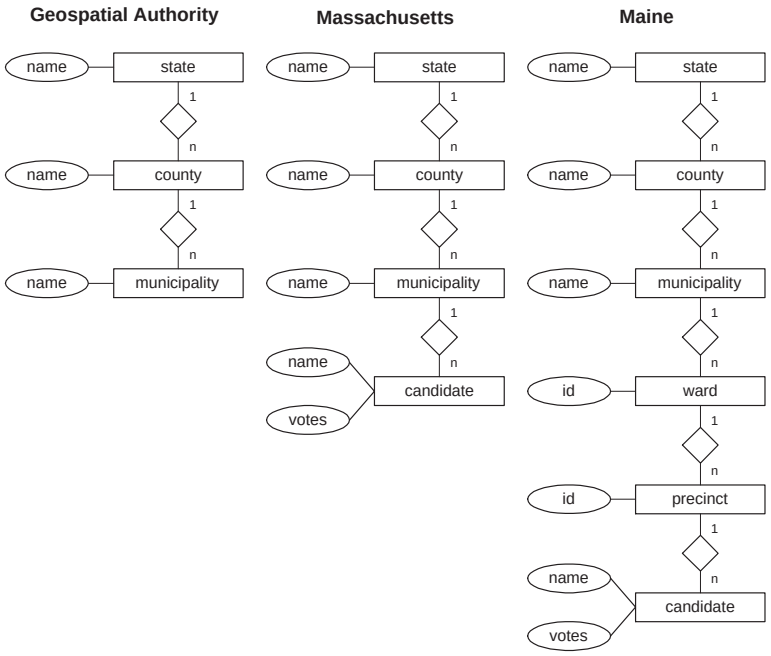

Figure 4: DTD E-R Diagrams

```
<!ATTLIST state        name  CDATA #REQUIRED>
<!ATTLIST county       name  CDATA #REQUIRED>
<!ATTLIST municipality name  CDATA #REQUIRED>
<!ATTLIST ward         id    CDATA #REQUIRED>
<!ATTLIST precinct     id    CDATA #REQUIRED>
<!ATTLIST candidate    name  CDATA #REQUIRED
                       votes CDATA #REQUIRED>
```

This represents the general structure of the election result data as presented at the Massachusetts and Maine state websites. In order to better understand these DTDs, Figure 4 shows the E-R diagrams of these document types.

Note that in the diagram, the cardinality of all relationships is one-to-many. This is due to the fact that, in the DTD, all subelements have the "+" qualifier, meaning that one or more instances of that subelement can appear. While this is fine for the state-to-county relationship and the county-to-municipality relationship, it does not tell the whole story for candidates. The municipality- or precinct-to-candidate relationship should have a cardinality of many-to-many, since there are many municipalities or precincts and many candidates. However, the DTD does not reflect this—it only states that each municipality or precinct can have more than one candidate. To determine a many-to-many relationship,

it would be necessary to examine the actual data.

It was stated above when discussing the geospatial authority document that classes for states, counties, and municipalities would be desirable. However, since each state's results document could potentially have a different structure, it would be unfeasible to define a separate state, county, municipality, and candidate class for each state's election results. It would make more sense to have just one state, county, and municipality class for the entire application and let the framework classes handle the necessary translation between the underlying XML formats of the various election result documents. To accomplish this, we introduce our lookup mechanism.

4 Lookups

Interoperability hinges upon a system that is able to seamlessly interchange data between various document types. Data from one document needs to be selected, possibly restructured, and then linked to data in another document. Our framework provides a way of linking data in this manner using what we call *lookups*. This section discusses the features of our lookup mechanism and some of the design rationale behind those features. We will illustrate this using a running example from our election scenario: defining a lookup for Massachusetts election results for municipalities.

Lookups in our system are specified declaratively in a lookup specification document. This document, stored in XML, allows application designers to specify all lookup types for an application. Each lookup is given a name, a description, and an identifier, as shown below:

```
<?xml version='1.0' standalone='no'?>
<!DOCTYPE lookup-spec SYSTEM 'lookup-spec.dtd'>

<lookup-spec>
<lookup name='MA Municipal Results'
        description='Massachusetts election results per
                     municipality'
        id='1'>
```

4.1 Predicates

Any node in an XML document used by our system can potentially have any number of lookups associated with it. Therefore, we first need a way in which we can define predicates that determine whether a lookup exists from a given source node. These predicates are expressed as XPath expressions. When checking if a node is associated with a lookup, the predicate XPath expressions are evaluated using this source node as the context node. If all of the predicates evaluate to non-null results, we can consider this node associated with this lookup and can

proceed to execute the lookup. In our example, we only want to link municipalities in the geospatial authority to municipalities in the Massachusetts election results document. Consequently, we define the following predicates:

```
<predicate>
<!-- ensure that the source node is a municipality -->
self::municipality
</predicate>

<predicate>
<!-- ensure that the source node is in Massachusetts -->
ancestor::state[attribute::name='Massachusetts']
</predicate>
```

4.2 Arguments

Lookups actually link *entities* (that is, entities in the database sense; these entities are expressed in XML as elements) in different documents, not actual instances of the entities. In other words, lookups act as a link between different data types, not their objects. It may only be possible to state a lookup in terms of variables whose values depend on the context of the lookup being performed (i.e., the source node's context). For instance, in our running example linking a municipality in Massachusetts with a municipality in the election results, one lookup can serve all municipalities in the state. Contextual data, such as the county and municipality name, is necessary for the lookup to be performed, but cannot be determined until the lookup is about to be executed. Therefore, we provide a mechanism for defining arguments whose values are computed before executing the lookup and then substituted into the body of the lookup. These arguments are associated with an XPath expression. When the lookup is about to be executed from a given source node, the XPath expression is evaluated using the source node as the context node, and the resulting value is substituted for the argument name in the body of the lookup. For example, we would need to define arguments for county and municipality names in order to properly link a municipality to the election results:

```
<argument>
  <name>$county_name</name>
  <value>parent::county/attribute::name</value>
</argument>

<argument>
  <name>$municipality_name</name>
  <value>attribute::name</value>
</argument>
```

4.3 Lookups: XPath or XSLT

The next step is to define the actual body of the lookup. There are two types of lookups supported by our framework. The first type of lookup involves simply selecting a set of nodes in a document using an XPath expression. To specify an XPath lookup, the actual XPath expression can be stated directly in the lookup specification document. The second type of lookup involves restructuring a subset of the data in a document using XSLT. To specify an XSLT lookup, the name of the XSLT file can be stated in the lookup specification document. When the time comes to execute a given lookup, the XPath expression or XSLT file is loaded, argument values are evaluated and substituted, and the lookup is performed. It is also necessary to state the file name of the target document. In our example, we want to find the municipality element in the target document and to select all children of that municipality. The XPath lookup to accomplish this, written in terms of the variables given above, is shown here:

```
<target-document>
MA_election_results.xml
</target-document>

<!--
  Start at the root, trace through the tree to find the
  municipality, and select all children of the municipality.
 -->
<xpath-expr>
/child::state[attribute::name='Massachusetts'] \
  /child::county[attribute::name='$county_name']  \
  /child::municipality[attribute::name='$municipality_name'] \
  /child::*
</xpath-expr>
```

4.4 Linking the Results

When an XPath expression is evaluated, a `NodeList` or a `NodeIterator` containing the selected nodes is returned. When an XSLT file is evaluated, the resulting DOM tree structure is returned. While DOM trees that result from XSL transformations can have XPath expressions evaluated against them, `NodeList`s and `NodeIterator`s that result from XPath expressions cannot. It would be useful to organize these results in such a manner that XPath expressions can be evaluated against them. Similarly, we would like to be able to cache and possibly serialize these results. It would be nice if we could simply add these nodes as children of the lookup's source node, but this is not allowed by the DOM because it could potentially make the containing document invalid. However, the DOM provides the `DocumentFragment` interface for representing lightweight collections of nodes. The `DocumentFragment` interface is intended to support movement of nodes for operations such as "cut" and "paste" [9]. Also, since `DocumentFragment`s are nodes, XPath expressions can be evaluated against them.

We can specify that the XSLT part of the Xalan-Java API produce a `DocumentFragment` with the results of the transformation. For XPath lookups, we can simply take each node in the resulting `NodeList` or `NodeIterator` and add that node as a child of a `DocumentFragment`.

Our framework provides the `LookupResult` class, a subclass of `XMLObject`, to store the resulting `DocumentFragment`. The details of this class are discussed later on, but it is worth noting here that in the lookup specification, the application designer can specify which result class to instantiate with the lookup results. Either `LookupResult`, or a subclass of it, can be specified to take the lookup results. In our running example, we simply use the `LookupResult` class:

```
<result-class>LookupResult</result-class>
```

4.5 Operations: Accessing the Results

Once a lookup is performed, code is needed to access the data contained in the nodes that result from the lookup. Potentially, each lookup can select nodes from different documents containing different structures. Therefore, each lookup would require its own class to be instantiated with the appropriate code to access its data. However, this can quickly become cumbersome as the number of lookups increase. Also, if the structure of the various documents change, those classes would need to be rewritten. In our election scenario, each state could potentially require a separate class for lookups for its election results. While these classes could be arranged into an inheritance hierarchy, having 50 election result classes would be a little excessive. It would be more useful if accessor methods could be defined at the lookup level.

Accessor methods can be defined using XPath expressions that will be evaluated using the `DocumentFragment` as the context node. The return values for such accessor methods can be computed using our aggregate operations discussed above. The application designer can therefore define accessor operations in the lookup specification using a name, aggregation type, and an XPath expression for the body of the operation. Parameters can also be defined when extra context is needed for the operation. Default values for those parameters can also be given, where necessary.

In our running example, we would define the following two operations:

```
<!-- select all candidate names, return a set of strings -->
<operation name='getCandidateNames' aggregation='sset'>
  <body>
    child::candidate/attribute::name
  </body>
</operation>

<!-- select vote total for a candidate, return an integer -->
<operation name='getVotesByCandidate' aggregation='isum'>
  <parameter>
```

```
    <name>$candidate_name</name>
  </parameter>

  <body>
    child::candidate[attribute::name='$candidate_name'] \
    /attribute::votes
  </body>
</operation>
</lookup>
</lookup-spec>
```

If all election result lookups define operations with the same name, multiple classes to represent the election results would no longer be needed. Rather, we can use one subclass of `LookupResult`. Operations defined in the lookup specification can be invoked using the `invokeOperation()` method provided by the `LookupResult` class. In our subclass, we can provide methods (e.g., `getCandidateNames()` and `getVotesByCandidate()`), which simply call `invokeOperation()` with the appropriate parameters.

4.6 The Lookup Specification

As stated above, all lookups are specified in an XML document. These documents must conform to the lookup specification DTD (`lookup-spec.dtd`), shown below:

```
<!ELEMENT lookup-spec     (lookup+)>
<!ELEMENT lookup          (predicate+,
                           argument*,
                           target-document,
                           (xpath-expr | xslt-file),
                           result-class,
                           operation*)>
<!ATTLIST lookup name        CDATA #REQUIRED
                 description CDATA #REQUIRED
                 id          ID    #REQUIRED>
<!ELEMENT predicate       (CDATA)>
<!ELEMENT argument        (name,
                           value)>
<!ELEMENT name            (CDATA)>
<!ELEMENT value           (CDATA)>

<!ELEMENT target-document (CDATA)>
<!ELEMENT xpath-expr      (CDATA)>
<!ELEMENT xslt-file       (CDATA)>
<!ELEMENT result-class    (CDATA)>
```

```
<!ELEMENT operation       (parameter*,
                           body)>
<!ATTLIST operation name        CDATA #REQUIRED
                    aggregation (nodelist |
                         and | or |
                         dsum | dprod | dmin | dmax | dmean |
                         isum | iprod | imin | imax | imean |
                         concat | slist | sset) #REQUIRED>
<!ELEMENT parameter       (name, default-value?)>
<!ELEMENT default-value   (CDATA)>
<!ELEMENT body            (CDATA)>
```

5 Conclusion and Open Issues

In this paper, we analyzed a geospatial application for the U.S. elections as a means of illustrating the problems that need to be solved in the mapping between different XML representations and their conceptual models. The framework that we presented allows application designers to treat XML documents as homogeneous graphs and to evaluate XPath expressions and XSL transformations against a document. Our framework also allows application designers to define a set of lookups that integrate data from multiple documents. Data retrieved using a lookup can also have operations defined for it declaratively in a lookup specification. Coupled with our aggregation operations, this provides uniform access to non-uniformly structured data.

We anticipate the need for introducing metadata in the semantic layer to guide the translation process between document types. In this section, we discuss metadata issues in the semantic layer and a number of other issues that still need to be investigated.

5.1 Metadata, Semantics, and Ontologies

A certain amount of metadata is necessary for a domain expert or application designer to appropriately identify the entities that can be linked and how the lookups should be performed. In some cases, this metadata is available from the DTD or from another external source that describes that document type. In other cases, this metadata may be implied. For example, one can reasonably expect measurements in documents from Europe to be in metric units, while measurements in documents from the United States to be in feet and inches, even though this information is not explicitly stated anywhere in the document. Some conversions may be handled by XSLT, but some may require more complicated computations and consequently must be performed by another mechanism.

There are also naming issues that need to be taken into consideration. For instance, it is possible for a geographic entity to have multiple accepted spellings (e.g., Foxboro, MA vs. Foxborough, MA; or Mt. Washington vs. Mount Washington). Often times, foreign place names have multiple accepted English

spellings, and this too must be taken into consideration. In some cases, this can be handled using a crosswalk table or some other form of dictionary data structure. In other cases, the accepted spellings of a place name may depend on the current context or the location of that place.

Further research is also needed to see how to use ontologies, especially as presented in [7] and [8], in our system.

5.2 Technical Issues

We need an engine capable of executing graph queries. With such an engine, it may be possible to take a generic graph query language that supports aggregation, such as the G+ language presented in [6], and compile queries in that language into XPath expressions or another XML based query language (e.g., XQL, or whichever XML query language reaches W3C recommendation status).

We can also look at performance issues, especially in a distributed environment. Each XML document that we are interoperating between can potentially be stored on a different server. We need to look at different ways of caching and processing data. We also are examining different security schemes for accessing the various XML data sources.

At this point in the development of our framework, much of the code necessary to perform the tasks described above must be written by hand. This may become too complex to be practical. In the future, we plan to analyze all DTDs that will be supported by and application and generate the necessary code to perform many of the tasks described above. It would still be necessary for the application designer or a domain expert to determine the links between the different document types. It may be possible, however, to automatically or semi-automatically generate the lookup specifications necessary to execute the lookups, but more research is needed in this area. In order to generate any code based on a DTD, it is first necessary to parse the DTD to determine its structure. This can be done by creating a DTD graph, using algorithms described in [10] and [12]. From there, we can begin to examine code generation for XPath-based accessor methods in the lookup specification. It may also be possible to automatically or semi-automatically generate the lookup code (using either XPath or XSLT). Code generation can be supplemented by the input of the application designer or domain expert via a visual tool that displays E-R diagrams based on a given DTD.

5.3 Document Structure

The election scenario example shown throughout this paper is somewhat contrived (the actual data was originally retrieved in HTML format from a website and then converted to XML) as we have created all of the documents and defined their structure. Obviously, in a real-world application, the document structures could potentially be quite different, requiring more restructuring of the data.

We also need to examine the effects that changing a document type would have on code that has already been written or generated. Ideally, our final

framework would be sufficiently powerful to handle such changes.

6 Acknowledgements

We would like to thank Nancy Wiegand and Steve Ventura for discussions on the subject of statewide geospatial applications.

References

[1] D. Box, A. Skonnard, and J. Lam. *Essential XML Beyond Markup*. Addison-Wesley, 2000.

[2] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 (Second Edition). W3C Recommendation, 2000.

[3] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, 1999.

[4] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation, 1999.

[5] J. Cowan and R. Tobin. XML Information Set. W3C Candidate Recommendation, 2001.

[6] I. Cruz and T. Norvell. Aggregative Closure: An Extension of Transitive Closure. In *IEEE International Conference on Data Engineering*, pages 384–390, 1989.

[7] M. Erdmann and R. Studer. How to Structure and Access XML Documents with Ontologies, 2001. DKE 36(3): 317–335.

[8] F. Fonseca. Gis_ontology.com. GIScience 2000. http://www.spatial.maine.edu/ fred/#Publications.

[9] A. Le Hors, P. Le Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 2 Core Specification Version 1.0. W3C Recommendation, 2001.

[10] D. Lee and W. Chu. Constraints-Preserving Transformation from XML Document Type Definition to Relational Schema. In *Proc. 19th Int'l Conf. on Conceptual Modeling*. Springer-Verlag, October 2000.

[11] S. Melnik and S. Decker. A Layered Approach to Information Modeling and Interoperability on the Web. In *ECDL 2000 Workshop on the Semantic Web*, Lisbon, Portugal, September 2000.

[12] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *Proceedings of the* 25^{th} *VLDB Conference*, Edinburgh, Scotland, 1999.