A FRAMEWORK FOR CREATING CUSTOMIZED MULTI-MODAL INTERFACES FOR XML DOCUMENTS

Sami Rollins

Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA, USA, 93106
srollins@cs.ucsb.edu

Neel Sundaresan

IBM Almaden Research Center
650 Harry Rd.
San Jose, CA, USA, 95120
neel@almaden.ibm.com

ABSTRACT

The eXtensible Markup Language(XML) is emerging as a new way to store and communicate data. Even though its primary application is as the future of the World Wide Web, it can be used in a variety of situations to structure electronic data. As XML becomes ubiquitous, there is a need to develop tools to allow users to view, navigate, and modify the underlying XML data via a high-level, multi-modal interface. Moreover, because XML can be used in a variety of situations, the tools must allow a user to access the data via non-traditional interfaces. The web, eCommerce, and digital classrooms are all possible applications for XML. This paper presents a framework for developing multi-modal tools to view, navigate, and modify XML structures.

1. MOTIVATION

XML[3] is a powerful language that enables a user to store and communicate semi-structured data. In addition to being the future of the World Wide Web[2], XML has the potential for use in distance learning[4] as well as business-to-business, eCommerce applications[1, 7]. As XML becomes ubiquitous, there is a call to allow users to receive XML data in different output modes as well as issue directives to navigate the data using different input modes. Circumstance as well as preference may restrict the input or output mode choices available. For example, a user might simply *want* to have a pictorial view of the data, or the user might be a child who cannot yet *read* a textual view of the data.

Currently, the only way to view an XML document is textually. XML programmers use basic text editors to view and modify XML documents. Some browsers process XML[5], however they still only present the same textual view that you would see in a text editor. There are specific XML editors[8, 9, 10], but first, those tools focus on XML modification, and second they still only present a text-based view of the XML document.

Languages like SMIL[6] are being developed with the goal of allowing a user to specify a multi-modal presentation using XML. Our system takes the opposite approach. Our aim was to develop a system that would automatically create a multi-modal presentation of *any* XML document. Moreover, we wanted to design our system such that the input and output modes of the engine could be customized by the user, therefore avoiding the overhead of having a static system with multiple different concurrent input and output modes.

Section two of this paper presents the design of our system that we call the MakerFactory. Section three is a detailed discussion of the user customization facilities we provide to the user while section four focuses on the applications for which our system is well suited. The paper concludes in section five.

2. DESIGN OF THE MAKERFACTORY

The MakerFactory is designed as an infrastructure for automatically creating customized interfaces for XML documents based upon a pre-defined XML schema. The schema provides us with semantic information about the document itself. It contains cues such as the size and depth of a document conforming to the given schema. By extracting those cues from the schema definition, we can build a smarter rendering system.

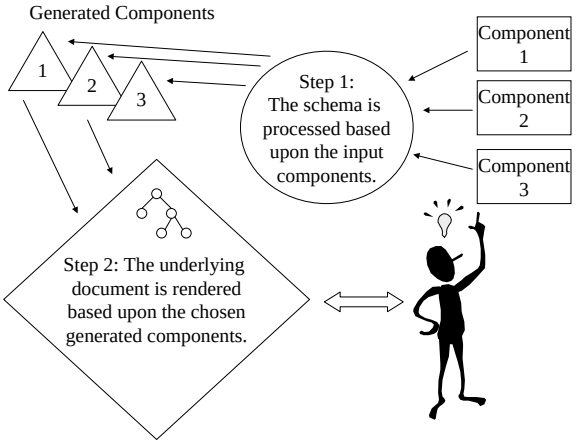

Figure 1: The MakerFactory operates in two stages. First, a set of code-generation components is chosen and fed into the engine. The result is a set of generated classes that can be invoked in step two to present a rendering to the user.

The design itself is two-fold as shown in figure 1. In

the first phase, the user chooses a series of customization components from the library provided. Each component is designed to provide the user with a different input/output mode. For example, one component may allow the user to drag and drop a pictorial view of the XML while another component may allow the user to view and edit a document via a speech-based interface. Each component analyzes the schema itself and generates a unique set of classes that will provide the given input and output mode for a document with the specified schema.

The second phase is the run-time phase. At run-time, the user chooses which generated classes should be used for the given situation. The rendering system invokes all of the specified classes and provides synchronized access to the XML tree. Each component will have the same view of the underlying structure, however the interpretation of that view that the component presents to the user varies. It is the job of each component to present the information to the user using a different *mode* of output and to listen for user input using a different *mode* of input from all other such components.

2.1. Phase 1: Code-Generation

Because each component chosen in the code-generation phase is responsible for *making* a set of classes to provide a specific interpretation of the tree, we call these components Makers. Each Maker is responsible for a single input/output mode. By designing the system is such a modular way, we avoid incurring the overhead of undesired modes of input or output. For example, the user who does not have access to a monitor does not invoke any visual modes of output. Therefore, the system does not incur the overhead of rendering video or complex images etc.

Given a node from the Schema, the Maker first determines how the node should be displayed to the user. For example, a TextMaker, that is to say a Maker that presents a textual view of the document to the user, might conclude that to render a node it simply prints the tag name of the node to standard output while a PictureMaker, a Maker that presents a pictorial view of the document, might conclude that the rendering of a node requires searching an image database for an image to match the tag name of the node. If it finds such an image it displays it on the screen.

The Maker then decides the type of input that the user must supply to direct navigation. A SpeechMaker might decide that a user may ask for any of the node's children by speaking the tag name while a TextMaker might decide that the user can edit a text field containing the tag name and the result will be the new tag name of the node.

The result of the Maker's analysis is a set of Java classes. The set of classes should *mediate* the communication between the user and the underlying XML structure, therefore the classes should minimally implement our Mediator interface. At run-time, a set of Mediators is chosen by the user and invoked.

An example Maker is shown in figure 2. In this example, the TextualEditorMaker creates a text-based interface for the user. The user can modify the underlying tree by filling in the appropriate text fields.

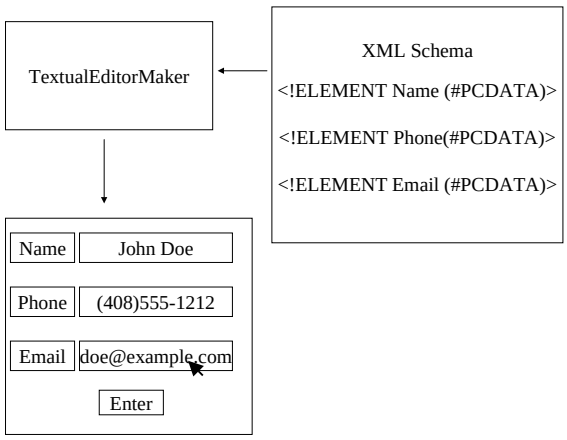

Figure 2: The schema is analyzed by the Maker and a set of Java classes is produced that mediates communication between the user and the underlying tree structure. For example, each node in the tree might have an associated text field in a textual editor. The user can edit each text field to change the tag name of the elements in the underlying tree.

2.2. Phase 2: Run-time Rendering

At run-time, the user selects a series of Mediators to be invoked. Each Mediator is selected based upon both user preference as well as any restrictions imposed by the run-time environment. For example, if the user is accessing the XML document with a cellular phone, it is likely that the user will only choose auditory Mediators, however, if the user is using a traditional keyboard and monitor, the user may want to hear and see the document choosing both an auditory Mediator and a pictorial Mediator as seen in figure 3.

When the user issues a command understood by one Mediator, that Mediator can change the tree view for the remainder of the Mediators. There are two ways that the tree view can change. First, the system implements the concept of a cursor. Therefore, the tree view can change based upon the location of the cursor. The current view is simply the subtree rooted at the node currently pointed at by the cursor. Second, the Mediator can actually change the contents of the tree. Therefore, if a Mediator changes the tag name of the current node, the rest of the Mediators should update their view to reflect the change.

3. CUSTOMIZING THE OUTPUT

One benefit of our system is that it provides three different levels if customization. By nature, the schema analysis of the code-generation phase provides one level of customization. We automatically exploit the underlying semantic information contained in a schema specification. Also, the user chooses the components that are invoked at run-time. This keeps the user from incurring the overhead of running components that cannot be used. Finally, the user

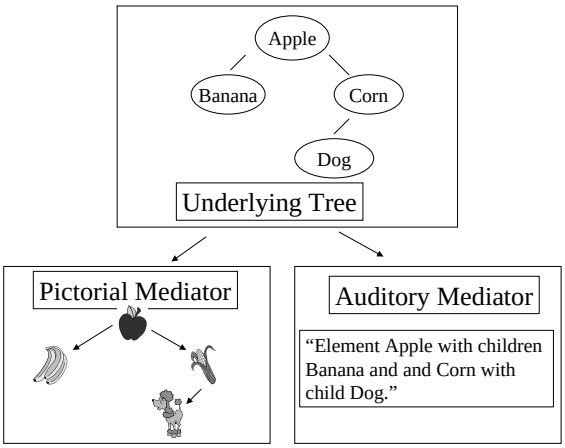

Figure 3: The same underlying XML tree structure may have multiple different representations. An auditory Mediator speaks the tag names of the nodes while a pictorial Mediator searches a database to find images that represent the information contained in the nodes.

may specify a series of customization rules during the code-generation phase. These rules are analyzed along with the schema to produce the run-time Mediator. These rules are specified in XML and are designed to be simple for the user to construct. Therefore, the user can control what the system will do with a given node without having to write one line of code.

3.1. Customization Rules

By generating our rendering system based upon the schema itself, we can exploit the inherent information contained in the schema. However, the information contained there may not be sufficient for a user who wishes to customize the system even further. Therefore, we provide a high-level rule specification language that is further refined by each different type of Maker. Since the rule language is specified in XML, we can describe its restrictions using an XML DTD.

```
<!ELEMENT RenderRules(ElementRule|AttributeRule)+>
<!ELEMENT ElementRule (ElementRendering)+>
<!ATTLIST ElementRule
          elementtid CDATA #REQUIRED>
<!ELEMENT ElementRendering (Render|Action)+>
<!ATTLIST ElementRendering
          cond CDATA #IMPLIED>
<!ELEMENT Render EMPTY>
<!ATTLIST Render
          depth(NONE|NODE_ONLY|CHILDREN_ONLY|
                ATTRS_ONLY|NODE_CHILDREN|
                NODE_ATTRS|CHILDREN_NODE|
                CHILDREN_ATTRS|ATTRS_NODE|
                ATTRS_CHILDREN|NODE_ATTRS_CHILDREN|
                NODE_CHILDREN_ATTRS|
                ATTRS_CHILDREN_NODE|
                ATTRS_NODE_CHILDREN|
                CHILDREN_NODE_ATTRS|
                CHILDREN_ATTRS_NODE|NAME|VALUE|
                NAME_VALUE|VALUE_NAME) 'NODE_ONLY'
          cond CDATA #IMPLIED>
<!ELEMENT Action EMPTY>
<!ATTLIST Action
          classname CDATA #IMPLIED
          ismodifier (TRUE|FALSE) 'FALSE'>
<!ELEMENT AttributeRule (AttributeRendering)+>
<!ATTLIST AttributeRule
          attribteid CDATA #REQUIRED>
<!ELEMENT AttributeRendering (Render|Action)*>
<!ATTLIST AttributeRendering
          cond CDATA #IMPLIED>
```

The user should provide an XML document conforming to this schema during the code-generation phase. It is easiest to explain the functions of each element and attribute using an example.

```
<RenderRules>
  <!-- Rule for all Name elements. -->
  <ElementRule elementid=''Name''>
    <ElementRendering cond=''noMiddleName''>
    <!-- If there is no middle name, render
         last name, then first name. -->
      <Render depth=''CHILDREN_ONLY''
              cond=''isLast''/>
      <Render depth=''CHILDREN_ONLY''
              cond=''isFirst''/>
    </ElementRendering>
    <ElementRendering cond=''hasMiddleName''>
    <!-- If there is a middle name, render
         first, middle, then last name. -->
      <Render depth=''CHILDREN_ONLY''
              cond=''isFirst''/>
      <Render depth=''CHILDREN_ONLY''
              cond=''isMiddle''/>
      <Render depth=''CHILDREN_ONLY''
              cond=''isLast''/>
    </ElementRendering>
  </ElementRule>
  <!-- For all occupation attributes... -->
  <AttributeRule attributeid=''occupation''>
    <!-- If the occupation is ``secretary'',
         then invoke the Java class to change
         the value to Administrative
         Assistant. -->
    <AttributeRendering cond=''isSecretary''
                        ismodifier=''TRUE''>
      <Action classname=''changeToAdminAssit''/>
    </AttributeRendering>
  </AttributeRule>
</RenderRules>
```

The ElementRule element specifies the rendering of elements with the tag name "Name". First, the rendering engine invokes the condition classes. The value of the cond attribute should be the name of a Java class that implements

a UnaryPredicate interface. The UnaryPredicate should take one argument, the node in question, and produce a boolean value. In the first case, the condition should evaluate whether or not the name contains a middle name portion. If so, the rendering engine will first render all children of the name element that satisfy the condition that they are *last* names, and then will render all children that satisfy the *first* name condition. However, if the "Name" node does contain a *middle* name, the children will be rendered in the order *first*, *middle*, *last*.

The AttributeRule has a similar structure. In this case, the AttributeRule applies to all nodes that have the name "occupation". If the value of the attribute meets the condition specified in "isSecretary", the the Java class "changeToAdminAssit" will be instantiated and invoked. Since the class is identified as a modifier of the tree, the underlying structure should be changed when the invocation of the class completes.

The rules are designed to give the user as much power to specify the rendering as possible, but still use a high-level specification language. Because these rules are designed to be general and apply to any XML schema, a specific Maker class is encourage to extend the specification.

4. APPLICATIONS - THE WEB AND BEYOND

The framework described has been implemented entirely in Java using the IBM XML4J parser. We have focused mainly on the SpeechMaker/Mediator, the component that provides the facility for speech-based input and output. We call this component Audio XmL(AXL). AXL has been implemented using the IBM Speech for Java implementation and the ViaVoice synthesizing and recognizing engine. There are many possible applications for such a framework. We describe three possible application scenarios.

The most obvious application of the MakerFactory is the World Wide Web[2]. As the web transitions to XML, there is a call for browsing facilities that will allow users to view the content that exists on the web. By providing such a customizable system, we allow a user not only to access the web, but to access the web through non-traditional means. A user does not have to have access to a traditional computer terminal in order to surf the web. More specifically, the MakerFactory can be used to allow voice access to the web for print disabled people or people who simply wish to browse the web using a cellular telephone.

Another application is a distance learning scenario. The digital classroom is becoming a reality. Even now, students bring laptop computers to lecture and attend a class from a city 100 miles away from the lecture itself. By taking advantage of the technology that exists for students, lecturers can provide a multimedia presentation to their students by simply specifying their lectures in XML[4]. If every student had a laptop with the XML version of the lecture, the MakerFactory could render each node in the tree at the appropriate point in the lecture. The input mode to determine tree traversal could be the instructor's voice while the output modes could include slides, pointers to relevant web pages, even videos. This would allow the students many different views of the material presented and hence, a greater understanding of the material.

Finally, as commerce becomes eCommerce, XML is stepping in as the standard way to communicate *Business-2-Business*[1, 7]. However, as long as human intervention is required at any step of the way, systems to view and modify the XML documents being sent from one business to another are imperative. The MakerFactory not only provides the ability to access that data, it provides the customization facilities needed to be able to filter the XML data. A secretary creating an order might require a very different view of the XML document than a computer professional stepping in to cancel an erroneous order. Therefore, each person that must see the XML document invokes a different set of Mediators when running the MakerFactory. Two people can see the same document, using the same rendering system, but still only see the information the relevant information at any given time.

5. CONCLUSION

XML is becoming ubiquitous. It has implications in many different scenarios. At this point, the only systems available for viewing, modifying and navigating XML documents are text-based and present a limited view of the data itself. We have developed a framework that extends XML interaction and representation to include an extensible set of input and output modes. Our system is completely customizable allowing the user to choose the view of the document that should be presented as well as the method of input the user should employ to change that view. This not only has implications for web-related applications, it applies to multimedia style and eCommerce style applications as well. Such a system will become increasingly important as XML is widely adopted as a standard means of data representation and communication.

6. REFERENCES

[1] corp.ariba.com *http://www.ariba.com/corp/home/*

[2] J. Bosak. *XML, Java, and the future of the Web* http://metalab.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm

[3] Extensible Markup Language (XML) 1.0 W3C Recommendation 10 February, 1998 *http://www.w3.org/TR/1998/REC-xml-19980210*

[4] T. Ferrandez *Development and Testing of a Standardized Format for Distributed Learning Assessment and Evaluation Using XML* MS Thesis, University of Central Florida, Orlando, Florida, Fall 1998.

[5] J. Luh *With Several Specs Complete, XML Enters Widespread Development* http://www.internetworld.com/print/1999/01/04/-webdev/19990104-several.html

[6] Synchronized Multimedia Integration Language (SMIL) 1.0 Specification *http://www.w3.org/TR/REC-smil/*

[7] webMethods *http://www.webmethods.com/*

[8] J. Murray *Conquering XML with Xeena* http://www-4.ibm.com/software/developer/library/xeenatech.html

[9] XMLPro *http://www.vervet.com/*

[10] XMLSpy *http://www.xmlspy.com/*