UNIVERSITY OF CALIFORNIA

Santa Barbara

Audio XmL: Aural Interaction with XML Documents

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in
Computer Science
by

Sami Rollins

Committee in charge:

Professor Kevin Almeroth, Chairperson

Professor Divyakant Agrawal

Professor Peter Cappello

Doctor Neel Sundaresan

March 2000

The Thesis of Sami Rollins is approved

Committee Chairperson

March 2000

Acknowledgments

I wish to thank Neel Sundaresan for his willingness to give me much needed guidance toward a research topic and his patience in dealing with the trials and tribulations associated with doing an industrial Master's thesis. I am fortunate to have had the experience of doing academic research in an industrial setting. The freedom and support Neel has given to me has helped me to triumph despite roach infestations, falling trees, leaping deer, multiple car loans, and unsigned CDAs.

I also wish to thank Kevin Almeroth for supporting my decision to complete an MS and for dealing with the beurocracy on my behalf. His advice about what it means to be a graduate student has given me much needed perspective on my situation and his time and effort in dealing with the admistrative side of this thesis has been integral in facilitating its completion.

Divy Agrawal, Mary Jane Archenbronn and Amy Mills also deserve much gratitude for the time and energy they have invested to make sure everyone has signed on the dotted line.

I would like to thank my brother Joe Rollins and my mentor Barbara Li Santi for their encouragement and for assuring me that my experience as a first year grad student at UCSB was not unique. Also, I would like to thank Ellen Spertus for helping me to see all of the options available to me as a computer scientist. And to my mother Nina Rollins, and to everyone who has listened to the trials and tribulations of the past two years of my life, thank you.

Abstract

Audio XmL: Aural Interaction with XML Documents

by

Sami Rollins

Computers have become ubiquitous. Not only is there a computer on every desk, as pervasive computing technology explodes, anyone anywhere may have access to the power of a processor and be connected to the Internet. 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 interact with their computers in non-traditional ways. Auditory interfaces for computers are not only gaining popularity among the users of devices such as cellular telephones, they have always been imperative for print disabled users. However, creating usable, intuitive ways for people to interact with computers using speech as the input and output mode presents a host of challenges.

Our goal was to design a system that would overcome the barriers associated with current audio-based human-computer interaction mechanisms. Current tools such as screen readers present a flat view of data with limited facilities for interaction. We propose that by representing electronic data in a semi-structured way, we can exploit the inherent structure to provide customizable, intuitive interfaces that allow navigation, rendering, and modification of data. The implications of such a technique enable users to access electronic information and services regardless of physical disabilities or restrictions imposed by circumstance.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

As the world flourishes in the proverbial *sixth generation* of computing technology, we not only choose to use computers for activities like surfing the web and keeping in touch with friends and family, we are required to use computers in our jobs, to access our bank accounts, even to place an order at a fast food restaurant. Computers themselves present tremendous opportunity for all users alike. The potential for user-specific representations of the same underlying data, be it an electronic textbook, or a web page from which a user can order dinner, means access for *all* possible users. However, the full extent of that potential has not yet been realized.

The WIMP(windows, icons, menus, pointing devices) model is no longer sufficient for input and output. As pervasive computing technology explodes, anyone anywhere may have access to the power of a processor and be connected to the Internet. 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 interact with their computers in non-traditional ways. Moreover, historically the WIMP model has posed problems for print disabled users[15]. The graphical representation of data on the web and desktop alike prevents many users from accessing the real content underlying the icons and buttons. Such limitations demand technological innovations that will allow users to access data using speech as the input and output mode.

The development of an aural means of human-computer interaction presents a host of challenges. First, audio is serial. By nature, a listener must assume a passive role where a sighted viewer takes a more active approach[38]. For example, screen reader that trivially reads a flat piece of text does not give a listener abilities afforded a viewer such as freedom to navigate. A listener cannot scan the information, move back or forward, or imply an underlying structure based on attributes such as font size or page layout. In order to provide a listener with as much control as a viewer, an aural interface must provide a mechanism to allow users to scan information and actively select portions of the document to be read.

Further, it is difficult at best to represent graphical UIs using speech[12]. The current solution employed by most screen readers is to rely on "alt" tags to provide information about the content of the images, although a general solution does not exist. However, by separating the content of the data from the presentation of that data, an interface can display the underlying information in the most appropriate manner. Information need not be represented *only* using graphics. The concept of multiple representations of the same data is also useful when an environment must be multi-modal as could be the case with a visually impaired student and a sighted teacher.

Finally, it is imperative to avoid the cumbersome task of generating a specialized interface for different types of data. Solutions like VoiceXML[55] are useful, but require manual intervention when new data need to be communicated. An all-purpose solution should provide an automatic way of generating a data to speech conversion.

The eXtensible Markup Language(XML)[13] is emerging as a new standard that provides a common format for storing and communicating data. It not only allows users to separate underlying data from its presentation, its semi-structured nature can be used to represent information in such a way as to emulate a visual scan. By representing data in XML format, we wish to devise a method that will provide the means for a user to interact with data using the input and output modes that are most appropriate for that user. Moreover, XML holds the benefit of schema definition. By analyzing a schema, we can *automatically* decide how to interact with the data using speech.

1.2 The eXtensible Markup Language

The eXtensible Markup Language(XML)[13] is gaining popularity as a paradigm for data representation. XML is a standard way to separate underlying data (i.e. the chapters of a textbook or the content of a web page) from the representation of the data (i.e. fancy graphics or just plain text). XML allows a user to define the structure of a document and hence add semantic information to the data itself. By providing the user that freedom, the user can store the data in such a way that it can be later retrieved in a manner convenient to the user's current set of circumstances.

Like its predecessor, the Standard Generalized Markup Language(SGML)[43], XML uses *tags* to add a semantic, hierarchical structure to the data itself. A *start-tag* marks the beginning of an XML *element*. A start-tag consist of a left angle bracket(<) followed by a *tag name* that describes the content of the element, followed by a right angle bracket(>). The start tag is followed by the content of the element itself. An *end-tag* marks the end of the element. The end-tag looks similar to the start-tag and should have the same *tag name*, however a forward slash(/) follows the left angle bracket. The following example shows a "Name" element with two element children, "First" and "Last".

```
<Name>
  <First> John </First>
  <Last> Doe </Last>
</Name>
```

The semantic structure implies that the data "John" is a *firstname* while the data "Doe" is a *lastname*. However, the same data might have a very different meaning if surrounded by different semantic tags.

The third type of tag is the *empty* tag. An empty tag is used if there is no content for the element and is equivalent to a start-tag immediately followed by an end-tag. An empty tag looks much like a start-tag, except that the right angle bracket is preceded by a forward slash(/). An empty tag can be useful if an element contains *attributes*. An attribute is a *<name, value> pair* where the value is the data itself and the name is a semantic description of the value. The data from the previous example can also be represented using attributes and an empty tag. The following example illustrates:

```
<Name First=''John'' Last=''Doe''/>
```

In this case, "Name" is an empty tag even though it contains attributes. An empty tag can also exist without attributes. For example, between the <First> and <Last> tags of the first example, the empty tag <Middle/> would indicate that a middle name could be present, but is not present in the given case. For a more thorough examination of the use of elements versus attributes, refer to [42].

A benefit of XML is that it allows the user to define the hierarchical structure that surrounds the underlying data. A Document Type Definition(DTD) allows the user to specify the structural template of the elements and attributes of an XML document. While XML Schema[63] is another emerging schema specification standard, this thesis focuses on DTD schema specification. A DTD for the first XML example given would look like the following:

```
<!ELEMENT Name (First, Middle?, Last)>
<!ELEMENT First (#PCDATA)>
<!ELEMENT Middle (#PCDATA)>
<!ELEMENT Last (#PCDATA)>
```

DTDs use a regular expression style language to indicate which elements are required, which are optional, and which can occur in a list. Furthermore, a DTD specifies the order in which elements must occur and may also indicate that an element may contain text or *parsed character data* by using the keyword PCDATA.

The latter example would have a DTD similar to the following:

```
<!ELEMENT Name EMPTY>
<!ATTLIST Name  First CDATA #REQUIRED
                Middle CDATA #IMPLIED
                Last CDATA #REQUIRED>
```

In this case, the attributes are of type CDATA. The *First* and *Last* attributes are required while the *Middle* attribute is optional.

XML has two parsing models. The first is the Simple API for XML(SAX)[44]. SAX is an event-driven model that parses a stream and notifies the application when a parsing event occurs. SAX parsing avoids building an entire XML structure in memory and is useful when parsing large documents. The alternate model is the Document Object Model(DOM)[11]. DOM parses the entire XML document and builds a tree structure in memory. Moreover, DOM provides an API for navigation within the parsed XML tree. This work employs the DOM model of XML tree navigation.

XML can be used in a slew of different application scenarios. XML not only represents the future of the World Wide Web[7], it is a far reaching solution that can be used to represent any data kept in electronic form. Electronic books[34], electronic commerce[4, 59], and distance learning[14] all promise to make use of XML.

1.3 The Focus of this Work

This thesis explores the concept of generating a speech-based method of human-computer interaction. To accomplish this task, we must overcome the following barriers:

1. Audio is serial.
2. It is difficult to represent graphical UIs using speech.
3. A speech-only UI may not suffice in many situations.
4. It is cumbersome to require manual creation of interfaces for different types of data.

We explore a solution that addresses these challenges by:

1. Employing XML as a data representation.
2. Exploiting the semi-structured nature of XML to enable scanning of information.

3. Taking advantage of the inherent separation of content from presentation of that content.

4. Utilizing the XML schema definition to provide an automatic means of creating unique interfaces for different data.

We have developed 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 itself, but may navigate between XML documents 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 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 for the given schema.

The focus of our work has been to develop an auditory-based component for our system. We call this component Audio XmL(AXL). AXL aims to provide a customizable user interface that uses speech as both its input and output mode. Moreover, by using the semi-structured nature of XML, we allow our users to navigate the general structure of the document at hand and actively *choose* which piece of information should be read by the rendering system. Additionally, by analyzing the XML schema of a potential document before hand, we extract as much information as we can to make the interface as intuitive as possible. Finally, we provide a customization language that allows the user to specify the type of interaction he/she wishes to have with the document. This facility allows a user to emulate a screen reading scenario if desired.

Chapter two discusses work related to the field of auditory human-computer interfaces and XML. Chapter three presents the design of the system we have developed while chapters four and five discuss the implementation details. Chapter six explores the details of a possible web application and chapter seven looks at observations we have made based upon user experience. Chapter eight examines future work and chapter nine concludes with a look at our contributions to the field of aural human-computer interaction.

Chapter 2

Related Work

As computers become ubiquitous and as they become integral in our everyday lives, there is a demand for the research community to explore different modes of human-computer interaction. Some research has begun to address the issue of auditory human-computer interaction, however much of what has been done examines specific applications, or auditory representations of more static types of data. This work seeks to learn from the more static approaches that exist to build a dynamic solution that can be used in a variety of different application situations.

In this chapter, we first evaluate the work that has been done on using audio in the human-computer interface in general. We then examine efforts to bring audio specifically to HTML. Next we survey current trends toward uniting speech and XML and conclude with a look at current work related to the application of voice web browsing by phone.

2.1 Audio in the Human-Computer Interface

Aster[38] proposes 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[23], 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.

DAHNI[27] addresses many of the same problems that arise in designing aural interfaces by developing a system that focuses largely on navigation *between* documents in a hypermedia system. It provides a keypad based interface to allow users to navigate across links. While a lot of the motivation behind our work and the DAHNI project is similar, there are two fundamental differences. First, DAHNI provides one *static* interface that may not be appropriate for all data. Moreover, DAHNI emphasizes navigation between documents. Our system seeks to develop a dynamic means of aural HCI that is useful for navigation between documents as well as navigation around one large document.

The hyperspeech system developed in [3] looks at presenting "speech as data". While it focuses on issues of aural navigation of information, the goal of the project is to provide navigation of pre-recorded speech. However, our goal is to present textual data in an aural form. Therefore, while [3] focuses mainly on structuring recorded audio, we assume a structure exists and investigate how to navigate that structure.

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. [8] 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. [5] is a similar system that looks at how to design sounds to represent the types of activities performed in your application. [36] 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. [60] examines what it means to represent data with sound. Similar work is detailed in [25]. 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 [17] and [18] 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. [22, 64, 65, 33, 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 [48, 55, 20, 50, 56]. 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[55] 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, AXL 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.

Another area of research is the use of XML in multimedia environments. Languages such as SMIL[47] 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.

2.4 Voice Browsing by Phone

We focus on voice web browsing as the primary application for our system. [52] 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 [37, 53] 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[57] looks at the network protocol itself while WML[58] 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 chapter. We wanted to design a general system to provide an interface not just for HTML, but for any type of XML document. 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.

Chapter 3

A Method for Automatic Generation of an XML-Based Browsing Tool

A primary goal in the design of our system was to overcome the barriers associated with current audio-based human-computer interaction mechanisms. Current tools such as screen readers present a flat view of data with limited facilities for interaction. We propose that by representing electronic data in a semi-structured way, we can exploit the inherent structure to provide customizable, intuitive interfaces that allow navigation, rendering, and modification of the document itself. We have chosen XML as the underlying data representation. XML not only gives us the benefit of a structural data representation, it also provides a schema definition mechanism. By exploiting the schema definition, we can *automatically* create a customized interface for each family of documents conforming to a given schema. The functionality of the interface should include linking, navigation, creation, and modification of XML documents. The MakerFactory architecture supports this goal. It is interactive, customizable, and useful not only for the XML reader, but for the XML writer as well.

One important feature of XML is the ability to link together XML documents. This is imperative for a web browsing situation and is quite useful in other scenarios as well. We have designed our system such that it can support linking. Our design considers XLink[61] as the link traversal mechanism. XLink[61] is currently being developed as a standard for integrating linking mechanisms into XML[61, 62] 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(see figure 3.1). 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 refine 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

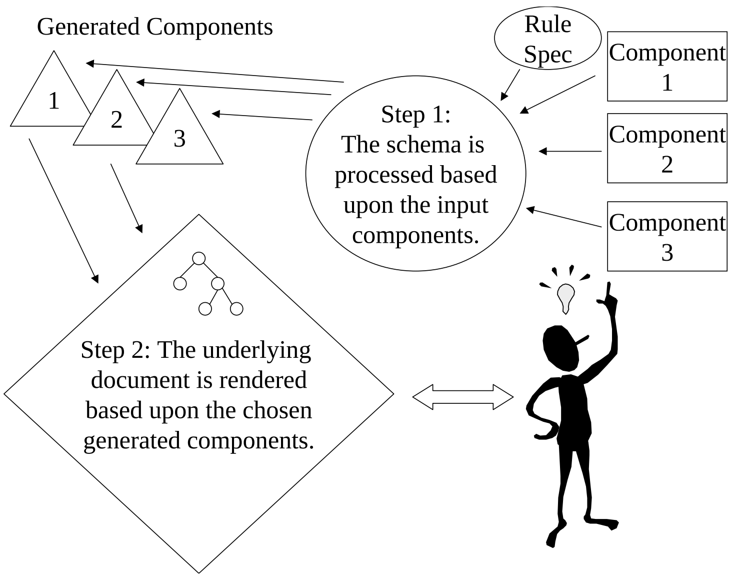

Figure 3.1: An overview of the system design.

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 3.2 shows the general architecture of the code-generation system. A given schema is analyzed and the results of the analysis are passed into a series of user-

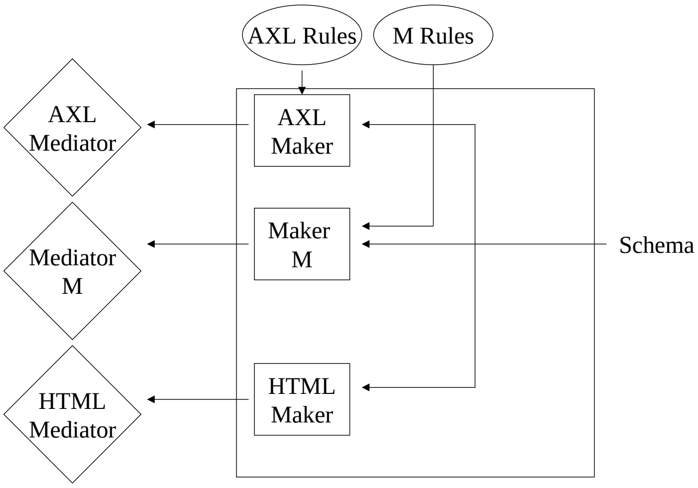

Figure 3.2: The MakerFactory code-generation phase.

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 methods) 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 spoken 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 each 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 textually displaying both HEADLINE and BODY.

3.1.2 Rendering

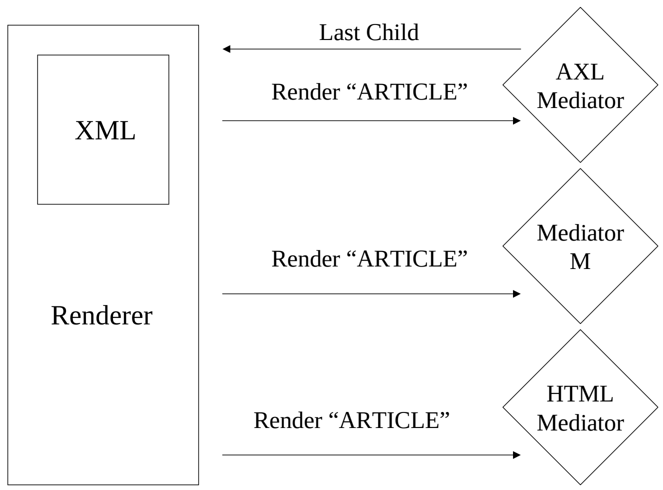

Figure 3.3: The MakerFactory Renderer.

The architecture of the Renderer is shown in figure 3.3. 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.

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. For example, moving to the grandparent of a node would require two requests to move the cursor to the current node's parent. To accommodate this situation, the Renderer defines 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, speech-based interface for 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. For example, a user that reads an electronic newspaper in the same order everyday might wish to simply configure the system to read the newspaper in that order without requiring user interaction. 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

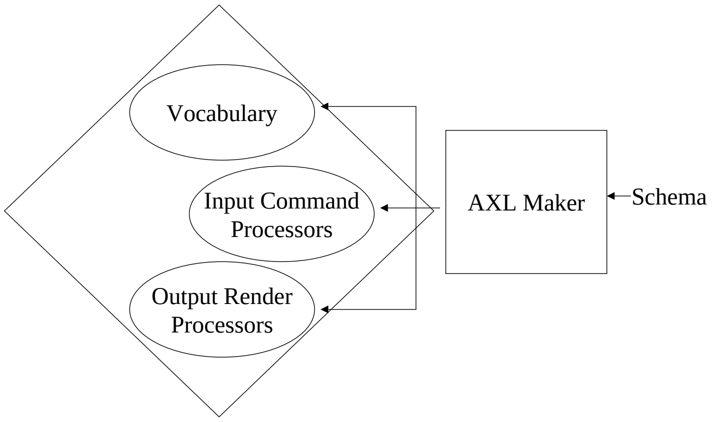

Figure 3.4: The AXL code-generation phase.

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 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 3.4. The resulting set of classes compose the AXL Mediator.

3.2.2 Rendering

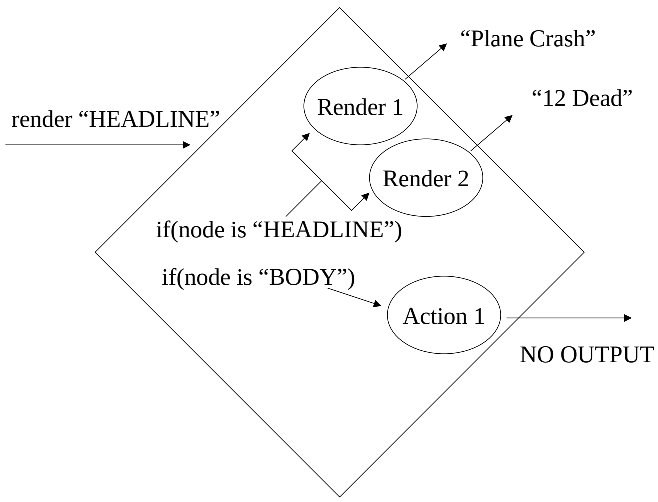

Figure 3.5: The AXL Rendering phase.

The AXL Mediator operates as shown in figure 3.5. 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 can include everything from a simple rendering of the node's tag name to performing the action specified in a pre-compiled Java class. Perhaps a HEADLINE node has two parts, TITLE and 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.

3.3 Link Traversal Support

Linking is an important feature of XML. Not only is it important for a web scenario, linking is important in a variety of XML applications. For example, an electronic book might provide links to annotations of the material.

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 may occur within the XLink namespace. Currently, we are looking only at XLink *simple* links.

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 *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

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 3.1: The XLink Specification Attributes.

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 to give the user more information about the link.

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, or may simply be the current tree with an additional subtree.

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.

Our design supports a concept of data-specific navigation and editing tools. The following two chapters describe our implementation of the framework, and of an aural component to support speech-based navigation and modification of XML data.

Chapter 4

A System for Navigating and Editing XML Data

The MakerFactory architecture is designed to provide an infrastructure for automatically generating interfaces for navigating and modifying XML data. This chapter examines the implementation decisions of the MakerFactory. Many of the implementation details of the MakerFactory are more clearly illustrated by using the speech component, AXL, as an example.

The entire system has been implemented in Java using the IBM Speech for Java[48] implementation of the Java Speech API[20]. 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 general concept of the MakerFactory is to traverse a schema and notify each Maker of the content model of each node in the schema. Based upon the schema and a given rule specification file, each Maker generates an interface for the user and the end result is a completely customized, multi-modal user interface to any XML document conforming to the given schema.

4.1.1 MakerFactory Configuration Schema

The MakerFactory is invoked with a configuration file. The following is the DTD schema that defines the MakerFactory configuration file:

```
<!ELEMENT MakerConfig (Maker*) >
<!ATTLIST MakerConfig
          SchemaProcessor CDATA #REQUIRED
          SchemaLocation CDATA #REQUIRED>

<!ELEMENT Maker EMPTY>
<!ATTLIST Maker
          Classname CDATA #REQUIRED
          Rulefile CDATA #IMPLIED>
```

The MakerConfig element has two attributes. The first attribute defines where the program should find a SchemaProcessor class. The value of this attribute should be a string that indicates the class name of a class that implements the SchemaProcessor interface. The current implementation provides a DTDSchemaProcessor class that supports the DTD schema definition. However, there are multiple ways to define a schema for an XML document. XML Schema[63] is another example. Therefore, to support other methods, any other schema processor conforming to the SchemaProcessor interface may be used. The second attribute of the MakerConfig element is the SchemaLocation. The value of this attribute should be a string that indicates where the actual schema definition can be found e.g. the file location of the DTD. This string should be a valid URL.

The MakerConfig may have a list of Maker elements as its child nodes. Each Maker element indicates a Maker to be used in interface generation. The first attribute of the Maker element is the Classname. The focus of this work is the SpeechMaker or AXL. Other examples might include an HTML Maker that transforms an XML document into an HTML document or a Picture Maker that creates a pictorial view of the XML data by querying an image repository. The second attribute is the name of a Rulefile. The MakerFactory defines a general schema for the set of rules, however each Maker may refine the MakerFactory rule schema to allow a user to write highly customized rules for rendering the elements in the manner defined by the Maker.

4.1.2 The SchemaProcessor Interface

The SchemaProcessor is defined by the following interface:

```
public interface SchemaProcessor {
    void init(String loc) throws InvalidSchemaException;

    boolean hasMoreDecls();

    SchemaNode getNextDecl();
}
```

The current implementation supports a DTD processor, however any class implementing the interface may be used in place of the DTDSchemaProcessor. The processor acts as an enumerator. It should process the given schema and generate a generic SchemaNode representation of each element declaration. A SchemaNode is the MakerFactory defined structure to hold all of the relevant information given in the schema definition of an element. The MakerFactory will iterate once through the schema and process each node.

Any implementation must support three methods. The first is the *init* method that takes as a parameter the location of the actual schema file. If the schema is found to be invalid, an *InvalidSchemaException* should be thrown. The schema defined in the file should be in the format that the SchemaProcessor is expecting. For example, the DTDSchemaProcessor should be given the location of a ".dtd" file.

The second method that must be supported by a SchemaProcessor is the *hasMoreDecls* method. This method is called to determine if all of the nodes in the schema have been processed. If the processor has iterated through all of the nodes, this method should return false, otherwise it should return true.

The final method that a SchemaProcessor must define is the *getNextDecl* method. This method should return the next unprocessed node from the schema. Each node should be processed once. The first step in processing a node of the schema definition is to create a generic SchemaNode object. This MakerFactory defined object contains all of the relevant information from a given node. By analyzing a SchemaNode, a Maker should be able to derive all necessary information in order to build the interface. This gives rise to the second step in processing a node. Each Maker should extract all of the information it needs from the SchemaNode and use it to create the appropriate components of the interface.

4.1.3 The Maker Interface

Each Maker must conform to the following Maker interface:

```
public interface Maker {
```

```
    public void processNode(SchemaNode snode);

    public void addRulefile(RuleFileProcessor rfp);

    public void export();
}
```

A Maker is notified of each node in the schema and should ultimately produce a set of one or more Java classes that will allow navigation of a document conforming to the schema specified.

A Maker must support three methods. The first is the *processNode* method. This method should take a SchemaNode as a parameter and do the processing necessary to create a Mediator. The processing may include using the SchemaNode API to extract information about the XML instance defined in the schema and generating the appropriate Java code.

The second method the Maker must support is the *addRulefile* method. If the configuration document specifies a rule file, the rules defined will be given to the Maker. The Maker should then use the defined rules when building the customized interface. At the very least, a Maker should be able to handle rules conforming to the rule schema defined for the MakerFactory in general. If a given Maker wishes to extend the general rules, it must also implement a way to analyze those extensions.

The final method that must be supported by a Maker is the *export* method. This method will be called when the entire schema has been traversed. The Maker should do any cleanup and make the results of the schema traversal persistent. Generally, this means that the Maker will create one or more Java files containing code that implements the Mediator interface.

4.1.4 The RenderRule Language

Each Maker in the factory can be given an XML document containing user-specified rendering rules. The default Maker generated rendering might not be sufficient for all user's needs. Therefore, the MakerFactory defines a RenderRule schema that may be extended by any given Maker. Any Maker implementation should provide the necessary code to process such a rule definition. At the most basic level, a RenderRule document must conform to the following DTD:

```
<!ELEMENT RenderRules (ElementRule|AttributeRule)+>
<!ELEMENT ElementRule (ElementRendering)+>
<!ATTLIST ElementRule
          elementid 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 RenderRules element should contain a list of ElementRule and/or AttributeRule children. The ElementRule element is used to specify a rendering for a given element in the input XML document to be rendered and the AttributeRule element is used to specify a rendering for an attribute in the given document. We explain each type of rule in the following paragraph.

The ElementRule should contain a list of ElementRendering elements. Each ElementRendering with a condition that evaluates to true will be invoked. Any ElementRendering child with no condition will be invoked by default. In addition, the ElementRule should have an elementid attribute. This attribute should specify the tag name of the elements that will match this rendering rule.

An ElementRendering element can contain one attribute *cond*. If specified, this attribute indicates the name of a class that implements the UnaryPredicate interface. The UnaryPredicate should take one argument, the node in question, and produce a boolean value. When the parent ElementRule is invoked for an element, the class indicated by the *cond* attribute is instantiated and passed the node currently being rendered as an argument. If the UnaryPredicate returns true, the rendering defined by the children of the ElementRendering element will be used to render the node.

The children of the ElementRendering element define a single rendering. The rendering is defined by a list of Render or Action children. The children are invoked in the order they are defined. A Render child will produce one view of an element and an Action child will invoke a user-defined Java code block. Any Action class must implement the following interface:

```
public interface Action {
```

```
  public void notify(Node node);
}
```

Like the Java event model[19], 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 it its subtree, the Renderer should be notified of the change following the completion of the *notify* method. This allows a user to render an element, change the content through the use of an action block, and then render it again.

The Render element has two attributes, *depth* and *cond*. The *cond* attribute acts the same as the ElementRendering *cond* attribute. Before a node is rendered it must meet the condition in the UnaryPredicate class. The *depth* attribute indicates the depth to which the node should be rendered. The valid values for this attribute are:

```
NONE
NODE_ONLY
CHILDREN_ONLY
ATTRS_ONLY
NODE_ATTRS
NODE_CHILDREN
CHILDREN_ATTRS
CHILDREN_NODE
ATTRS_CHILDREN
ATTRS_NODE
NODE_ATTRS_CHILDREN
NODE_CHILDREN_ATTRS
CHILDREN_ATTRS_NODE
CHILDREN_NODE_ATTRS
ATTRS_NODE_CHILDREN
ATTRS_CHILDREN_NODE
```

The node, its children, or its attributes may be rendered. In addition, any combination of the three may be specified and in any order. For example, NODE_CHILDREN means to render the node and then its children whereas ATTRS_CHILDREN_NODE means to render the attributes of the node, the children of the node, and then the node itself. Finally, NONE is the option that indicates that the node should have no rendering at all.

The Action element contains two attributes. The first is the *classname* attribute. This should contain the name of a class which implements the Action interface. When invoked, the **notify** method of this class will be passed the node currently being visited. The second attribute is the *ismodifier* attribute. This attribute should be TRUE if the Java action code modifies the node it is passed and FALSE otherwise. If this attribute is TRUE, the Maker that generates the code should generate the appropriate code to replace the global XML in the Renderer itself with the modified local copy. In the future, the rule language will be extended to support Java code embedded within the XML rules.

The ElementRule and AttributeRule elements are very similar. The only difference between the two is that the valid values for the *depth* attribute of an AttributeRendering node are NODE, NAME, VALUE, NAME_VALUE, and VALUE_NAME. This allows the user to specify whether just the name, just the value, or both properties of an attribute are rendered.

It is up to the programmer to extend the schema for a given Maker. See the AXL discussion in the next chapter for an example extension to these base rules.

4.2 The MakerFactory Renderer

The MakerFactory also provides a Renderer that may be used to navigate and modify an XML document. The Renderer interacts with each Mediator in two ways. First, the Renderer receives directives from the Mediators that indicate that the current cursor should be moved or that the tree should be modified. Second, the Renderer issues directives to the Mediators to update each Mediator's current view of the tree. Like the MakerFactory, the Renderer also defines a configuration file schema that specifies which Mediators to use. The Renderer can interface with any class that implements the Mediator abstract class.

The Renderer reads in an XML document and instantiates a series of Mediators that will listen for changes in the traversal of the document. Each Mediator interfaces with both the Renderer and the user. Based upon the user input, the Mediators notify the Renderer to change the current view of the document or modify its contents. When one Mediator updates the view, all Mediators are notified by the Renderer.

4.2.1 Renderer Configuration Schema

The following is the DTD schema that defines the Renderer configuration file:

```
<!ELEMENT RendererConfig (Mediator*) >
<!ATTLIST RendererConfig
          XMLFile CDATA #REQUIRED >

<!ELEMENT Mediator EMPTY>
<!ATTLIST Mediator
          Classname CDATA #REQUIRED >
```

The RendererConfig element has one attribute. The XMLFile attribute defines the name of the XML file that is to be traversed. This string should be a valid URL of a file that contains a valid XML document. The XML document should conform to the schema used when creating the Mediators defined.

A RendererConfig element contains a list of Mediator elements. It is likely that each Mediator will be an automatically generated MakerFactory Mediator, however, any class extending the Mediator abstract class can be specified. Each Mediator element specifies a Classname attribute. The Classname should be a string indicating the name of the Mediator class.

4.2.2 The Renderer Class

The Renderer is a fixed class that exports a series of methods that may be called by a Mediator class. The methods are generally wrappers around the org.w3c.dom.Node interface. The methods exported support navigation through and modification of an XML document. The Renderer implements the concept of a cursor. There are conceptually two cursors. One points to the current element, and the other points to the current attribute of the current element. When the element cursor moves, the attribute cursor will point to the first attribute of the new element. The methods exported by the Renderer can move the cursors, or change the data to which the cursors point. The methods supported by the Render class are outlined in table 4.1.

Because most of the commands are iterative, that is to say that attribute(attributename) and element(elementname, indexOfElement) are the only methods that allow a user to randomly access a node, a Mediator may want to move a cursor multiple hops before settling on the node that should be rendered. For example, to access the grandparent of the current node, two calls must be made to the *parent* method. To

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 4.1: Methods Supported by the MakerFactory Renderer.

accommodate this case, the Renderer implements the concept of a lock. Before the Mediator can call any of the methods available to alter the cursor state, the lock must be acquired. Once the lock is acquired, the Mediator may move the cursor as many times as it likes. Only when the lock is released are the remaining Mediators notified of the new state of the tree. If the lock is released and the cursor has moved, the node that the cursor now points to is queued for update. Each Mediator is notified of the current state of the cursor.

The locking mechanism is also useful for write interaction. Most of the discussion of this thesis focuses around navigation within an XML document. However, modification is also a desired quality of such a system. In order to allow multiple Mediators to participate and possibly modify the XML tree itself, the Renderer allows a Mediator to lock the tree and replace the current node with a new node specified by the Mediator.

The Renderer interfaces with any Mediator class. In most cases, the Mediators will be automatically generated by the MakerFactory. However, it is possible that a Mediator is physically implemented by a developer using the Renderer. All Mediators must conform to the Mediator abstract class definition.

4.2.3 The Mediator Class

The purpose of a Mediator is to provide a mode-specific interpretation of the basic DOM methods supported by the Renderer. For example, the AXL Mediator provides a means to navigate the XML tree using speech as the input/output mode while a Picture Mediator might provide the facility to drag and drop a pictorial representation of the data.

Any Mediator must implement the *render* method. This method should render the given node in a manner appropriate for the Mediator. The parameter to the method is the node to be rendered. This node is a deep cloned copy of the actual node and all of the subtrees of the node.

Mediators must also implement a *cleanup* method. This method should deallocate all resources appropriately and stop listening for user input.

The *init* method of the Mediator is provided. This method sets the parent Renderer object. This is necessary so that the Mediator can make calls to the parent to traverse the XML document.

The focus of our implementation was an auditory component, although we have implemented a series of classes that extend the Mediator class and provide visual view of the data. The following chapter discusses our implementation of AXL within this framework.

Chapter 5

An Auditory Component for XML Navigation and Modification

The main focus of this work has been to provide the facility to *automatically* generate audio-based components for navigating and modifying XML data. This chapter evaluates the implementation choices of AXL. At the lowest level, AXL provides an auditory version of the Document Object Model. Beyond that, it provides a customized interface through schema analysis and user-customization.

5.1 Speech DOM

At the most basic level, AXL provides a speech version of the Document Object Model[11]. DOM defines a series of methods that allow access to a parsed XML tree. Our Speech DOM provides a set of spoken commands that perform the same function. 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 4.1). 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

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 5.1: Command Set Used to Direct the AXL Engine.

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. Anything spoken before 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".

Table 5.1 illustrates what the user may say 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.

The AXL Speech DOM 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

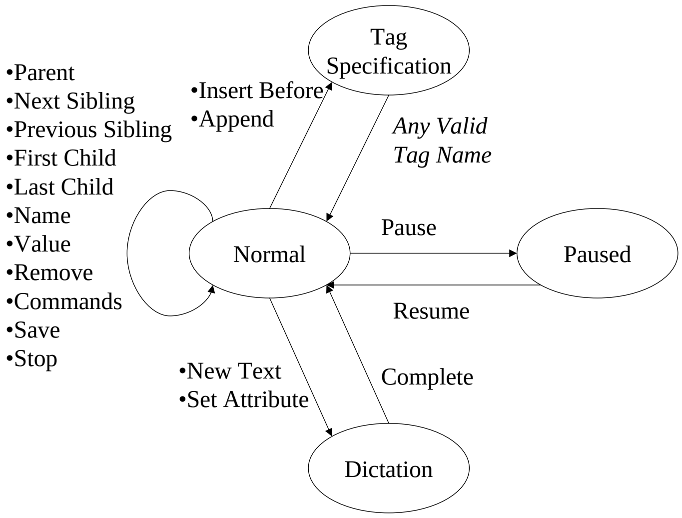

Figure 5.1: The AXL engine state transition diagram.

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 the user finishes dictating, *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.

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 document. A schema-specific interface not only allows more intuitive navigation, it provides the means to customize the output rendering of each type of node. Ultimately, given an XML document conforming to one schema, we would to interact with it in a way specific to that schema.

5.2 A Schema-Driven Interface

Primarily, the DTD tells us what kind of element 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 look at the general 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. 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 more high-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 a user's desired usage of the system. In that case, user input is required.

5.3 The ABL Rule Language

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.

5.3.1 ABL Implementation of the MakerFactory Rules

The general MakerFactory architecture provides a high-level rule language that we can use to specify general rules. However, the meaning of those rules can be interpreted differently by different types of Makers. This section examines the ABL interpretation of the MakerFactory rules.

The *cond*, *elementid*, and *attributeid* attributes are straightforward and implemented as they are described in section 4.1.4 of this thesis. Both *elementid* and *attributeid* specify the tag name of the given node that should invoke the defined rule. For example, if "Person" occurs as the value of the *elementid* attribute, then the rendering defined in the rule will be invoked when a Person element is encountered. When the *cond* attribute occurs as a child of an ElementRendering element, the given ElementRendering will only be invoked if the condition is met. If the *cond* attribute is a child of an AttributeRendering element, the same logic applies. When *cond* occurs as a child of a Render element, that specific rendering will only be invoked when the condition is met. Suppose that the content model of an A node were the following:

```
<!ELEMENT A ((B|C)*)>
```

Suppose that the logic behind the rendering were as follows:

```
if A has less than 5 children
  render all B children then render all C children
else
  render all C children then render all B children
```

To implement this logic, there should be two ElementRendering nodes. The first should be invoked if the condition $numChildren(A) < 5$ is met, and the second should be invoked in all other cases. Furthermore, the first rendering element should have two Render children. The first should indicate that CHILDREN_ONLY should be rendered upon condition $tagName = B$ and the second should indicate that CHILDREN_ONLY should be rendered upon condition $tagName = C$. The second ElementRendering node would also have two Render children. The first of its Render children would indicate that CHILDREN_ONLY should be rendered upon condition $tagName = C$ and the second would indicate CHILDREN_ONLY upon condition $tagName = B$.

The ABL implementation of the *depth* attribute also warrants explanation. In the speech interface, all rendering is done recursively. The default implementation specifies that only a node is rendered. However, the user may wish to change that implementation. It may make sense to render an entire subtree rather than just the node at the root. To illustrate, we use the following XML fragment:

```
<Name>
  <First> John </First>
  <Last> Doe </Last>
</Name>
```

Arriving at the Name node does not give the user much information. In order to be able to identify whether or not this is the entry that the user is looking for, the user needs more information. Furthermore, since the subtree itself is very small, the user may actually want to specify that when the cursor arrives at a Name element, the entire subtree should be rendered. This is the purpose of the *depth* attribute. In the ElementRendering case, the user can specify that the node and its children should both be rendered by specifying NODE_CHILDREN as the value for the attribute. In this case, a depth of NODE_CHILDREN would cause the system to render "Name, First John, Last Doe" when it arrived at the Name node. The *depth* attribute when used in the AttributeRendering context is very similar. If the user wants only the name of the attribute to be rendered, or perhaps just the value, then the *depth* attribute should change accordingly.

Finally, if the speech rules define an Action element, the *classname* attribute indicates that a class of the given type will be instantiated. If the *ismodifier* attribute is TRUE, then the AXL Mediator will acquire the Renderer lock before invoking the given class. After the class is invoked, the Mediator will replace the current node with the modified node and unlock the Renderer. This is the expected behavior of any Mediator.

5.3.2 ABL-Specific Rules

ABL also defines its own set of rules. ABL aims to allow the user to customize each Render element. The current implementation supports the following children of the Render element.

```
<!ELEMENT Tag EMPTY>
<!ATTLIST Tag
          rendername CDATA #REQUIRED>
<!ELEMENT Phrase EMPTY>
<!ATTLIST Phrase
          prenode CDATA #IMPLIED
          postnode CDATA #IMPLIED
          prechild CDATA #IMPLIED
          postchild CDATA #IMPLIED
          preattr CDATA #IMPLIED
          postattr CDATA #IMPLIED>
```

The Tag element allows the user to change the rendering from the tag name of the element to some user-specified value. This is useful for many reasons, one in particular being the discrepancy between tag names and actual English. Instead of speaking a tag name such as "STGDIR", which the engine may not even be able to convert to speech, the user could specify that the output should be "stage direction", a more understandable phrase.

The Phrase element allows the user to define different phrases that will be spoken before and after the rendering of each type of node. For example, a *prenode* phrase might be "You have arrived at a ", while a *postnode* phrase might simply be " node". The resulting rendering of a Name node would be "You have arrived at a Name node". Similarly, a *postchild* phrase might be "followed by" such that the rendering of the children might be something like "First followed by Last". Of course, all of these can be left blank. However, any or all of the attributes can be defined such that nodes can be connected together with a more sentence-like flow.

Another desired function is to specify the way in which the user can access a node of a given type. For example, the Tag element allows the user to specify how the tag of a given node should be interpreted in the spoken rendering of the node. It makes sense to have the semantics of this element attached to a given Render element because a given node may be spoken in a variety of different ways depending upon the given condition. However, it is also useful to allow the user to specify the way in which they wish to access a node of the given type. However, this specification does not depend on any given condition of the node. For example, every time the user wishes to hear the "stgdir" node, the user should speak the same command. Therefore, we attach a *Speak* attribute to the ElementRule element of the rules. If the user wishes to access a "stgdir" node by saying "Give me the stage direction", then the value of the *Speak* attribute will be "Give me the stage direction".

The final extension that we have made to the schema is to add a Default element to the content model of the RenderRules element. The Speak and Default modifications yield the following changes to the base schema:

```
<!ELEMENT RenderRules ((ElementRule|AttributeRule)+, Default?)>
<!ELEMENT ElementRule (ElementRendering)+>
<!ATTLIST ElementRule
          elementtid CDATA #REQUIRED
          Speak CDATA #IMPLIED>
<!ELEMENT Default EMPTY>
<!ATTLIST Default
          parent CDATA #IMPLIED
          next CDATA #IMPLIED
          previous CDATA #IMPLIED
          nextAttr CDATA #IMPLIED
          previousAttr CDATA #IMPLIED
          name CDATA #IMPLIED
```

```
value CDATA #IMPLIED
exit CDATA #IMPLIED
pause CDATA #IMPLIED
resume CDATA #IMPLIED
whatcanisay CDATA #IMPLIED
stop CDATA #IMPLIED>
```

Each attribute represents one of the default commands in the current version of AXL. If the user wishes to change one of the default commands, the new command should be given as the value of the attribute. For example, if there were an element "name" in the actual schema of the document, there would clearly be a conflict between the command to access the name element and the command to get the name of the element pointed to by the current cursor. This conflict can be resolved if the user changes the default name command to something else.

```
<Default name=''my identifier''>
```

The preceding example would change the default name command to "my identifier". When the computer hears the user say that phrase, it will reply with the tag name of the current node.

ABL and the implementation of the system have been designed so that able can be developed further as the AXL implementation evolves.

Chapter 6

Voice-Enabling the Web Using AXL

This chapter details AXL's use as a front-end application enabling web browsing by voice. Specifically, we examine AXL as a way to read the morning newspaper by phone. Most current voice web 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[55] 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.

Our proposal is a more generalized view of the phone-web. Push technology[28, 56] is not sufficient for the user who wants to actively browse the entire web. 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, a user can make a request to AXL which contacts the XML server, caches the document, and navigates it as the user directs.

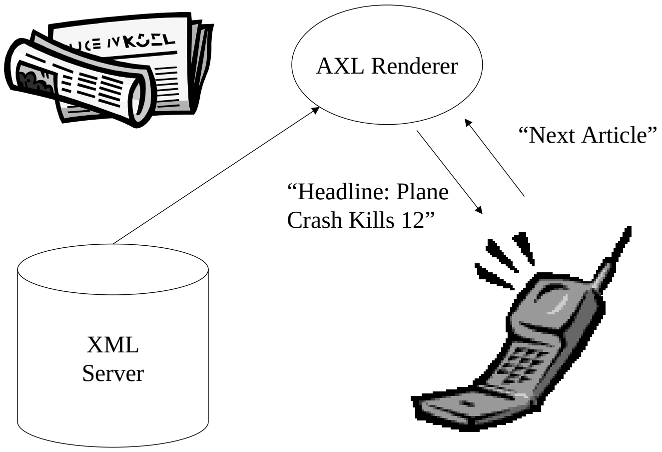

Figure 6.1: AXL allows a user to browse the web by phone.

Figure 6.1 illustrates this example. AXL contacts an XML server and caches the XML document it receives from the server. The user then issues directives to the Renderer that specify navigation of the document. These can include Speech DOM level commands as well as schema-based commands. The Renderer responds by doing a text-to-speech conversion of the data in the document based upon the rendering rules. If the user chooses to traverse a link, the system will contact the server to cache the new document.

6.1 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 newspaper 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[31]:

```
<!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 causes the system to listen 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. As another example, suppose that the crossword section of the newspaper were expressed in XML. The user could actually fill out the crossword puzzle while on a cellular telephone. 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.

Chapter 7

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.

This chapter discusses the observations we have made regarding the design of the system based upon observation of actual users. First, we discuss how the test itself was designed and conducted. We then examine the preliminary observations and conclusions we have made.

7.1 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. The document we designed followed the following schema:

```
<!ELEMENT Travel (Flight-List, Rental-Car-List, Vacation-Package-List)>

<!ELEMENT Flight-List (Flight*)>
<!ELEMENT Flight (Destination, Airline, Price)>
<!ELEMENT Destination (#PCDATA)>
<!ELEMENT Airline (#PCDATA)>
<!ELEMENT Price (#PCDATA)>


<!ELEMENT Rental-Car-List (Rental-Car*)>
<!ELEMENT Rental-Car (Type, Price)>
<!ELEMENT Type (#PCDATA)>

<!ELEMENT Vacation-Package-List (Vacation-Package*)>
<!ELEMENT Vacation-Package (Description, Airfare, Hotel-Cost, Price)>
<!ELEMENT Airfare (#PCDATA)>
<!ELEMENT Hotel-Cost (#PCDATA)>
<!ELEMENT Description (#PCDATA)>
```

Users were first 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 7.1 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 told the user which commands to issue to navigate within the document. The user was then expected to execute each

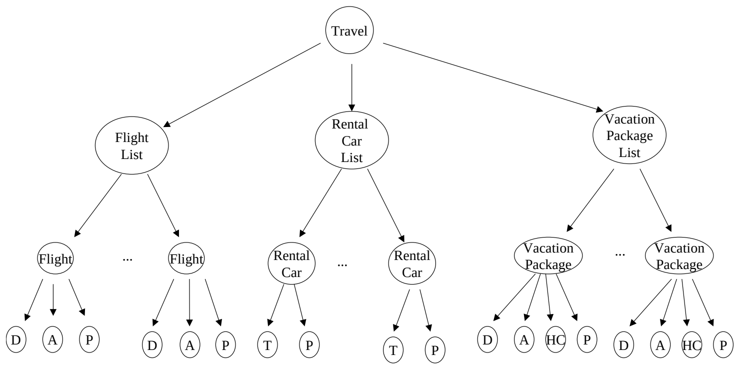

Figure 7.1: A tree representation of a travel document.

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.

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.

7.2 Subject Profiles

We surveyed three preliminary subjects and found that the study itself needed to be modified. The subjects found that some of the instructions needed to be enhanced with further explanation. After slight modification of the study we surveyed five other people, all of whom were able to perform the given tasks. We found that the observations of both sets of subjects were very similar and thus discuss observations made by all eight subjects.

All three preliminary subjects were familiar with XML. Of the subjects, two were male and one female. Moreover, two subjects had non-American accents. Of the five remaining subjects, four were female and one was male. One of the female subjects had a non-American accent and only one was familiar with XML.

7.3 Observations

We made five main observations:

1. The users' frustration with the rendering of some nodes facilitates the need for customization rules.

2. The automatic generation of a command set based upon schema yields commands considered natural by users.

3. The limited vocabulary produced by schema analysis aids the ViaVoice recognition engine.

4. There is a need to develop a better *help* command.

5. There is a need to develop enhanced feedback mechanisms.

7.3.1 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 suitable manner. A more extensive user study will evaluate the customization language by asking the users to write customization rules.

7.3.2 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.

7.3.3 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[54], but tends to require extensive training in order to perform well. However, we were encouraged to discover that with only two sentences worth 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 and hence it is beyond the scope of this thesis.

7.3.4 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 by 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.

7.3.5 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.

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.

Chapter 8

Future Work

The extensible nature of the system we have built makes it suitable for many possible additions. First, we would like to perform more extensive schema analysis to provide an even higher-level query language and summarization facility. Further, we would like to examine the potential of interface generation based upon a user profile. Additionally, we would like to develop the concept of *Speech Spaces* that would allow the use of multiple Renderers simultaneously. The future possibilities for our system not only include extensions, but integration into other applications as well. The final section of this chapter looks at how our system can be used in the areas of accessibility, electronic books, and distance learning.

8.1 Query and Summarization

The ability to navigate and modify a document as it exists is useful, however we would like to integrate the ability to preprocess a document to provide a summarization and query mechanism with which to interact with the document. First, we would like to produce meaningful summaries for each node. If we can look more closely at the content model and then perform a preprocessing step of the document itself at run-time, we can produce summaries that may be more useful to a user. Such a summary may consist of the relevant identification information that defines a given type of node.

Similarly, we may be able to use both the content model and the run-time document to produce a high-level query language. By integrating work already done in the query area[26], we can produce a set of indices and a vocabulary with which we can interact with the document. This will enable us to ask for information from the document at a high-level rather than navigate through the nodes to discover the information we wish to obtain.

8.2 Profile-based Interface Generation

Another area we want to consider is interface generation based upon profile. By building a profile of the user before the interface is generated, the interface can be customized based upon the type of information a user may wish to derive from the document. For example, suppose that a book were represented in XML. A student may wish to navigate the book very actively by stopping to take notes after every subsection. However, a person who is reading for pleasure may want to assume a more passive role by just listening to the contents of the book.

8.3 Speech Spaces

In order to fully implement the concepts defined by XLink[61], 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.

8.4 Application Integration

AXL is not only useful in a phone web scenario. For any electronic data that can be represented in XML, AXL can provide a useful interface for navigating and modifying the data. This chapter discusses three other possible applications. First, we look at how AXL can help to make the web more accessible to people with disabilities. The electronic book scenario is another well suited applications. Finally, AXL can be integrated into a more general multimedia scenario as well.

8.4.1 Accessibility

As more information is represented electronically, it affords the perfect opportunity to make that information accessible for print disabled users. Current screen reading technology is not only limiting in its ability to represent non-textual information, by many standards it does not adequately represent text alone[18]. By nature, a listener must assume a passive role where a sighted viewer takes a more active approach[38]. A screen reader that trivially reads a flat piece of text does not give the user the abilities afforded a sighted viewer. A listener cannot scan the information, move back or forward, or imply any underlying structure based on font size or page layout(figure 8.1). We propose that as XML gains popularity as a data representation mechanism, we can use AXL to generate usable speech-based interfaces that allow print disabled users to fully navigate electronic documents.

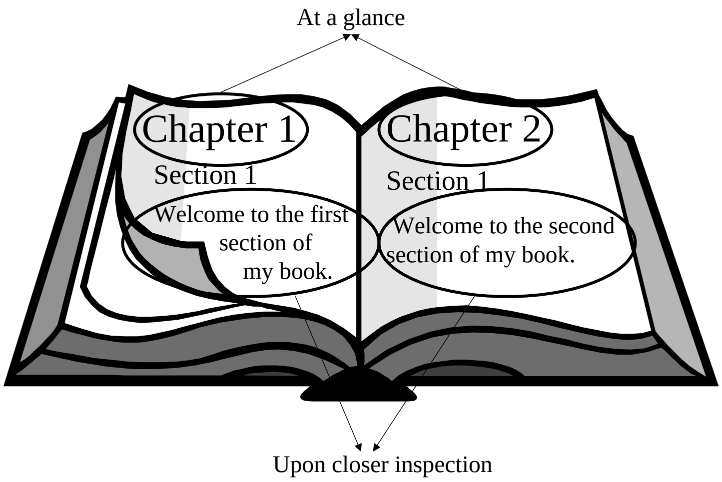

Figure 8.1: A reader can glance at a printed page and instantly understand the structure.

Aurora[16] seeks to make the web more accessible by classifying functions performed on the web into a series of different domains. One example of a domain would be the search domain. There are many different search engines, Yahoo, Excite, Infoseek, etc. Each serves the same function, but through a slightly different interface. Aurora generalizes the function of searching by defining a single schema that could be used to represent the function of all search engines. By putting all search engines into a common format, Aurora eliminates many of the problems encountered by print disabled users. AXL can take advantage of the schemas published by Aurora and create speech-based interfaces that are more useful than the current static technologies.

8.4.2 Electronic Books

Electronic books are quickly gaining popularity. It is not unrealistic to think that the eBook paradigm will soon be as popular as reading a traditional book. [34] details the specification of eBook as based upon XML. The specification outlines the DTD that an eBook must conform to. Given that eBooks are represented in XML, users of eBooks can take advantage of the system that we have designed.

The most obvious use is for print disabled users to use AXL to *listen* to their electronic book[10]. Unlike the traditional "book on tape" scenario, an AXL interface would give the user the opportunity to navigate within the book. Movement would not be restricted to the serial nature of fast forward or rewind. The user would virtually have random access to the contents of the book(figure 8.2). Additionally, different types of books require different reading methods. A user may wish to read a text book very differently than a novel, one requiring more interaction than the other. AXL would allow a user to stop the reading at will and resume when the user was ready.

By taking advantage of the nature of the entire MakerFactory system, any user may prefer to use the MakerFactory Renderer to access the contents of their eBook. The MakerFactory allows a user to generate a customized, multi-modal interface to the XML document. Therefore, a user could generate an interface that would allow them to switch between listening to the document and writing annotations on the material. This could be useful for anyone who wants to do more than simply read a book.

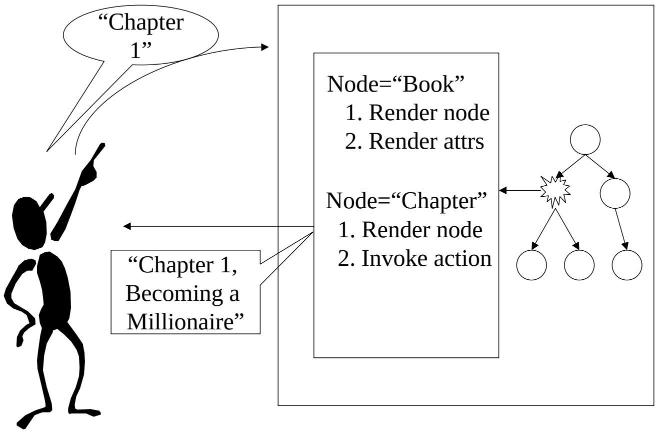

Figure 8.2: AXL renders an eBook

8.4.3 Distance Learning

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[14]. 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.

The system can also be used in other types of multimedia environments. Imagine a professor's lecture specified in XML. As the speaker or lecturer presents the content of the presentation, their spoken commands move the current cursor from one node to another. The rendering of a node may be defined by many different Mediators, where one speaks a phrase, another shows a slide, and yet another opens a relevant web page. The rendering may be shown on a global screen seen by all participants, or perhaps on individual laptops as one might imagine in a digital classroom scenario.

As XML emerges as a solution in more and more situations, the applications of our system emerge as well. Its extensibility as an infrastructure make it a powerful tool to be embedded in many different applications.

Chapter 9

Conclusions

The aim of our work has been to expand the concept of user interfaces beyond the current WIMP model. Inspired by the restrictions facing people without the ability to interact with their computer using sight, this work has addressed the question of creating a method for allowing users to speak to their computers and receive spoken responses. The creation of aural human-computer interfaces is not without challenges. Aural interfaces must provide users with an intuitive way to interact with computers while still providing them with as much control over the data navigation and manipulation as a sighted user would have.

We chose to explore the possibility of providing such a method using XML as the underlying data representation and exploiting the inherent schema definition mechanisms it provides. The framework that we have developed not only addresses the question of creating aural interfaces, it proposes a generalized result. By employing the technique we developed for aural interface generation, we can compose data-specific, user-customized means of access to electronic data.

The contributions of this work include:

- A means for aurally browsing data.
- A framework for developing multi-modal means of HCI.
- A method that enables automatic generation of interfaces that are unique for given data.

The implications of our technique suggest that computing power can be exploited by any user regardless of restrictions imposed by disability or current circumstance. As computers become a part of everyday life for people around the globe, it is imperative that no one be excluded from the information and services they provide. Allowing users to access information and services through the means most appropriate to their own circumstance in an intuitive, instinctive manner will enable anyone, anywhere to be a part of the technological revolution.

Bibliography

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

[2] B. Arons. Authoring and Transcription Tools for Speech-Based Hypermedia Systems. In *Proceedings of 1991 Conference, American Voice I/O Society*, Sep. 1991, pp. 15-20.

[3] B. Arons. Hyperspeech: Navigating in Speech-Only Hypermedia. In *Proceedings of Third Annula ACM Conference on Hypertext*, Dec. 1991, pp. 133-146, San Antonio, TX.

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

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

[6] E. Bergman and E. Johnson. *Towards Accessible Human-Computer Interaction* Advances in Human-Computer Interaction, Vol 5., New Jersey, 1995.

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

[8] 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.

[9] J. Davis. A Voice Interface to a Direction Giving Program. MIT Media Laboratory Speech Group memo 2, April 1998.

[10] D. DeVendra. Features and Benefits for the Interface of the Next Generation Digital Talking Book. In *CSUN 98*, Los Angeles, CA, March, 1998.

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

[12] K. Edwards, E. Mynatt, and K. Stockton. Access to Graphical Interfaces for Blind Users. *Interactions* Vol 2, No 1, January, 1995, pp 54–67.

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

[14] 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.

[15] E. Glinert, R. Kline, G. Ormsby, and G.B. Wise . UnWindows: Bringing Mulitmedia Computing to Users with Disabilites. In *Proc. 1994 IISF/ACMJ International Symposium on Computers as Our Better Partners*, Tokyo, March 7–9, pp. 34–42.

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

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

[18] 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.

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

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

[21] D. Karron. Developing an Audio-Enabled Intraoperative Computer-assisted Surgical System. *DARPA Proposal*, Arlington, Virginia, 1997.

[22] 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, .

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

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

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

[26] M. Mani, N. Sundaresan. On the Design and Implementation of an XML Query System. *IBM Technical Report, December, 1999.*

[27] S. Morley, H. Petrie, A.M. O'Neill, and P. McNally. Auditory Navigation in Hyperspace: Design and Evaluation of a Non-Visual Hypermedia System for Blind Users. In *Proc. of ASSETS '98*, Los Angeles, CA, 1998.

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

[29] M. Muller and J. Daniel. Toward a Definition of Voice Documents. In *Proc. COIS 90*, 1990.

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

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

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

[33] 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.

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

[35] H. Petrie, S. Morley, P. McNally, A.M. O'Neill, and D. Majoe. Initial Design and Evaluation of an Interface to Hypermedia Systems for Blind Users. In *Proc. of the eighth ACM conference on Hypertext '97*, 1998, pp 48–56.

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

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

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

[39] S. Rollins and N. Sundaresan. AVoN Calling: AXL for Voice-enabled web Navigation. To appear in *Proceedings of the 9th Annual World Wide Web Conference*, The Netherlands, May 2000.

[40] S. Rollins and N. Sundaresan. An Auditory Interface for Making Electronic Data Accessible to Print Disabled Users. Submitted for publication, December, 1999.

[41] S. Rollins and N. Sundaresan. A Framework for Creating Customized Multi-Modal Interfaces for XML Documents. Submitted for publication, December, 1999.

[42] K. Sall. To E or Not To E: Elements vs. Attributes. http://www.stars.com/Authoring/Languages/XML/Tutorials/DoingIt/eltsVSattrs.html

[43] The SGML/XML Web Page http://www.oasis-open.org/cover/

[44] SAX: The Simple API for XML http://www.megginson.com/SAX/index.html

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

[46] S. Sinks and J. King. Adults with Disabilities: Perceived Barriers that Prevent Internet Access. *CSUN '98*, March 1998.

[47] Synchronized Multimedia Integration Language (SMIL) 1.0 Specification W3C Recommendation 15-June-1998 *http://www.w3.org/TR/REC-smil/*

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

[49] L. Stifelman, B. Arons, C. Schmandt, and E. Hulteen . VoiceNotes: A Speech Interface for a Hand-Held Voice Notetaker. In *Proc. INTERCHI 1993*, The Netherlands, April 1993, pp. 179—186.

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

[51] M. Teel, R. Sokolowski, D. Rosenthal, and M. Belge . Voice-Enabled Structured Medical Reporting. In *Proc. CHI 1998*, Los Angeles, April 1998, pp. 595—602.

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

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

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

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

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

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

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

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

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

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

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

[63] XML Schema Requirements. http://www.w3.org/TR/NOTE-xml-schema-req

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

[65] 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.