AVoN Calling: AXL for Voice-enabled web Navigation

Sami Rollins
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106-5110
srollins@cs.ucsb.edu

Neel Sundaresan
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
neel@almaden.ibm.com

Abstract

The World Wide Web is a rich source of information that has become a universal means of communication. XML promises to be the future of the WWW. However, as HTML is replaced by its more powerful counterpart, traditional browsers are not sufficient to display the information communicated in an XML document. Today's browsers are capable of showing only a textual version of an XML document. This is limiting not only for a viewer in a traditional scenario, but is a barrier for a user who wishes to access the information without having access to a traditional keyboard, mouse, and monitor.

This paper presents a framework for developing non-traditional, schema driven, customizable interfaces used to navigate and modify XML documents that may be served over the web. Our system, Audio XmL(AXL) focuses on developing a speech-based component within that framework. At the most basic level, we provide a Speech DOM, a spoken equivalent to the Document Object Model. Beyond that, we provide an intuitive set of commands based upon schema as well as a customization language. AXL enables voice-based web browsing, but without requiring extra effort on the part of the web page designer. Given any XML document, AXL allows the user to navigate, modify and traverse the structure and links of the document entirely by voice. To illustrate, we focus on how AXL can enable a user to browse the web via a cellular phone.

Keywords

voice browsing, audio, multi-modal, accessibility, portable devices

1 Introduction

The eXtensible Markup Language(XML)[6] is emerging as a new way to both store and communicate data. It affords the user the benefit of separating the content of information from its presentation which makes it well suited for a variety of applications. The primary application of XML is as the new language of the World Wide Web[3]. As HTML is replaced by its more powerful counterpart XML, traditional web browsers do not provide the necessary means to display and navigate XML data[23]. Currently, the only browser support for XML provides a textual view of the XML document. This is insufficient for the user who wishes to have a graphical view of the document in her browser. Further, a user may actually wish to interact with the document using a completely different input and output mode.

As pervasive computing technology brings the web to anyone anywhere, XML accommodates the vast and diverse types of communication that must occur. With the explosive growth of portable devices such as Palm Pilots and cellular phones, there is a growing demand for technology that will allow users to be connected to the Internet from anywhere through devices that are not suitable for use with a traditional keyboard, mouse, and monitor[17]. Current technology[30, 29, 25] allows limited access to static data extracted from websites such as **Yahoo!**, however a current goal is to find an all-purpose solution to allow access to *any* website. Along with that goal comes the challenge of finding a way for the user to interact with the browsing system without having to have a mouse in hand.

This paper presents a fully customizable system that provides a user-specified, multi-modal view of any XML document. The system that we call the MakerFactory allows the user to select a set of rendering components as well as define an optional rule document that further specifies the rendering. Additionally, the MakerFactory defines a link traversal mechanism. Therefore, a user can navigate not only within the

document, but may navigate the entire web using our Renderer. This system eliminates many of the constraints imposed by current browsing systems. First, the user is not constrained to a visual representation of the data. In fact, the system eliminates the requirement of a monitor. Moreover, the system also eliminates the requirement of keyboard/mouse input. A further benefit of the system is that we take advantage of the XML schema model. Given a published schema, the MakerFactory generates a renderer that is unique to that schema.

The focus of our work has been to develop an auditory component for our system. We call this component Audio XmL(AXL). AXL allows a user to access any XML document without requiring special directives or alternative designs from the web page designer point of view. When instantiated, AXL automatically generates an aural interface to an XML document conforming to a given schema. The generated class can be used in conjunction with the MakerFactory Renderer to navigate and modify an XML document using speech.

The use of speech for input and output is inherent for the user of a cellular telephone. AXL provides a system that uses speech as both the input and output medium. As long as a document is in XML form, AXL provides a mechanism for a user to navigate and modify that document. As the web transitions to XML, AXL provides the means to navigate the web by voice. Even if a page has an unknown schema, AXL will provide a basic navigational mechanism. With a published schema, a user can specify a customized view of any page conforming to that schema. Moreover, unlike many of the standards that are currently being developed, AXL does not require any extra stylesheet type specifications. All that is required is the XML document itself.

As an example, let us suppose that a business executive wishes to browse his newspaper on his way to work. AXL can be customized to allow the user to first browse all of the headlines, then choose a story to be read. Additionally, the user can stop the current story, move to the next story, return to the previous story, ask for the author, and all without taking his eyes off of the road. Furthermore, if the user reads the sections of the paper in the same order everyday, AXL can be customized to automatically browse in that order saving the user the trouble of navigating the document.

To test the usability of our system, we designed a test schema and observed user's reactions to using voice input and hearing speech output when navigating a document. The results indicate that our design supports many of the goals we set out to achieve. However, we also found that AXL needs to be extended to support features like a logical help command and more intuitive error feedback mechanisms.

In section 2 we present a description of work done in related areas. Section 3 examines the design decisions and architecture of the MakerFactory system and of AXL in particular. Section 4 details the implementation of the system. Section 5 elaborates on one specific web-based application and section 6 looks at the observations we have made about our system based upon user experience. Section 7 concludes with a look at the contributions of this work and future directions of this research.

2 Related Work

We first evaluate work that has been done involving audio in the human-computer interface in general. We then examine efforts to bring audio specifically to the web. Next we survey current trends toward uniting speech and XML and conclude with a look at current work in voice browsing by phone.

2.1 Audio in the Human-Computer Interface

Aster[22] proposed a new paradigm for users of aural interfaces. It suggests that the reader of a document can scan a page worth of information and, virtually simultaneously evaluate the content of the entire page. Based upon cues like font size and page layout, a reader can determine which part of the page is useful and which part of the page is not. The user can then choose to read one portion or another of a given page. A listener does not have that luxury. Audio by nature is serial. However, by using the semi-structured nature of LaTex[13], Aster creates system to allow a user to "listen to" a LaTex document in a more interactive manner.

We propose that XML provides an even greater benefit. Aster provides a static navigational mechanism. However, XML is more dynamic. Each document changes based upon the given schema. By using the knowledge of a given schema, we provide not one, but many customized aural interfaces.

Other types of systems have been constructed that look at how to use sound, speech or non-speech, to represent the structure of a document. [4] develops a set of guidelines for integrating non-speech audio into user interfaces. It presents a comprehensive look at the components of sound and how each can be used to convey information. [2] is a similar system that looks at how to design sounds to represent the types of activities performed in your application. [20] looks specifically at how sound can convey structure and location within a document. We leverage off of the results presented in this work to determine the most effective method of integrating sounds into the AXL system.

Data sonification is another related area. [33] examines what it means to represent data with sound. Similar work is detailed in [14]. The concepts defined in these bodies of work help us to define what it means to represent something visual using sound. These concepts are integral in determining the best way to generate speech-based interfaces. Our system is without benefit unless a user can actually make use of the system. The work detailed in this section helps to define the standards by which such a system can be evaluated.

2.2 Audio Web

There is a large body of work that has addressed the question of how to make the current World Wide Web accessible through a speech-based interface. Most of the work done has focused around how to make the web accessible to users with print disabilities. By looking at previous research in this area, we generalize the concepts developed for the HTML case in order to create a system that makes XML-based documents accessible to any person who is not accessing electronic data through a traditional computer interface using a keyboard, mouse, and monitor.

Both [8] and [9] look at how to make the web more accessible by integrating sound components like voice and pitch. This work focuses around designing useful experiments to determine how a user would best react to different aural representations. It details a thorough analysis of the parameters involved with designing an aural interface. Where this work mainly focuses on designing a set of tactics used to represent HTML structures, we want to design a more general system that takes into account these findings and examines how useful an interface we can generate automatically, in the general case.

Other work on making the web accessible has taken a similar approach, looking at HTML in specific and determining, for example, how one should best represent an <H1> tag. [12, 36, 37, 19, 1] all look specifically at HTML. Our goal is to generalize the approach presented in this body of work.

2.3 XML and Speech

A slew of different XML instances are currently being developed with the goal of making the web voice-enabled [24, 29, 11, 25, 30]. The attention seems to be centered around developing a language in which the web site developer or voice web service provider can specify how the user can access web content. In other words, it is up to the developer to create an XML specification of the speech-based interaction that a user can have with the server.

VoiceXML[29] is emerging as the standard and essentially subsumes SpeechML in functionality. VoiceXML provides the programmer with the ability to specify an XML document that defines the types of commands a voice-enabled system can receive and the text that a voice-enabled system can synthesize. Similar to the Java Speech Markup Language, it allows the XML programmer to specify the attributes used to render a given excerpt of text. This may include information about parameters such as rate and volume. Beyond that, VoiceXML, VoXML, and TalkML provide the infrastructure to define dialogues between the user and the system.

Our system takes a slightly different approach and can actually incorporate these concepts at a lower level. We would like to eliminate the burden of having to specifically design a voice-enabled document. Rather, our system seeks to automatically voice-enable *any* document. Our system actually analyzes the

XML schema and automatically discovers the specification that a developer would otherwise have to write. In fact, our system generates instances of SpeechML.

2.4 Voice Browsing by Phone

We focus on voice web browsing as the primary application for our system. [27] is the W3C note describing necessary features of voice web browsers. Some of the features it describes are alternative navigational mechanisms, access to forms and input fields, and error handling. Other work such as [21, 28] present standards and issues related to voice-enabling the web, however they focus on designing systems with limited functionality. There are a limited and static set of commands that any user can give to the browser.

There is also a body of work that is specifically developing protocols for communication with cellular phones. WAP[31] looks at the network protocol itself while WML[32] is the XML-based language used to communicate on top of the WAP protocol. However, like VoiceXML, VoxML, SpeechML, and the rest, WML requires that the web page designer provide either a manually generated WML version of a web page, or a means of transcoding the page into WML. Both solutions require human intervention. This work does not intend to replace the standards developed in this body of work. Rather, we hope to discover a means to automatically perform the conversion from general XML to a format such as WML or VoiceXML.

Our work seeks to leverage off and generalize the concepts described in the work discussed in this section. We wanted to design a general system to provide an interface to any type of XML document, not just HTML. By doing so, we eliminate the restrictions posed by similar voice browsing systems. Furthermore, we wish to use the findings of previous auralization work to integrate sound and its different components into our generated interfaces. Given any XML document, our system provides a speech-based interface.

3 Design of the System

Existing XML editing and browsing tools are limited in their support. They produce a standard view of *all* XML documents. Every document is presented in the same way. However, we propose that we can take advantage of the knowledge of an XML schema to produce a customized rendering system that will be unique for every different type of document and possibly for every user as well. The primary goal in the design of the MakerFactory was to create a system that would automate the process of generating a usable, customizable interface for an XML document. We not only wanted to address the issue of generating an XML-based web browsing system, we wanted to develop a system that would provide automatic generation of an interface that would allow a user to do anything that she would want to do with an XML document. This could include linking, navigation, creation, and modification of XML documents. The MakerFactory architecture supports this goal. It is completely interactive, customizable, and useful not only for the XML reader, but for the XML writer as well.

Our design considers XLink [34] as the link traversal mechanism. XLink is currently being developed as a standard for integrating linking mechanisms into XML[34, 35] and will likely be the standard implemented by most XML parsers. We focus on developing a voice-enabled mechanism for traversing XML links as defined in the XLink working draft.

3.1 The MakerFactory

The MakerFactory is designed to operate in two phases. The first phase is the code-generation phase. In this phase, the user selects a series of interface generation components from the library provided. To generate a customized interface, the user need only select those components that will be relevant to the run-time rendering scenario. Each component is responsible for *making* a specialized interface for an XML document and hence must implement our Maker interface. In addition, the user may optionally specify a set of customization rules that further define how the document will be rendered. The result of code-generation is a set of Java classes designed to *mediate* communication between the user and the synchronized tree manager. Therefore, the Maker-generated classes should minimally implement our Mediator interface. Since

each Mediator is designed to be independent of the others, the user need only select to invoke the Mediators that are relevant to the current scenario and hence not incur the overhead of having to run all Mediators simultaneously.

The second phase is the run-time rendering phase. The MakerFactory provides a Renderer that is responsible for controlling synchronized rendering of the XML tree. Each Mediator acts as an intermediary between the Renderer and the user allowing its own specialized input and output mode. Specifically, AXL is designed to act as the auditory Mediator, allowing the user to navigate the XML document and traverse its links via a speech-based interface.

3.1.1 Code-Generation

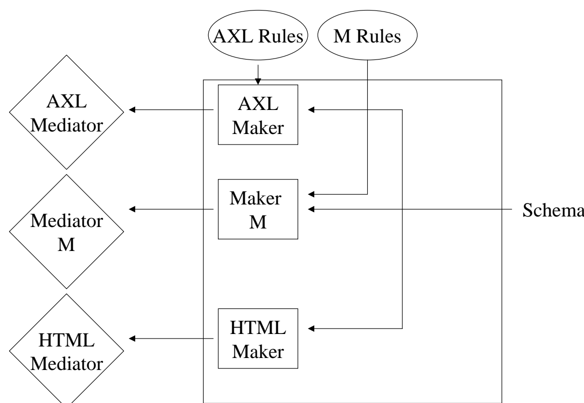

Figure 1: An XML schema is fed into a user-selected series of Makers along with an optional set of customization rules for each type of Maker. The Maker analyzes the schema as well as the rules to produce a customized Java Mediator class for each type of Maker. Each Mediator provides the user with a different input/output mode that can be customized by the user via the optional rules. At run-time, a user-selected set of Mediators are used to mediate communication between the user and the underlying XML structure.

Figure 1 shows the architecture of the code-generation system. A given schema is analyzed and the results of the analysis are passed into a series of user-selected Maker classes. Given the schema and any optional customization rules specified by the user, the system produces a set of customized Mediator classes that can be used with the Renderer to interact with any XML document conforming to the given schema. For example, if the user specifies the AXL Maker (a component used to generate auditory input and output) and the HTML Maker (a component used to produce an HTML representation of the data), the result should be two independent Mediator classes. The AXL Mediator will listen for spoken commands from the user and provide aural output whereas the HTML Mediator may display the XML in HTML format and disallow input from the user. Additionally, if the given schema were a NEWSPAPER with a list of ARTICLES containing HEADLINE and BODY elements, the AXL Maker might determine that the AXL Mediator should render an ARTICLE by rendering the HEADLINE whereas the HTML Maker may determine that the HTML Mediator will render an ARTICLE by displaying both HEADLINE and BODY.

3.1.2 Rendering

The architecture of the Renderer is shown in figure 2. The MakerFactory produces a set of classes that listen for user input and provide directives to the Renderer. When instantiated, the Renderer acts as the

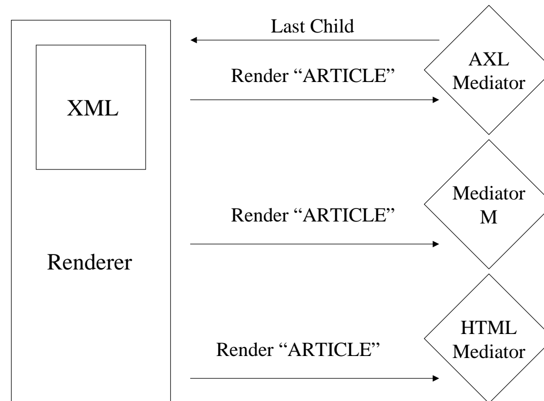

Figure 2: The Renderer controls synchronized access to the XML tree. Each Mediator may receive input from the user in the mode supported by the Mediator. The Mediator interprets the command and issues a corresponding set of commands to the Renderer. The Renderer changes the tree view based upon the command and updates the synchronized view for all other Mediators. Once the Mediator view changes, it may provide output to the user via the Mediator's supported mode of output.

tree manager. It controls synchronized access to the tree by receiving commands from the Mediators and in turn performs the task given by each Mediator and informs all of the other Mediators of the result.

The Renderer defines the concept of a cursor. At any given point, all of the registered Mediators should be rendering the portion of the tree pointed to by the cursor. When the cursor is moved, the new view of the tree should be rendered. However, it is possible that a Mediator will have to move the cursor more than one time to achieve the desired view. This is because the methods to move the cursor are generally incremental and somewhat limited. To accommodate this situation, the Renderer implements a locking mechanism. Before calling a method that will move the cursor, the given Mediator must acquire the lock. After all movement is complete, the lock should be released. When the lock is released, all of the Mediators are notified that the cursor has changed. For example, if a Mediator directs the Renderer to move the current cursor to an ARTICLE node, the Renderer will in turn ask all of the other Mediators to render the given ARTICLE.

3.2 AXL

The primary goal in the design of AXL was to create a component that would produce a usable, voice-based interface to any XML document. There are many questions we sought to answer in conjunction with this goal. First, we wanted to produce user interfaces that would give the user as much control as possible over the navigation of the document. In the case of a large web page, the user would likely want the ability to choose which portion of the page she wanted to hear. However, we recognized that there are situations where the user may actually wish to be the passive participant while allowing the rendering system to navigate and present the XML document. This would likely be the case in the newspaper example. Therefore, we wanted our design to allow the user the freedom to choose the level of interaction she wishes to have with the document.

3.2.1 Code-Generation

In the code-generation phase, the AXL Maker analyzes the schema and produces a vocabulary for interacting with a document conforming to the given schema. For example, the previously cited NEWSPAPER example

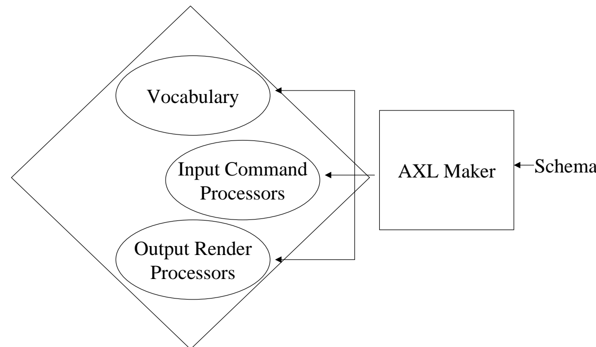

Figure 3: The AXL Maker produces an AXL Mediator by (a) constructing a vocabulary that can be used to access the information in a document conforming to the given schema, (b) constructing a set of classes to listen for spoken commands within that vocabulary and (c) constructing a set of classes that will perform a text-to-speech conversion of the information contained in the appropriate nodes of a given XML subtree.

might generate a vocabulary {article, body, headline}. Each word in the vocabulary has a semantic meaning associated with traversing a document of the given type. The "headline" command may mean to read the headline of the current article. Given that vocabulary, the AXL Maker generates a set of classes that process commands within the vocabulary as well as a set of classes that know how to render nodes of the given schema via a speech synthesizing engine as shown in figure 3. The resulting set of classes compose the AXL Mediator.

3.2.2 Rendering

The AXL Mediator operates as shown in figure 4. When the Mediator is asked to render a node, AXL will determine the type of node it is being asked to render and then perform the appropriate operations. The rendering may include everything from a simple rendering of the node's tag name to performing the action defined in a pre-compiled Java class. Perhaps a HEADLINE node has two parts, one being the title of the ARTICLE and one being the subtitle. The rendering of a HEADLINE may indicate to render first the title and then the subtitle. Correspondingly, perhaps rendering a BODY node simply means to perform some action and provide no output to the user. This situation is depicted in figure 4.

3.3 Link Traversal Support

In order to make the MakerFactory usable as a web browsing system, it is imperative that we support a method of link traversal. At the most basic level, we want to support the kinds of links defined in HTML using the "A" tag and "href" attribute. For XML, the XLink standard is being developed. Not only does XLink provide the mechanism to define HTML style links, it also has the benefit of attaching semantic meaning to the action of link traversal. We investigate how to design our system to support XLink style link traversal.

3.3.1 MakerFactory Linking

In order to support a linking mechanism, the Renderer must provide a mechanism for the Mediator classes to request that a link be followed. We investigate this requirement by looking at the different attributes that

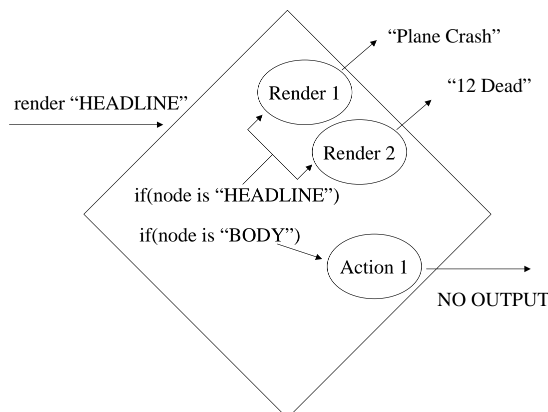

Figure 4: The AXL Mediator receives "render" directives from the Renderer. It determines the node type based upon the tag name and invokes the appropriate set of render and action methods. Each render or action method should provide the appropriate output to the user either by doing a text-to-speech conversion of the information in the subtree or performing a specialized action method.

may occur within the *xlink* namespace. Currently, we are looking at XLink *simple* links. Table 1 outlines XLink defined links an their corresponding AXL interpretation.

The *href* location attribute is necessary for determining where to find the given node. The arc end attributes are not required at this time for the purposes of the MakerFactory. The behavior attributes help the MakerFactory in determining how to load the referenced resource. The resource may be appended to the current tree, replace the current tree possibly causing the system to instantiate a new set of Mediator classes, or may require that a new Renderer be instantiated. Additionally, the Renderer may automatically load the resource, or may wait for a Mediator to explicitly ask to load it. Finally, a Mediator may use the semantic attributes in its own rendering of a node.

3.3.2 AXL Linking

It is the job of AXL to appropriately interpret the MakerFactory supported commands and provide the user with a suitable method of accessing those commands via a speech-based interface. Therefore, AXL must allow the user to issue a command that *follows* a link. If the user has reached a node that is identified as a link, the user may ask to follow the link. AXL will subsequently invoke the Renderer method used to perform the traversal. The result may be a new tree which will then be rendered appropriately by all of the Mediators including AXL.

XLink provides the definition of *role* and *title* attributes. These define the link's function and provide a summary of the link. AXL can take advantage of both of these attributes. If the node currently being rendered is a link node, AXL will render the node by synthesizing both semantic attributes for the user. These attributes provide a built-in way for the XML writer to indicate to AXL the best rendering for the node.

4 Implementation

This section details the implementation of the MakerFactory and AXL. Many of the implementation details of the MakerFactory are more clearly illustrated by using AXL as an example.

The entire system has been implemented in Java using the IBM Speech for Java[24] implementation of

Conceptual Classification	Attribute	Function	MakerFactory Implications
Location	href	provides URI of resource	Renderer uses value to load resource
Arc End	from	defines where link originates	NONE
	to	defines where link traverses to	NONE
Behavior	show	defines how the resource should be presented to the user	**parsed** - resource is appended as child of current node **replace** - resource replaces current data; new Mediators may be instantiated **new** - requires a new Renderer be instantiated; see future work on *Speech Spaces*
	actuate	defines how link traversal is initiated	**user** - a Mediator must request link traversal **auto** - the Renderer traverses the link upon rendering the XLink node
Semantic	role	defines function of link	may be used to enhance rendering
	title	description of link	may be used to enhance rendering

Table 1: The XLink standard defines four classes of attributes. Each type of attribute can be specified in an XLink node and helps the parser to determine how to traverse and display the attribute as well as provides metadata describing the function and content of the link itself. The MakerFactory considers how to integrate all of the possible values of those attributes given the XLink definition of their functions.

the Java Speech API[11]. Furthermore, the Speech for Java implementation requires that IBM's ViaVoice be installed and used as the synthesizing and recognizing engine.

4.1 The MakerFactory

The MakerFactory is invoked using a configuration file defined by an XML DTD. The first piece of information that must be specified is the SchemaProcessor class. For each different type of schema specification(e.g. DTD, XML Schema) a class implementing our SchemaProcessor interface must be written. The appropriate class should be specified to deal with the type of schema used for the current MakerFactory invocation. The second item to be specified is the SchemaLocation. This should be the URL of the schema definition itself(e.g the file name of the DTD file). Finally, for each different type of Maker used for a particular invocation, the Classname of the Maker as well as the location of a Rulefile must be specified.

Each Maker is instantiated and notified of each different type of element defined in the schema. Based upon this information as well as the rules defined in the Rulefile, each Maker generates a set of Java classes that implement the Mediator interface and can be invoked using the MakerFactory Renderer. The Renderer is invoked at run-time with a configuration file similar to that defined by the MakerFactory. The XMLFile to be traversed as well as the set of Mediator classes to be instantiated are specified. The Renderer traverses and modifies the XML document according to the Mediator directives and notifies each Mediator when there as been a change. Table 2 details the Renderer supported methods to be called by the Mediator classes.

All of the methods function like their org.w3c.dom.Node counterparts with the exception of the *setData* method. This method allows the user to change the tree that is being traversed. This is especially useful for link traversal where the current document changes often.

The remainder of the implementation including the rule language is best explained by examining AXL, the SpeechMaker.

Method	Behavior	Return Value
appendChild(Node newChild)	appends the newChild as the last child of the current element	true if the append was successful
attribute(String attrname)	moves the current attribute cursor to the attribute of the current element with the given name	true if the move was successful
element(String eltname, int n)	moves the current element cursor to the nth occurrence of element "eltname"	true if the move was successful
firstChild()	moves the current element cursor to the first child of the current element	true if the move was successful
insertBefore(Node newChild)	inserts newChild as the previous sibling of the current element node	true if the insert was successful
lastChild()	moves the current element cursor to the last child of the current element	true if the move was successful
name()	gets the tag name of the current element	String - tag name of the current element node
nextAttr()	moves the current attribute cursor to the next attribute of the current element	true if the move was successful
nextSibling()	moves the current element cursor to the next sibling of the current element	true if the move was successful
parent()	moves the current element cursor to the parent of the current element	true if the move was successful
previousAttr()	moves the current attribute cursor to the previous attribute of the current element	true if the move was successful
previousSibling()	moves the current element cursor to the previous sibling of the current element	true if the move was successful
remove()	removes the current element and moves the current element cursor to the parent	true if the remove was successful
replaceChild(Node newChild)	replaces the current element with newChild	true if the replace was successful
setAttr(String name, String value)	sets attribute name of the current element to the value	true if the set was successful
setData(Document newDoc)	replaces the XML tree with newDoc and renders the root	true if the set was successful
value()	return a cloned copy of the current element node and its children	Node - cloned copy of the current element and its children

Table 2: Methods supported by the MakerFactory Renderer, their functions, and what they return to the calling Mediator. Each Mediator should implement a mode-specific way of invoking this set of methods. Each user-issued command should be transformed into a call to one or more of these methods and the result should be an update to the view held by each Mediator.

4.2 AXL

At the lowest level, AXL provides an auditory version of the Document Object Model(DOM). DOM is a parsing model that parses an entire XML document into a tree structure and provides a series of methods to navigate within the XML tree. Beyond an auditory DOM, AXL provides a customized interface through schema analysis, and also via user-customization.

4.2.1 Speech DOM

At the most basic level, AXL provides a speech version of the Document Object Model[5]. At the core level, an XML tree is a collection of org.w3c.dom.Node objects. Therefore, the AXL Speech DOM provides a set of commands conforming to the org.w3c.dom.Node API.

The basic commands available are:
Parent
Next Sibling
Previous Sibling
Next Attribute
Previous Attribute
First Child
Last Child
Name
Value
Remove
Insert Before
Append
New Text
Set Attribute

For each command, the Mediator simply calls the corresponding method of the Renderer class(see table 2). The only commands that warrant explanation are *Insert Before*, *Append*, *New Text*, and *Set Attribute*. Each of these commands requires more input than the command itself. *Insert Before* causes the system to beep indicating that it is waiting for more user input. It waits for the user to speak the tag name of the node to be inserted. When it hears the name it inserts the node and returns to normal operation mode. *Append* is exactly the same, except that the new element is appended as the last child of the current node rather than inserted as the current node's previous sibling. *New Text* and *Set Attribute* both require actual dictation from the user. Therefore, the *New Text* command causes the system to beep indicating that it is waiting for free dictation. It records everything the user speaks until the user issues the command *Complete*. At that point, all of the text is appended as the last child of the current node. Finally, *Set Attribute* is much like the *New Text* command. However, there are two parts to the new node, the name and the value. Therefore, *Set Attribute* puts the system in the dictation mode. The first word spoken in this mode is interpreted as the name of the attribute. The remainder of the spoken phrase until the *Complete* command is issued is interpreted as the value of the attribute. For example, "*Set Attribute* [Title] [A Midsummer Night's Dream] *Complete*" would set the Title attribute of the current element to be "A Midsummer Night's Dream".

In addition, AXL defines a set of commands used by the engine itself . Table 3 illustrates what the user may say to direct the engine.

The AXL engine can be in one of four states as shown in figure 5. The *Normal* state simply waits for another command. If in the *Tag Specification* state, the engine expects a valid tag name for the given schema. The *Paused* state means that the engine will not listen to any user input except for the *Resume* command. Finally, the *Dictation* state indicates that the engine is waiting for free dictation from the user.

This basic implementation does not require a schema or user-specified rules. In fact, this interface can be created one time and used to navigate any XML document. However, it is not enough to provide a generic, low-level interface to an XML document. We would like to leverage off of the functionality provided by the Speech DOM, and build a higher-level interface on top of it to give users a more intuitive way to navigate a

Command	Behavior
Complete	ends dictation mode
Pause	pauses input; engine only recognizes Resume command
Resume	ends paused state; engine listens for all commands
Command List	lists the set of commands available at the current node
Save	saves the current XML tree
Stop	stops the rendering of the current node
Exit	performs any necessary cleanup and exits the program

Table 3: AXL defines a set of commands to direct the engine itself. Each command tells the engine to perform a function independent of the underlying document. This command set should not result in an update to the view of the XML structure.

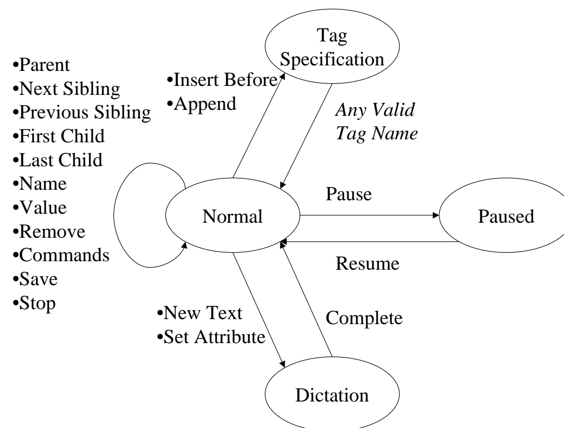

Figure 5: The AXL engine can be in one of four states. The transition from one state to another is initiated by a spoken command from the user. To transition from *Normal* to *Paused*, the user speaks the command *Pause*. *Resume* returns the system to normal operation. *Dictation* mode is entered by speaking either the *New Text* or *Set Attribute* command. After dictation is finished, *Complete* returns the system to normal operating mode. *Tag Specification* is entered by either the *Insert Before* command or the *Append* command. Once the tag name is spoken, the system automatically returns to the *Normal* state. The remaining commands cause the system to perform some action which may or may not result in output to the user, however the system itself remains in the *Normal* state.

document. Ultimately, given an XML document conforming to one schema, we would like to interact with it in a way specific to that schema. Given another document conforming to a different schema, we would like for the user-interface to be different.

4.2.2 A Schema-Driven Interface

Primarily, the DTD tells us what kind of children and attributes an element is allowed to have. The first task is to use that information to provide a rendering of the node. For example, in addition to rendering the node by speaking the tag name, we can also speak the names of the children and attributes that the node is allowed to have. For example, the rendering of an ARTICLE node might be "An ARTICLE is a HEADLINE and a BODY". This provides the user with a brief overview of the structure at any given node without having to know the structure before hand.

The second task is to use the schema derived information to produce a more intuitive vocabulary with which we can interact with the document. First, we can use the knowledge of the elements and attributes that exist in the document to develop a vocabulary. Suppose that a user wished to navigate to the BODY element of an ARTICLE. Instead of navigating all of the children using the Speech DOM vocabulary, we allow the user to actually speak the command "BODY". This is more intuitive for the user and also provides faster access over the iterative DOM methods. In addition, developing such a vocabulary allows us to easily disallow illegal directives. If the user asks for the NEWSPAPER node while at the BODY element, we can immediately give the user feedback indicating that no such node exists without having to actually look for the node in the subtree. This creates a more efficient system.

Even though this design allows us to query using a higher-level language, it still does not provide all of the facilities that one might desire. The user may want to have more control over how each type of node should be rendered. The automatically generated interface may not be sufficient for all different uses of an interface. In this case, user input is required.

4.2.3 User Customization

There are many reasons why a user might want to specify how they would like a document to be rendered. The complete semantic definition of the structure might not be specifiable in an XML schema. Furthermore, the user may just have a different preference than that which is defined by the default AXL Maker. Therefore, AXL defines its own rule language, the Audio Browsing Language (ABL). ABL allows a user to provide input about the rendering of the element and attribute nodes in the XML tree.

The general MakerFactory architecture provides a high-level rule language that we can use to specify general customization rules that direct how rendering should occur. For each type of element or attribute, a rule can be defined that specifies the rendering of that element or attribute. Attached to each rendering is an optional condition that specifies when the rendering should be invoked. A rendering will be invoked if the condition is true or if no condition is specified. To define a condition, an implementation of a generic UnaryPredicate interface must be written as a Java class and specified as the condition in the rulefile. The UnaryPredicate takes a single argument, in this case the node to be rendered, and returns a boolean value indicating whether or not the condition has been met. At run-time, the class is loaded and the condition evaluated. Additionally, each different rendering specifies a depth to which the node should be rendered. The depth specifies any combination of one or more of the node, its children, and/or its attributes. For example, the value ATTRS_NODE would render the attributes of the node and then the node itself. For the attribute rules, the options are one or more of the attribute's name or its value.

Finally, the rendering rule for a node may specify an action. To specify an action, the user must specify the name of a Java class that will be instantiated and invoked if the given node type is encountered. All action classes should implement the following interface:

```
public interface Action {
 public void notify(Node node);
}
```

Like the Java event model[10], a render *event* causes the *notify* method of the Action class to be invoked. The current node to be rendered is passed as an argument. If the action modifies the node or its subtree, the Renderer should be notified of the change following the completion of the notify method.

ABL also defines its own set of rules specific to a speech-based rendering. First, because tag names are not always in an understandable format ABL allows the user to specify the output and input phrases that represent each type of tag. The rendering of a node may be defined as the tag name of that node. Similarly, to access such a node, the user may need to speak the tag name of the node. However, there may be some tags that a user simply cannot speak. Suppose the HEADLINE tag were actually specified as HDLN. The user can define how the tag should be spoken by the system as well as by the user. The user can specify that upon seeing a HDLN node, render "HEADLINE". Also, upon receipt of a "HEADLINE" command, look for a HDLN node.

Another function provided by ABL is to allow the user to specify a phrase that may be spoken before and after a node, attribute, or child node. These phrases should be defined such that nodes can be connected together with a more sentence-like flow. Finally, ABL defines a way to change how the AXL default commands are spoken by the user. For example, if a user wanted the command "this node" to retrieve the name of the current node rather than "name", that change could be easily specified in the rule file.

5 Voice-Enabling the Web Using AXL

This section details AXL's application as a front-end component enabling web browsing by voice and looks at AXL as a way to read the morning newspaper in specific. Most current voice browsing technology focuses around browsing the web via a cellular telephone. Cellular phones are becoming ubiquitous. People use their phones while driving, shopping, even while at the movie theater. Moreover, current work looks at using technologies such as VoiceXML to extract web content and push a static view of information such as stock quotes or the weather to the user on the other end of the wireless network.

This work seeks a more generalized view of the phone-web. Push technology[15, 30] is not sufficient for the user that wants control over their own web browsing. We want to enable a user to access any web page and more specifically, we want to provide full interaction for the user to be able to navigate the page. By using AXL as the front-end system(figure 6), a user can make a request to the AXL Renderer which contacts the XML server, caches the document, and navigates it as the user directs.

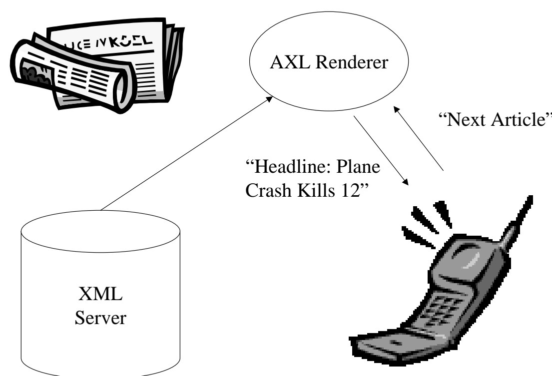

Figure 6: The Renderer, using AXL, caches the XML document from a given server. The user navigates the document by speaking commands and the Renderer responds by doing a text-to-speech conversion of the data in the document. If the user chooses to traverse a link, the Renderer will contact the server to cache the new document.

5.1 The Daily News

To illustrate, we use the scenario in which a person wishes to browse their morning newspaper online. AXL not only allows a user to navigate the paper document, the user can customize a rendering of the paper so that they hear their favorite sections in the order that they would normally read the non-electronic version. We use the following DTD influenced by[16]:

```
<!ELEMENT NEWSPAPER (ARTICLE|AD)+>
<!ELEMENT ARTICLE(HEADLINE, LEAD, BODY, NOTES)>
<!ATTLIST ARTICLE
                    AUTHOR CDATA #REQUIRED
                    EDITOR CDATA #IMPLIED
                    DATE CDATA #IMPLIED
                    EDITION CDATA #IMPLIED>
<!ELEMENT HEADLINE (#PCDATA)>
<!ELEMENT LEAD (#PCDATA)>
<!ELEMENT BODY (#PCDATA)>
<!ELEMENT NOTES (#PCDATA)>
<!ELEMENT AD EMPTY>
<!-- An AD is a link to a subtree containing
     information about the company being advertised. -->
<!-- The attributes of an AD are those link attributes
     detailed in table 1. -->
<!ATTLIST AD
                    HREF CDATA #IMPLIED
                    SHOW CDATA #IMPLIED
                    ACTUATE CDATA #IMPLIED
                    ROLE CDATA #IMPLIED
                    TITLE CDATA #IMPLIED>
```

At the most basic level, AXL would render the root NEWSPAPER node as "Newspaper". The user could then issue the command "Article 1" which would move the current cursor to the first article in the paper. Similarly, the user can ask for the nth article and the cursor will correspondingly move to the nth article. Once the cursor moves, the ARTICLE node is rendered as "Article". At the ARTICLE node, the user has the choice of asking for the HEADLINE, LEAD, BODY, or NOTES. Similarly, the user can ask for the AUTHOR, EDITOR, DATE, or EDITION attributes. Any of these commands will result in the movement of the current cursor to the corresponding node and the system reading the tag name and contents of that node. For example, the command "Body" will move the current cursor to the BODY element and will read the body of the article.

Additionally, AXL provides the user with the ability to traverse links. Given the plethora of advertisements that exist in traditional newspapers, it is not unrealistic to think that on-line newspapers will adopt the same format. Suppose that an *eNewspaper* were littered with *eAdvertisements*. In this example, the AD element is simply a link to another tree containing the body of the ad. If the user wanted to follow an AD link, the user would only have to traverse the tree to the AD node, and then issue a *Follow* command. If the SHOW attribute had the value *parsed*, the advertisement body would simply become part of the NEWSPAPER tree. However, a value of *replace* or *new* would either replace this tree or open a new document containing just the body of the ad.

AXL also holds the benefit of customization. Suppose that the user always wanted to hear all of the headlines before reading any of the paper. A simple set of rendering rules could be written that indicated that the rendering of the NEWSPAPER element included rendering the HEADLINE element of each ARTICLE. An example rule specification follows:

```
<RenderRules>
  <ElementRule elementid="NEWSPAPER">
    <ElementRendering>
      <Render depth="CHILDREN_ONLY"/>
    </ElementRendering>
  </ElementRule>
```

```
  <ElementRule elementid="ARTICLE">
    <ElementRendering>
      <Render depth="CHILDREN_ONLY" cond="isHeadline"/>
    </ElementRendering>
  </ElementRule>
</RenderRules>
```

The NEWSPAPER element is rendered by rendering all of its ARTICLE children. Each ARTICLE is rendered by rendering all of its children that meet the *isHeadline* condition. It is assumed that *isHeadline* is the name of a Java class that will check the node to make sure that its tag name is "HEADLINE". Once the user hears the headline that she is interested in, the user issues the command "Stop" which stops the current rendering and listens for a new command. The user then navigates to the body of the desired article.

AXL also allows the user to modify the current document. For some types of documents, it may be sufficient for the user to have read-only access. For the most part, the user does not need to modify a newspaper document. However, suppose that a user was listening to the classified ads looking for a new house and wanted to annotate a given ad. AXL would allow the user to insert a text node with an annotation indicating the given ad was appealing. Moreover, there are lots of web pages that require user input. For example, suppose that a search engine expressed its search form in XML. The user could create a new search node with the text to use for the search and execute the search all without access to a keyboard.

6 User Experience

Our main goal was to produce a system that would be usable for people with a variety of skill levels. We hoped that the command set we provided was intuitive enough to allow any user to extract information from an XML document without knowing about the underlying data representation. Further, we hoped that any user would be able to extract information from the document with a minimal learning curve. Finally, we hoped that users would be able to understand the context of the information they received from the document.

6.1 Design of the User Tasks

The study we conducted asked a set of users to navigate and extract information from an XML document outlining possible travel plans. The concept was that a user could plan a vacation online by navigating such a document to find the necessary information and choose a destination, find a flight to that destination, etc.

A total of eight users participated. Our subject set included both male and female users, users who were both familiar and not familiar with XML, and users with non-American accents. First, the users were asked to read six words and two sentences as minimal training for the ViaVoice system. Then, the subjects were given a brief set of oral directions and were shown the structure shown in 7 and told that the document they would be navigating would follow the same structure.

Following the oral directions, any questions were answered and the user was given a set of written instructions that contained the commands to issue to navigate within the document. The user was then expected to execute each instruction without help from the proctor.

The first set of written directions was intended to familiarize the user with the system and the commands provided. The directions led the user down one path of the tree to a *Flight* element and investigated all of the information at the leaf level including price, destination, and airline. The users then continued to the second set of instructions which were designed to test whether or not the user could remember and apply the commands from the first section and successfully navigate the structure without explicit directions. The instructions asked the user to navigate from the root of the tree to a sibling *Flight* element and retrieve the same information retrieved in the first set of directions. The third set of directions led the user down a different path of the tree to the *Rental Car* element and tested the user's ability to find the price without explicit instructions on how to find the price of a *Rental Car*. The final set of instructions asked the user to navigate the tree to a *Vacation Package* element and asked them to insert a new node into the tree. The intention was to examine the user's ability and reaction to modifying the XML structure.

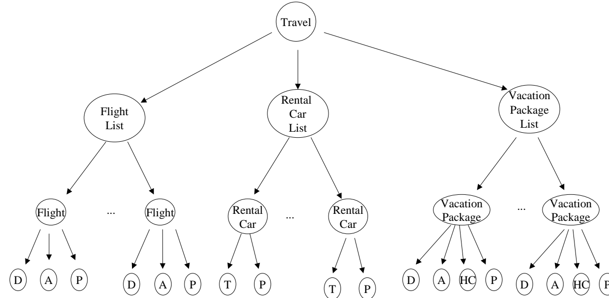

Figure 7: A tree representation of a travel document.

Finally, the users were asked a series of questions about their experience using the system. The questions involved rating and commenting on the command set provided, the appropriateness of the feedback provided by the system, and the user's prior experience including familiarity with XML.

6.2 Observations

We made five main observations:

1. The automatic generation of a command set based upon schema yields commands considered natural by users.
2. The limited vocabulary produced by schema analysis aids the ViaVoice recognition engine.
3. The users' frustration with the rendering of some of the nodes facilitates the need for customization rules.
4. There is a need to develop enhanced feedback mechanisms.
5. There is a need to develop a better *help* command.

6.2.1 The Command Set

We observed the users' perceptions of the command set in two ways. First, the initial set of questions we asked the subjects was designed to probe the user's opinion of the commands provided. We wanted to know how natural or intuitive the commands were for the user and whether or not it was difficult to understand the function of any of the commands. Moreover, we wanted to know if the subjects felt that the command set provided was complete or if the users found themselves in want of different commands. Second, the directions provided included instructions to access information, but allowed the user the freedom to choose between a Speech DOM level command or a schema based command. For example, the users were asked to get the price of a given flight, but were able to choose between the command *price* and *last child*. We expected that the Speech DOM level commands would seem less natural to users and that the commands generated through schema analysis would be more natural.

We found that the users were generally pleased with the command set provided. As we had anticipated, most users indicated that the natural commands were the generated commands like *price* and *destination* while commands like *first child* and *parent* were less natural. When able to choose between a Speech DOM

command and a schema based command, most users chose the schema based commands(i.e. *price*). However surprisingly, one user did choose to use a Speech DOM command. We propose that those familiar with tree structures and the terminology related to them would be more inclined to use the Speech DOM commands. However, more novice users find the more natural commands most useful.

6.2.2 Recognition Accuracy

We were somewhat skeptical of the recognition accuracy of the ViaVoice system. It has been given accuracy ratings in the 85 percent range[26], but tends to require extensive training in order to perform well. However, we were encouraged to discover that with only two sentences of training, the system was able to understand a variety of people who had a variety of accents with high accuracy. We attribute this benefit to the restricted vocabulary used by the system. Because there are only a limited number of commands a user can speak, ViaVoice performs much better than in a general dictation scenario.

There were some instances of misunderstanding however. Background noise and misuse of the microphone(i.e. touching the microphone causing extra noise) accounted for some instances of the system not hearing or mishearing the user. In some cases, the system's complete lack of response caused the users frustration. We attribute this behavior to the speech recognition software itself.

6.2.3 The Need for Customization Rules

The study itself did not provide customization rules nor did it allow the users to write their own customization rules. Therefore, the rendering of all elements was simply the rendering of the tag name or the tag name and text data if the element had the content model (PCDATA). Many users expressed frustration with the fact that when they asked for "Flight eight", that is to say that they moved the current cursor to the eighth flight element, the response from the computer was simply "Flight". Many of the users wanted the computer to say "Flight eight". One user even suggested that when she asked for the "Flight List" the response should be the price, destination, and airline of each flight in the list.

Despite the users' frustrations, this observation indicates that our customization language is imperative to allow users to easily specify their preferred method of rendering. By writing a simple set of customization rules using the customization language we have designed, the user could easily program the system to render each node in a more descriptive manner. A more extensive user study will evaluate the customization language in place by asking the users to write customization rules themselves.

6.2.4 System Feedback

AXL attempts to provide feedback by using different sounds to convey different errors. In the initial instructions given to the users, they were told what each of the sounds meant, but most were unable to recall the meaning when they encountered the sounds while later traversing the document.

We believe that more extensive research must be done to determine which sounds are best suited for providing error feedback. In this case, one uniform beep would likely have been less confusing than the three different sounds the system emitted. However, three different sounds could be most useful if the sounds were carefully selected. Moreover, users expressed frustration in general with the feedback of the system in terms of communicating positioning within the document. Based upon this observation, we would like to undertake a more exhaustive study to examine how to aurally communicate a relative position within a document in an understandable way.

6.2.5 A Help Command

The only help facility provided by the system is the *command list* command which lists all possible commands at the given node including all schema based commands as well as all Speech DOM commands. We made a

couple of observations based upon the use of this command.

First, users expressed the desire for two separate help commands, one to hear the schema based and another to hear the Speech DOM commands. Some wanted to hear only the schema based commands, but were inundated with all of the possible commands for the system. Second, one user expressed frustration with the fact that the commands listed were all *possible* commands. She wanted to hear only the commands that were valid for the given node in the given subject tree. For example, if an optional node were absent in the subject tree, the command to access that node would not be given.

We can first conclude that a thorough help facility needs to be integrated into the system. Because the system is designed to be used by a person who cannot see the document or its structure, we need to examine how to best communicate possible navigation paths and how to access them using only auditory input and output.

However, we also anticipate that there is a steep learning curve related to AXL. If each user underwent a short period of AXL training, they would be less likely to need such a help facility. Moreover, we predict that more advanced users would be likely to use the Speech DOM level commands more often simply because they would not want to learn a new set of commands for every new schema.

We can conclude from this study that a more extensive study must be undertaken in order to fully analyze the behavior of the system and of the users. We predict that the results would change considerably if the users were given a practice document first. Moreover, we would like to examine the effects of using elements versus attributes in the structure of the document. We predict that attributes would be easier for a user to navigate since the structure is flatter.

7 Conclusions and Future Work

Current work on AXL focuses on integrating the system into a variety of applications. One important use of AXL is as the front-end application of the Aurora[7] project. Aurora seeks to make the web more accessible to people with disabilities by using XML to simplify the services people access over the web. By integrating AXL as the user interface, we can provide a comprehensive solution to allow print disabled people easy access to web data and services.

We are also investigating the integration of AXL and the Open eBook Standard[18]. Unlike the traditional "book on tape" scenario, an AXL interface to an eBook document would give the user the opportunity to navigate within the book. This would eliminate the restriction posed by the serial nature of audio. Moreover, by integrating the multi-modal nature of the MakerFactory system, a user may choose to switch between listening, reading, and taking notes in her eBook.

Finally, in order to fully implement the concepts defined by XLink[34], we introduce the concept of *Speech Spaces*. If multiple Renderers exist simultaneously, the AXL Mediator classes attached to a given Renderer should only listen for user commands that fall within the scope of the given Renderer. By attaching a prefix command to any command in the grammar of a given AXL Mediator, only commands beginning with the given prefix will be processed.

XML promises to be one of the most popular ways to store and exchange data. It holds a host of benefits, not the least of which is allowing non-traditional access to the World Wide Web. As XML becomes more universal, there is a desperate need to develop tools that will allow all users to access and update XML data in traditional and non-traditional scenarios. This work provides an infrastructure to develop usable, customized ways to view and modify XML structures. Furthermore, by supporting the XLink standard, we enable users to navigate not only the current document, but to traverse across links as well. As the world becomes more and more computerized, our infrastructure can be used and extended to provide essential, non-traditional methods of human-computer interaction.

References

[1] M. Albers. Auditory Cues for Browsing, Surfing, and Navigating. In *Proceedings of ICAD 1996 Conference*, 1996.

[2] S. Barrass. Auditory Information Design. PhD Thesis, Department of Computer Science, Australian National University, 1997.

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

[4] S. Brewster. Providing a Structured Method for Integrating Non-Speech Audio into Human-Computer Interfaces. PhD Thesis, Department of Computer Science, University of York, August 1994.

[5] Document Object Model (DOM) Level 1 Specification Version 1.0 W3C Recommendation 1 October, 1998 *http://www.w3.org/TR/REC-DOM-Level-1/*

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

[7] A. Huang and N. Sundaresan. Aurora: Gateway to an Accessible World Wide Web. *IBM Technical Report*, December, 1999.

[8] F. James. Lessons from Developing Audio HTML Interfaces. *ASSETS 98*, April 1998, pp. 15–17, .

[9] F. James. Representing Structured Information in Audio Interfaces: A Framework for Selecting Audio Marking Techniques to Represent Document Structures. PhD Thesis, Department of Computer Science, Stanford University, Palo Alto, CA, June 1998.

[10] Java Beans Specification Document. ftp://ftp.javasoft.com/docs/beans/beans.101.pdf

[11] Java Speech. http://java.sun.com/products/java-media/speech/

[12] M. Krell and D. Cubranic. V-Lynx: Bringing the World Wide Web to Sight Impaired Users. *ASSETS 96*, Vancouver, British Columbia, Canada, April 1998, pp. 23–26, .

[13] L. Lamport. *LaTex: A Document Preparation System.* Addison-Wesley, Reading, Mass. 1986.

[14] The Voice: Seeing with Sound http://ourworld.compuserve.com/homepages/Peter_Meijer/

[15] Motorola Introduce Voice Browser. http://www.speechtechmag.com/st16/motorola.htm, December/January, 1999.

[16] Newspaper DTD. http://www.vervet.com/Tutorial/newspaper.dtd

[17] The Next Net Craze: Wireless Access. http://www.businessweek.com/bwdaily/dnflash/feb1999/nf90211g.htm

[18] Open eBook Publication Structure 1.0 *http://www.openebook.org/OEB1.html*

[19] T. Oogane and C. Asakawa. An Interactive Method for Accessing Tables in HTML. In *Proc. ASSETS '98*, Marina del Rey, CA, April 1998 pp 126-128.

[20] S. Portigal. Auralization of Document Structure. MSc Thesis, Department of Computer Science, University of Guelph, Canada, 1994.

[21] Position paper - Standards for voice browsing. http://www.w3.org/Voice/1998/Workshop/ScottMcGlashan.html

[22] T.V. Raman. Audio System for Technical Readings. PhD Thesis, Department of Computer Science, Cornell University, Ithica, NY, May 1994.

[23] K. Sall. XML Software Guide: XML Browsers. http://www.stars.com/Software/XML/browsers.html

[24] SpeechML. http://www.alphaworks.ibm.com/formula/speechml

[25] TalkML. http://www.w3.org/Voice/TalkML/

[26] Voice Recognition Software: Comparison and Recommendations http://www.io.com/ hcexres/tcm1603/acchtml/recomx7c.html

[27] Voice Browsers. http://www.w3.org/TR/NOTE-voice

[28] Voice Browsing the Web for Information Access. http://www.w3.org/Voice/1998/Workshop/RajeevAgarwal.html

[29] VoiceXML. http://www.voicexml.org/

[30] VoXML. http://www.voxml.org/

[31] Wireless Application Protocol White Paper, October 1999. *http://www.wapforum.org/what/WAP_white_pages.pdf*

[32] Wireless Application Protocol Wireless Markup Language Specification Version 1.2, November 1999. *http://www.wapforum.org/what/technical/SPEC-WML-19991104.pdf*

[33] C.M. Wilson. Listen: A Data Sonification Toolkit. M.S. Thesis, Department of Computer Science, University of California at Santa Cruz, 1996.

[34] XML Linking Language (XLink). http://www.w3.org/TR/WD-xlink

[35] XML Pointer Language (XPointer). http://www.w3.org/TR/1998/WD-xptr-19980303

[36] M. Zajicek, C. Powell and C. Reeves. A Web Navigation Tool for the Blind. *3rd ACM/SIGAPH on Assistive Technologies*, California, 1998.

[37] M. Zajicek and C. Powell. Building a Conceptual Model of the World Wide Web for Visually Impaired Users. In *Proc. Ergonomics Society 1997 Annual Conference*, Grantham, 1997.

Vitae

Sami Rollins is a PhD student at the University of California at Santa Barbara in the Networking and Multimedia Systems Lab. She completed an industrial Master's thesis at the IBM Almaden Research Center on the topic of using XML to produce usable speech-based interfaces. Her previous XML related work includes a visual, schema-driven XML editing tool as well as XML to XML transcoding engine. She has also done work in the areas of global computing and multicast content scheduling. Her interests include HCI, Internet technologies, and networking.

Dr. Neel Sundaresan is a research manager of the eMerging Internet Technologies department at the IBM Almaden Research Center. He has been with IBM since December 1995 and has pioneered several XML and internet related research projects. He was one of the chief architects of the Grand Central Station project at IBM Research for building XML-based search engines. He received his PhD in CS in 1995. He has done research and advanced technology work in the area of Compilers and Programming Languages, Parallel and Distributed Systems and Algorithms, Information Theory, Data Mining and Semi-structured Data, Speech Synthesis, Agent Systems, and Internet Tools and Technologies. He has over 30 research publications and has given several invited and refereed talks and tutorials at national and international conferences. He has been a member of the W3C standards effort.