2001

Translation of XML applications to VoiceXML applications.

Hong Ying. Dou
University of Windsor

TRANSLATION OF XML APPLICATIONS TO VOICEXML APPLICATIONS

By

Hong Ying Dou

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of the Master of Science at the
University of Windsor

Windsor, Ontario, Canada

0-612-67599-8

ABSTRACT

Translation of XML Applications to VoiceXML Applications

By Hong Ying Dou

XML, the eXtensible Markup Language, is used widely and in many different ways. With the explosive growth of the World Wide Web, XML is becoming a universal accepted format for document publishing and data exchange. Voice access to a wealth of information has been challenging. Besides for visually impaired people, the need for speech is greatly increased by restrictions imposed by circumstance. However, the development of speech applications has been hindered by a lack of easy-to-use standard tools for managing the dialogue between user and computer as well as its expensive development and deployment. VoiceXML, the Voice eXtensible Markup Language, is based on XML. It provides a high-level programming interface to speech and telephony resources for creating distributed voice applications.

Our objective is to investigate means, techniques, and common patterns to build a generalized mapping model for the transformation from XML to VXML. This model is used to simplify the development of speech applications, especially for voice access and input of information in the XML documents. We present GATXV & XVMA, a generic architecture and a mapping algorithm of translating XML to VXML. They are designed to allow non-speech-experts to build and use VoiceXML to input data in a specified XML format or access such XML data via voice. This approach is absorption of the component technologies, the relevant information sources, and the user interface. The generic architecture and algorithm cover comprehensive considerations on speech user interface design and mapping strategies in the translation process.

Keyword: XML, VoiceXML, XML DTD, XML Schema, translating, architecture, speech user interface design, mapping

DEDICATION

This thesis is dedicated to my parents, for their constant loving presence in my life and for modeling tenacity, courage and honor. To my sisters and brother, Xia, Mei and Guang for their love, encouragement and support all along the way of my life and for being there for me always. To the many friends, teachers throughout my life for their love and guidance who have helped me to discover new ways to be. Finally, to my wonderful husband, Junlei, for his caring, understanding and love. They are what make it all worthwhile.

ACKNOWLEDGMENTS

My heartfelt gratitude is extended to my thesis advisor, Dr. Richard A. Frost, for all of his patience, encouragement, constructive criticism, support, long-sighted thinking and willingness to give me much needed guidance through completion of the thesis.

My appreciation also goes to committee members, Dr. Walid S. Saba, Dr. Diane Kao, Dr. Scott Goodwin and Dr. Vassil Dimitrov for having the patience to read drafts and for their attentive, invaluable comment, suggestions and relevant feedback.

I deeply appreciate all the staff and faculty members in computer science for the help they have given me during last two years. Especially, Dr. Indra A. Tjandra, Dr. Jessica Chen, Dr. Christie Ezeife, Dr. Alioune Ngom, Dr. Weidong Zhang, Ms. Margaret Garabon Cookson, Ms. Mary Mardegan and Ms. Gloria Joyce Mensah.

I also would like to thank Yue Su, etc. for the encouragement and helping me to test the system at the final stage of the completion of the thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EXAMPLES

GLOSSARY

Terms Used in the Thesis

XVMA	XML to VXML Mapping Algorithm
XVMA4DIA	XVMA for Data Input Application
XVMA4DAA	XVMA for Data Access Application
GATXV	Generic Architecture of Translating XML to VXML
VXML	An interchangeable term to VoiceXML in this paper

Speech-related Terms and Abbreviations

Active grammar	A speech or DTMF grammar referred to by the currently executing element
ASR	Automatic Speech Recognition
Directed dialog	The application directs the user to make an utterance.
DTMF	Digital Tone Multiple Frequency.
Grammar	A schema for designating the well-formed phrases or sentences drawn from a vocabulary
GUI	Graphic User Interface
IVR	Interactive Voice Response
Mixed initiative	A human-machine interaction in which either the human or the computer can initiate an action
Recognizer	A engine that converts incoming speech to text
SUI	Speech User Interface
Synthesizer	An engine that converts text to speech output (See TTS)
Text-to-Speech	Machine synthesis of human speech from text
TTS	Abbreviation of Text-to-Speech

VUI	Voice User Interface, also referred to Speech User Interface
Wizard of Oz scenario	A VUI design exercise involving two people in which one person plays the role of the machine and the other plays the role of the user

XML-related Terms and Abbreviations

CGI	Common Gateway Interface
DTD	Document Type Definition.
DOM	Document Object Model
JSP	Java Server Pages
HTML	Hypertext Markup Language
MathML	Mathematical Markup Language
MIME	Multi-purpose Internet Mail Extensions
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VoiceXML	Voice extensible Markup Language
W3C	World Wide Web Consortium
WAP	Wireless Application Protocol
WML	Wireless Markup Language
WWW	World Wide Web
XHTML	eXtensible Hypertext Markup Language
XML	Extensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

Chapter 1

INTRODUCTION

This work is an effort on investigating the techniques and common patterns to build a generalized mapping model for the transformation from XML to VXML. We present GATXV and XVMA, a generic architecture and a mapping algorithm of translating XML to VXML. They are designed to allow non-speech-experts to build and use VoiceXML to input data in a specified XML format or access such XML data via voice. This approach is absorption of the component technologies, the relevant information sources, and the user interface. The generic architecture and algorithm cover comprehensive considerations on speech user interface (SUI) design and mapping strategies in the translating process.

The mapping rules are based on the underlying structure and data types embedded in the DTD and XML Schema. This work not only addresses the mapping rules and some issues about speech interfaces but also presents a generalized model for constructing a voice application using VXML. The user-customized specification of access to XML data is discussed as well.

In this Chapter, first we talk about the inspiration behind this work. Then some related works are introduced. Following that, objectives of this work and thesis statement are presented. Finally, we give the organization of the thesis report.

1.1 Motivation

XML, eXtensible Markup Language, is a simple and very flexible text format used for document publishing and data exchange in the World Wide Web (WWW). XML plays the most prominent role on the Internet, and it promises to be the future of WWW [Bosak, 1997]. "As XML is emerging as the data format of the Internet era, there is a substantial increase of the amount of data encoded in XML format" [Bonifati, 2001]. To access and query such XML data is currently an ongoing research topic in the W3C and many other research groups around world [W3C Query] [Deutsch, 1999] [Chamberlin, 2000] [Robie, 1999].

The explosive growth of the Web and the rapid revolution in communications promise to provide ubiquitous access to multimedia communication services. Speech is the most natural ways for people to communicate with each other and to get information; in addition, there is much need for voice applications to access information stored in XML format. However, it has been more challenging than a traditional application is. The difficulties come from two aspects: one, it is hard to build a voice application due to the lack of easy-to-use standard tools, and this leads to the high cost to develop and deploy. Two, there is no such standard on how to access, query and use XML data in voice applications.

To solve the first problem, the VoiceXML Forum [http://www.voicexml.org], founded by Motorola, IBM, AT&T, and Lucent, developed a Voice eXtensible Markup Language – VoiceXML. The goal of VoiceXML (for simplicity, we also call VXML) is "to bring the full power of web development and content delivery to voice-response applications, and to free the authors of such applications from low-level programming and resource management"[VxmlOrg, 2000]. VXML is easy to learn and easy to use. It provides a high-level programming interface to speech and telephony resources for creating distributed voice applications, which facilitates the building and deploying of voice applications easier for both telephone services providers and developers.

With VXML, the subject changed to "is it possible that we can translate the XML data to VXML", the answer is "yes". XML provides transformation mechanisms such as, XSLT [W3C XSLT] to transform one XML to another. However, the target VXML faces numerous issues related to spoken dialogue system [Allen, 1995] [Agarwal, 1998] and speech user interface design [Boyce, 1996] [Frost, 1998] [IBM, 2000]. Moreover, little work has been done on how XML can be mapped to VXML meanwhile to deal with the speech issues, and there is no regulations described on the similar matters. The observations are:

1 XML has become one the fastest evolving topics in information technology, exceptionally in data exchange on the Web;

2 There is much need for speech, especially for voice accessing of large of information on the Web;

3 Speech applications are costly in development and deployment;

4 VoiceXML is easy to use and is designed to build voice/telephony applications quickly.

Following these observations, our research is motivated by the idea that if XML can be automatically mapped to VXML, and this VXML is satisfied with a certain design requirements of speech user interface, we can manage to voice accessing/inputting large of information in XML format on the fly. This work is an attempt on exploring the techniques and common patterns to build a generalized mapping model for the transformation from XML to VXML.

1.2 Related Work

In this section, we first review the work that has been done on spoken dialog applications for the Web. We then evaluate the speech user interface related work. Next, we examine the work that is related to XML.

1.2.1 Spoken dialog Applications for the Web

Plenty of work has been done in this category.

[Frost1, 1999] presents a system called SpeechNet/SpeechWeb, which integrates distributed speech-accessible hyperlinked objects and executable specifications of attribute grammars. It is independent of HTML. If you want to build SpeechWeb for a specific domain, you have to construct it from the very beginning. The detailed comparison between SpeechWeb and VoiceXML is given in Chapter 4.

[Agarwal, 2000] classifies the variety of web applications that can be built using voice browsers into three categories: Web browsing, limited information access, and spoken dialog systems. [Hakkinen, 1997] describes a browser that parses HTML to supplement the audio rendering with structural descriptions for visually impaired people [Lau, 1997] presents a system allowing user querying and navigation of the Internet. [Hemphill, 1997] addresses the issues of voice browsing the Web. Some work, such as [Krell, 1996] and [Zajicek, 1998], in this category focused on how to treat HTML tags in a voice manner. Our goal is to map XML components rather than a particular tag to VXML components.

Another work that needs to be mentioned in this category is [Goose, 2000]. "Vox portal" this paper describes a generic solution for HTML and VoxML [MotorolaVox] conversion and is presented by multi-tier Client/Server architecture. The paper mainly deals with HTML forms, tables and linearization of arbitrary-depth HTML framesets. Nevertheless, detailed core transformation between HTML and VoxML is not published in the paper. It has some similarities with our work in the conversion part. However, for us the source is XML, the target is VoiceXML that is a standard voice markup language. HTML, VoxML, VoiceXML and XML are all markup languages, the relation of which can be described as: XML is used to define the source of the information. In contrast, VoxML, VoiceXML, and HTML can present this information in different formats that in turn to be represented by telephone, voice browser or Web browser, and through which to communicate with the user. XML is the future of the Web; we are dealing with the future trend.

Some work in this category also suggests that it is important to have some degree of dialog management built into the system to handle all kinds of errors, which indicates that we should have a certain level help or error recovery mechanism built in our system to deal with the ambiguous or unrecognized utterances from the user.

1.2.2 Related Work on Speech User Interface

This category concerns how and when to use speech. For example, what a user can say and when it can be said in the input vs. the designing computer prompts, and feedback to users in the output. It also addresses the issues about the dialog flow, timing help, disambiguation and more [Hunt, 2000].

[Frost, 1998] and [Frost, 1995] indicate that: one way to improve the speech-recognition accuracy is to re-engineer the spoken input language so that it is better suited to the recognition process. The paper suggests that the improvement could be made in some cases by modifying the input language, a word or a phrase, replacing with equivalent phrase, etc. This indicates that we need to pay attention to design the grammar file in the mapping process.

[IBM, 2000] introduces the speech user interface guidelines based on industry research and lessons learned in the process and author's experiments of developing VXML and telephony applications. This paper covers many aspects that are important in developing voice applications. For example, it recommends having a *'welcome messages'*, building *'always-active command list'*, which is adopted in our generic architecture.

[Zue, 2000] presents a weather reporting system called Jupiter. It has the ability to hold long conversations, recover from communication errors and handle complex dialogs. However, such a system is special designed to achieve the above capability.

1.2.3 XML-related Applications

[Rollins, 2000] makes one step toward using an aural interface to XML documents. AXL is implemented by Java speech API, and the results are a set of Java classes that provide a speech interface to navigate and traverse XML documents. One limitation of AXL is that it fully depends on the document DTD that is expressionless on the tags' data type, so it could lead to poor understanding on TTS engine output. Furthermore, AXL does not provide user-customized specifications. We propose that the mappings should make use of plenty of data types provided by the XML Schema and figure out how to provide customization rules in the mapping process.

[Badros, 2000] describes an XML conformed representation of Java-source code called JavaML. It claims that JavaML has the ability to directly represent the structure of the Java source program.

[Ide, 2000] instantiated the Corpus Encoding Standard as an XML application called XCES. It provides guidelines for encoding various features in written text, annotation, and alignment information in natural language processing. The paper also indicates that XCES is going to adopt XML specifications such as XPointer, XPath, XSL, and XML Schema.

A large amount of work related to XML has been done by the database community. In addition, W3C Voice browser working group's several voice-related specifications such as speech recognition grammar etc., and speech synthesis markup language are underway. A speech interface framework is also defined by [W3C Voice].

As we can see that a speech interface can be constructed by Java speech API [SUN JSAPI], executable specification of attribute-grammar or multi-tier Client/Server architecture combining with HTTP protocol and enabled technologies such as ASP, JSP, Java Servlet and CGI. Those techniques can be used in our implementation. However, they do not provide an easy way to revise the interface and need much longer iterations between design and development phases. Moreover, those technologies are hard to learn and require domain specific knowledge to develop. We propose to map XML to VXML. In this way, we can take advantage of VXML because first, it is designed and developed to be easy to use; second, if further requirements arise, it is easy to be modified to improve the functionality.

From the related work above, we can also see that XML is rapidly becoming one of the most widely adopted technologies for information exchange and representation in many different areas, and this trend is just getting started and it is going to continue. Our research is motivated by such an XML-embraced environment. The aim is to bring the power of the XML and VoiceXML to build a translating model for voice accessing and inputting information in XML documents.

1.3 Objectives

The ultimate goal of this work is automatically convert any XML document to a corresponding VXML document. Due to the explosive growth of XML family and the various purposes of XML applications, we limit the source XML document to be non-domain specific XML applications. The current objective is to investigate means, techniques, and common patterns to build a generalized mapping model for the transformation from XML to VXML, which will simplify the development of speech applications, especially on voice accessing and inputting information in the XML documents.

The focus of our work has been to design a Generic Architecture of Translating XML to VXML (GATXV) and to develop an XML to VXML Mapping Algorithm (XVMA). The generic architecture should cover the issues of the functionality, speech user interface and customized specifications in the transforming process. The mapping algorithms will be depicted in two forms, data access application and data input application. The generic architecture and algorithms should address not only the mapping rules but also a generalized model for constructing a voice application using VXML. Furthermore, we need to provide the user-customized means of access to the XML data.

This work is also trying to make a point that the multi-modal interface particular speech interface will proliferate. Speech will empower people to access information anywhere and anytime on a number of different devices, which is particularly important for mobile computing. Speech is the only modality that can provide users with a consistent interaction model for scenarios from office to home to mobile computing.

1.4 The Thesis

The thesis statement is:

"It is possible to automatically translate XML applications to VXML applications."

The following shows that how the thesis will be defended:

- Analyze XML.
- Analyze VXML.
- Divide applications into (a) data access applications (b) data input applications.
- Develop a notation for accessing XML data in documents.
- Define some desirable features of speech applications.
- Design generic architecture of translation.

- Develop algorithms for data access and data input from XML specifications and XML data.
- Use the algorithms to manually generate VXML applications.
- Investigate the result and analyze future process of automating the algorithms.
- Draw some conclusions as to what extent the thesis statement is true.

1.5 Organization of the Thesis Report

Chapter 2 introduces XML and its two schemas: XML DTDs and XML Schemas. Chapter 3 states the need for speech and presents the speech components that will be used in the design. Chapter 4 provides a general overview of VoiceXML, and its mechanisms that are most relevant to the mapping process. Chapter 5 presents the details of the generic architecture of transformation including the customization rules. We will investigate relationship between XML and VXML in Chapter 6. Then we will look at the possible mapping approaches that can be applied in both data input and access applications in Chapter 7. Next, we move on to look at mapping from XML Schema data types to VXML built-in grammars in Chapter 8. We present the XML to VoiceXML Mapping Algorithm (XVMA) for data access and data input applications in Chapter 9. Chapter 10 presents a speech system that is created as part of this work. Chapter 11 analyses the system design, looks at the observations we have made during the testing, suggests the further refinement of the system, and points out future research directions. Chapter 12 provides critical analysis of work done. Chapter 13 presents our conclusions.

Chapter 2

XML – THE EXTENSIBLE MARKUP LANGAUGE

As a markup language, XML is designed to be straightforwardly usable over the Internet, and supports a variety of applications [W3C XML]. Because of that, XML has been widely adopted and used in many areas since its release.

Our goal is to create rules for translating XML documents to VoiceXML documents. In order for better understanding the mappings between XML and VXML, this Chapter gives an introduction to the concepts of XML and discusses the syntax and characteristics of XML documents, particularly XML DTDs and XML Schemas.

2.1 Introduction to XML

XML, the eXtensible Markup Language [W3C XML], is a standard developed and introduced by the World Wide Web Consortium (W3C) for storing and exchanging data on the Internet. It is a markup language for representing data, rather than describing the presentation of the data. [Holman, 1999] noted that "*markup* is everything in a stream of information that is used to describe the data but isn't the data itself", which means that markup is the information about the data – metadata. As a markup language, XML provides a mechanism for storing structured data in text files.

XML is similar to HTML (HyperText Markup language) that is used for representing data in the web browsers. Similar in that XML contains tags and attributes as HTML does, and they both have similar hierarchical tree structure. HTML, However, is restricted to a predefined fixed set of tags and attributes. XML, on the other hand, allows users to define any tags and attributes. SGML (Standard Generalized Markup Language), the predecessor of XML, provides a mechanism for defining the markup vocabulary. Its notation for defining grammar rules is called DTD (Document Type Definition) [W3CGDTD]. Each element type is declared along with a rule restricting the form and the order of the contents of elements of that type. SGML is therefore not only a markup language but also a meta-markup language for formally describes different markup languages such as HTML. In fact, XML is more similar to SGML. Both of them are meta-markup languages (Figure 2.1).

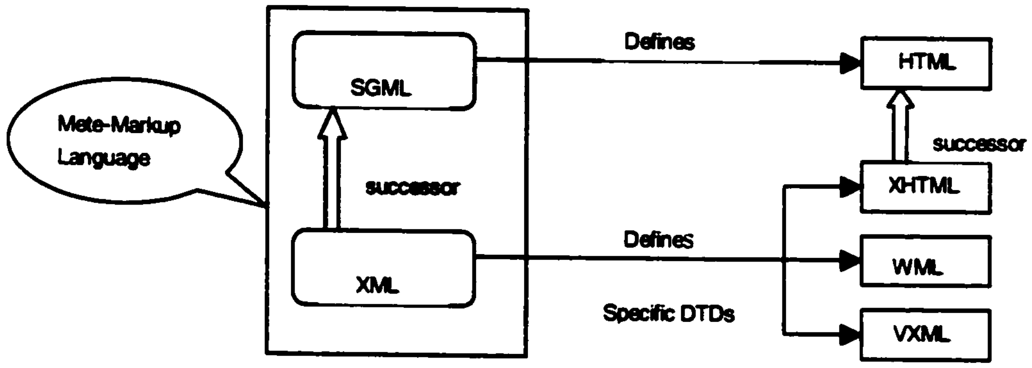


Figure 2.1 XML as a meta-markup language

The figure above shows that XHTML [W3C XHTML](a successor and a reformulation of HTML 4 in XML), WML [W3C WML] (Wireless Markup Language) and VXML [VxmlOrg, 2000] (Voice eXtensible Markup Language) are actually XML-defined applications. With object-oriented analysis, the relation between XML and its applications is a classic example of generalization. XML itself can be seen as an abstract base language whereas e.g. VXML is a specialized form of XML with limited vocabularies defined by XML.

There are many XML-related specifications such as XML Namespace, XPath, XPointer, DOM, SAX etc.; some of them will be discussed later. This section introduces the basic concepts of XML. A more comprehensive discussion with technical details can be found in [WroxXML, 2001] and [W3C XML].

2.2 XML Structure and Components

XML is a markup language that describes structured data in a way that separates the data from its presentation. This section will introduce structure and components of an XML document.

```
<?xml version="1.0" ?>
<courses>
```

```
  <course id="0360100">
     <title> Key concepts in computer science </title>
     <credit> 3 </credit>
  </course>
  <course id="0360797">
     <title> M.Sc. Thesis </title>
     <credit> 12 </credit>
  </course>
</courses>
```

Example 2.1 A sample XML Document courses1.xml

The first line of the above example is the XML declaration, which indicates that this file is an XML document, and it conforms version 1.0. The rest of the document defines the root element courses, also called document element, which includes element course that contains an attribute called id, elements title and credit. [RefsnesData] has some suggestions on when to use attributes, and when to use elements in XML documents.

http://www.w3schools.com/xml/xml_attributes.asp

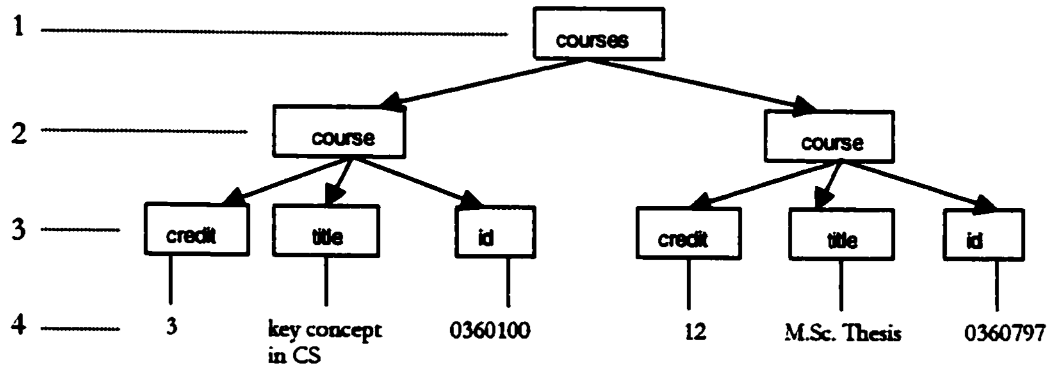


Figure 2.2 XML Tree structure for example 2.1

It is easy to see how the document listed in example 2.1 can be mapped into a tree structure in which the data are in a hierarchical format. The Figure 2.2 tree structure shows that elements and attributes construct the skeleton of an XML document. Each square box in the diagram represents a node. No distinction is shown between attribute nodes and element nodes for building the structure of the tree. We use shadowed squares to represent attribute nodes. The root element – *courses* is at level one, the branches in the second level of this tree structure are two *course* elements, and leaves in the third level of this graph are pointing to leaf contents that are either element' content or attribute's value at the fourth level.

With XML, it is possible and very easy to express the semantics of data clearly. For example (Example 2.1), a number three (line 5) stands for the credit of a course with the course number 03-60-510 rather than the number of students or the amount of money.

We have already seen that an XML document can form a tree structure by elements, attributes and its contents. Next, we will introduce and describe five components of an XML document. Example 2.2 shows an XML document that extends the previous example with a choice of *section* and *status* elements, *prerequisite* attribute and other elements. This XML document contains data about courses information offered by the University of Windsor for a semester, from which we will explain the components of XML documents.

1. **Processing instructions** Line one of example 2.2 contains the processing instruction that identifies the file as an XML document. A processing instruction is always in between "<?" and "?>" delimiters.

   Declarations in the DTD Line two of the example references a DTD file, *courses.dtd*, which contains the definition of the elements used in a set of document instances. A DTD file for an XML document is optional and it can be used for XML parsers to validate a class of XML document instances. We will give detailed discussion on declarations in the DTD in the next section.

```
1 <?xml version="1.0"?>
2 <!DOCTYPE courses SYSTEM "courses.dtd">
3
4 <courses>
5
6    <course id="0360100">
7       <title> Key Concepts in Computer Science</title>
8       <credit> 3 </credit>
9      <status> not offering </status>
10   </course>
11   <course id="0360140"> .. .. </course>
12   <course id="0360212"> .. .. </course>
13
14   <course id="0360254"
```

```
15           Prerequisite="0360100 0360140 0360212">
16       <title> Data Structure </title>
17       <credit> 3 </credit>
18       <!-- This is a comment -->
19       <section num="1" activity="lecture">
20       <classDay> Monday Wednesday </classDay>
21       <startTime> 11:30 </startTime>
22       <stopTime> 14:20 </stopTime>
23       <room> ER 3132 </room>
24       <instructor> Dr. Tjandra </instructor>
25       <examSlot> 1 </examSlot>
26    </section>
27    <section num="51" activity="lab">
28       <classDay> Thursday </classDay>
29       <startTime> 15:00 </startTime>
30       <stopTime> 16:20 </stopTime>
31       <room> ER 3146 </room>
32       <instructor > TBA </instructor>
33       <examSlot> 0 </examSlot>
34     </section>
35    </course>
36    ..
37    <course id="0360797">
38    <title> M.Sc. Thesis </title>
39    <credit> 12 </credit>
40    <section num="1">
45       <instructor > TBA </instructor>
46       <examSlot> 0 </examSlot>
47     </section>
48    </course>
49    ..
50 </courses>
```

Example 2.2 Sample of XML Document courses.xml

2. **Elements and attributes** Main parts of the document, which contain the actual data, begin at line four. Typically, an XML document consists of elements, delimited by "<" and ">", and attributes, for example in line 6, a name-value pair placed inside "< >" brackets after the tag name *course*. We will introduce elements and attributes declarations in 2.3.

3. **Comments** Line 18 is an example of a comment in an XML document, which is the same as in HTML document. Comments usually contain useful information for human readers of XML document.

4. **CDATA sections** In XML document some characters are reserved for markup, for example, "<" character initiates a start-tag. If such a character is needed as character data, it must be entered as a decimal or hexadecimal character reference, such as "`&#60;`" or "`&#x3C;`" for "<". CDATA sections are used to escape larger parts of text containing markup characters. For example, JavaScript code can be embedded in CDATA section in the format: `<![CDATA [anything goes here ]]>`.

An XML document to be considered well formed, it must meet the some criteria: (a) all un-empty tags must include an open and a close tag. (b) All empty element tags either have an open and a close tag or include a forward slash "/" right before the close tag ">". (c) All element open tags and close tags must be properly nested. (d) All attribute values must be enclosed in double quotation marks (line 14).

In addition to being well formed, XML specifications provide two different ways to specify that an XML document must be structured in a particular manner. This means that the appearance, order and some values of elements, attributes and other components of an XML document must conform to a predefined pattern. Such a pattern can be a DTD or XML Schema. An XML document is said to be valid if it conforms the constraints defined in a DTD or XML Schema. The transformation from XML documents to VXML is based on both DTD and Schema. The next two sections will discuss them in details.

2.3 DTD – The Document Type Definition

DTD comprises a set of declarations such as elements, attributes, entities, and notations, dictates where they may appear in the context of a document, and describes the types of the elements, attributes and acceptable attributes' values. A DTD is the metadata for an XML document instance. An XML document is perfect legal without a DTD, e.g., in the example 2.1. A DTD can either be included or referenced (example 2.2 line 2) by an XML document.

Four different types of declarations may appear in a DTD and they are element, attribute, entity and notation. We will introduce element and attribute because they are most relevant to the mapping process.

2.3.1 Element Declarations

Element declarations have the following format:

```
<!ELEMENT name content-model>
```

In this format, *`name`* is the name of the element being defined and the *`content-model`* is used to constrain the validation of the children of this element. Typically, the following examples appear at most cases.

```
<!ELEMENT courses (course*) >
<!ELEMENT instructor (#PCDATA|EMPTY)>
<!ELEMENT course  (title, credit, (status | section+))>
```

The first line defines the element *`courses`* which contains a repeated subelement *`course`* that can appear zero or more times. The second line indicates that the *`instructor`* contains a choice, which is either text content or being empty. The third line defines the *`course`* element as a sequence of a *`title`* element, followed by a *`credit`* element, and followed by a *`status`* element, or by one or more *`section`* elements. The order of these subelements is significant to an XML document instance in order to be considered valid. Some characters used in DTDs have special meanings, for example, `'*'`, `'+'`, and `'?'` are used to indicate 'zero or more', 'one or more', and 'zero or one' respectively.

2.3.2 Attribute Declarations

Attribute declarations have the following format:

`<!ATTLIST element-name name attribute-type default-value>`

In this format, *`element-name`* is the element with which the attribute is associated. *`name`* is the name of the attribute. *`attribute-type`* is the type that the attribute may have. *`default-value`* is the default value for the attribute that is optional. The following examples show the ways an attribute can be defined in a DTD.

```
<!ATTLIST course
      id            ID             #REQUIRED
      Prerequisite  IDREFS       #IMPLIED>
<!ATTLIST section
      num           CDATA          #REQUIRED
      activity      (lecture|lab)  #IMPLIED >
```

Above example line 1 to 3 are the definitions of attributes for the element *course*, in which two attributes *id* and *prerequisite* are defined.

The type of *id* is *ID*, which indicates that the value of *id* provides a unique name for the element *course*. The default value of *id* is *#REQUIRED* that indicates that an XML document instance must always supply a value for this attribute *id*.

The type of *prerequisite* is *IDREFS*, which indicates that the value of *prerequisite* must be some list of values of the *ID* attribute defined in the document instance. For example, from the example 2.2 line 14 we can see that the *prerequisites* of course *0360255* are *0360100*, *0360140*, and *0360212*. The default value for *prerequisite* is *#IMPLIED* that means that the attribute *prerequisite* is optional and has no default.

From line of 4 to 6 are attribute definitions for the element *section*. The type *CDATA* indicates that the value of attribute *num* could be any text string. The attribute *activity's* type is "*lecture|lab*", which is an enumerated list of the possible values of *activity*. The example can be seen on the example 2.2 in line 19 and 27.

Besides *#REQUIRED* and *#IMPLIED*, another possible default value of an attribute is *#FIXED*, which means that the attribute is not required, but if it exists, the value of this attribute must be set to the specified value. The complete DTD file *courses.dtd* for the example 2.2 *courses.xml* is given below:

```
<!ELEMENT courses(course*)>
<!ELEMENT course  (title, credit, (status | section+))>
<!ELEMENT title   (#PCDATA)>
<!ELEMENT credit (#PCDATA)>
<!ELEMENT status  (#PCDATA)>
<!ELEMENT section (classDay, startTime, stopTime, room,
                   instructor, examSlot)?>
<!ELEMENT classDay (#PCDATA)>
<!ELEMENT startTime (#PCDATA)>
<!ELEMENT stopTime (#PCDATA)>
<!ELEMENT room    (#PCDATA)>
<!ELEMENT instructor (#PCDATA)>
<!ELEMENT examSlot (#PCDATA)>
<!ATTLIST section num      CDATA        #REQUIRED
                  activity  (lecture|lab) #IMPLIED>

```

```
<!ATTLIST course id          ID #REQUIRED
                 prerequisite IDREFS #IMPLIED>
```

Example 2.3 Sample DTD courses.dtd

2.4 XML Schema

Both DTD and XML Schema are used "for describing and constraining the contents of XML documents" [W3C Schema]. So far, all the XML applications have been defined by DTDs. Using a DTD to describe and constrain the contents of an XML document has raised some attentions due to its lack of data types and different syntax from XML instances. XML Schema is a mechanism created to overcome these limitations, which just became a W3C recommendation in May 2001. Unlike the DTD syntax, an XML schema allows the user to define the rules governing the relations between elements and attributes using a standard XML instance syntax, which can be parsed and managed by using XML applications. An XML schema also adds support for namespaces, data types, and features like range constraints, where we can declare, reuse, and inherit data definitions and structures.

2.4.1 An Overview of XML Schema

In this section, we will illustrate some basics of XML Schema through examples. For instance, in the example 2.1 courses1.xml, an element *credit* in DTD is shown as follows:

```
<!ELEMENT credit (#PCDATA)>
```

The element type *#PCDATA* (parsed character data) is a text string, and the value for an attribute of type *ID* must be unique, which is a text string too. Using DTD, we are unable to specify *credit*, for example, to be an integer number with the range from 0 to 12. On the other hand, we can constrain the value of the element *credit* to be exactly what we want by the following XML Schema snippet:

```
<?xml version="1.0">
<xsd:schema>
   <xsd:element name="credit">
       <xsd:simpleType>
           <xsd:restriction base="xsd:integer">
                <xsd:minInclusive value="0"/>
                <xsd:maxInclusive value="12"/>
```

```
            </xsd:restriction>
        </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Example 2.4 Schema - Element credit

If we don't know anything about the XML Schema, from the above example, we at least can tell that this is a well-formed XML document and the root element is `xsd:schema`. Each element with a prefix `xsd` is associated with the XML Schema namespace. The namespace should be always declared at the beginning of every W3C XML Schema (see Appendix-1, Appendix-2 for examples). XML namespace is a solution to the problem of name conflicts [W3C Namespace].

Example 2.4 defines the type of element `credit` as a `simpleType`, which can hold an integer value, a XML Schema built-in data type, and its minimal value is 0, maximal value is 12. This `simpleType` is a new data type that can be defined by specifying the value of one or more *facets* from an existing data types – the base type. The facets of each built-in data types can be found in [W3C Schema]. For example, the integer data type in XML Schema has facets such as *pattern, enumeration, whitespace, maxInclusive, maxExclusive, minInclusive, minExclusive, precision, and scale.* In this example, we specified minInclusive and maxInclusive facets.

XML Schema has two principal types of definitions – *complexType* and *simpleType*, which are created by restricting or extending other type definitions. Elements that contain subelements or carry attributes are said to have complex types, whereas elements that contain numbers (and strings, and dates, etc.) but do not contain any subelements are said to have simple types. Some elements have attributes; attributes always have simple types.

An alternate form of example 2.4 is to set a named simpleType `creditType`, and use that type as a value of the `type` attribute in `xsd:element`. See below:

```
<xsd:element name="credit" type="creditType">

<xsd:simpleType name="creditType">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="0"/>
```

```
        <xsd:maxInclusive value="12"/>
    </xsd:restriction>
  </xsd:simpleType>
```

Example 2.5 Schema – element credit with type attribute

The attribute *type* can also point to a complexType. There are other alternatives to declare elements. We will see the examples later. Following the *credit* element declaration in XML Schema, now let us look at the attribute *id*. The DTD for *id* attribute is like:

```
<!ATTLIST course id ID #REQUIRED>
```

In DTD, we can only denote that the type of *id* is *ID*, which is a text string. What if we want to indicate that *id* has the pattern *"ddddddd"*, where *d* stands for a digit number? XML Schema has ability to describe such constraints for elements or attributes, see below:

```
1 <xsd:element name="course">
2    <xsd:complexType>
3       <xsd:attribute name="id" use="required">
4         <xsd:simpleType>
5           <xsd:restriction base="ID">
6               <xsd:length value="7"/>
7               <xsd:pattern value="\d{7}"/>
8           </xsd:restriction>
9         </xsd:simpleType>
10      </xsd:attribute>
11   </xsd:complexType>
12</xsd:element>
```

Example 2.6 Schema - Attribute id – use of inline format

Element *course* has an attribute *id*, so it has a *complexType*. Line 3 to 10 defines an attribute with *name="id"*, *use="required"*, two attributes of the element *xsd:attribute*. In line 4, a new data type is created for attribute *id*, which is a refinement of a built-in type (*ID*), so we use *simpleType* to identify it. Line 6 and 7 indicate that this *simpleType* based on type *ID* that is a name must be of *length* 7 and the name follows the *pattern*: 7 digit numbers. The expression *"\d{7}"* used in pattern is regular expression. The following XML fragment illustrates the effect:

```
<course id="0360100"> </course>
```

Another format for example 2.6 is to use attribute `ref` of *xsd:attribute*, and this `ref` attribute points to a declaration of attribute *id*. See the following Schema:

```
<xsd:element name="course">
     <xsd:complexType>
     <xsd:attribute ref="id" use="required">
     </xsd:complexType>
 </xsd:element>

 <xsd:attribute name="id">
     <xsd:simpleType>
      <xsd:restriction base="ID">
        <xsd:length value="7"/>
        <xsd:pattern value="\d{7}"/>
      </xsd:restriction>
     </xsd:simpleType>
 </xsd:attribute>
```

Example 2.7 Schema – Attribute id – use of ref format

The similar format can apply to the element too. For example:

```
1 <xsd:element name= "section">
2    <xsd:complexType>
3       <xsd:sequence>
4         <xsd:element ref="instructor"
5                    minOccurs="0" maxOccurs="1"/>
6         <xsd:element ref="examSlot"/>
7       </xsd:sequence>
8    </xsd:complexType>
9 </xsd:element>
10
11 <xsd:element name="instructor" type="xsd:string"/>
12 <xsd:element name="examSlot" type="xsd:integer"/>
```

Example 2.8 Schema – element declaration – ref format

The above example illustrates referencing a child element declaration with attribute `ref` in line 4 and line 6. In line 5 two facets, *minOccurs* and *maxOccurs*, are used to constrain the element *instructor* to occur at most once or no occurrence at all. The default value is the same for minOccurs and maxOccurs, and it is "1", so in the line 6 no minOccurs and maxOccurs are specified for element *examSlot*, which indicates that the element *examSlot* should occur exactly once in section.

If there are several subelements for one element, we can group them together by using *xsd:group* element with attribute *ref* to point to the actual group definition. The following schema fragment is an equivalent way of representing the schema shown in example 2.8.

```
<xsd:element name= "section">
   <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="sectionElements"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:group name="sectionElements" />
   <xsd:sequence>
      <xsd:element name="instructor" type="xsd:string"/>
      <xsd:element name="examSlot" type="xsd:integer"/>
   </xsd:sequence>
</xsd:group>
```

Example 2.9 Schema - element declaration - group

Similar to the element declarations, we have several choices to declare an attribute: referencing declaration, using in-line declaration, using named type or using group definition. In addition, we can use XML Schema built-in data types and previously declared types including simpleType and complexType to extend or restrict from them to create new data types.

From the examples showed above, we can see that XML Schemas provide a means to define an abstract data model for a class of documents. While having some of the capabilities provided by XML DTDs, they significantly extend their power and provide for much tighter validation of document. In the Appendix -1, -2, we provide the complete XML Schema files for the example 2.1 and 2.2 in which we try to use different formats to declare the elements and attribute, to define simpleTypes, and complexTypes. We also use some built-in data types, namespace, identity-constraint component, notations, schema elements such as, xsd:choice, xsd:list in the examples.

In the next sections, we will try from conceptual level to summarize the advantages of XML Schema and identify the main components that will useful in the transformation.

2.4.2 The features of XML Schema

DTD and XML Schema are used to specify the structure and content constraints of XML instance documents. Comparing DTD with XML Schema, we can see that DTD uses a syntax that is inconsistent with XML format. In addition, DTD supports a very limited capability for specifying data types – only 10 data types total, which are all text strings. On the other hand, XML Schema has advantages over DTD:

- XML Schema has the same syntax as XML instance documents, which takes advantages of numerous XML supporting tools.

- XML Schema supports more than 44 given data types, and allows users to define their own data types.
- XML Schema and its extensibility put a great deal of effort in its expressiveness for data models and relationship between them. For example, you can always derive a new data type from existing types, define substitutable element, abstract type and final type, which are the features derived from object-oriented design in a sense of inheritance and overriding. XML Schema has the mechanism to express sets – collections of data as well.

- XML Schema supports namespace, and its modularity makes it easy to reuse parts of a schema. Namespace allows the use of any prefix in instance documents, but also opens schemas to accept elements and attributes from known or unknown namespaces. Annotations provide an alternative to XML comments and processing instructions that might be easier to handle for supporting tools.

2.4.3 XML Schema Components

"XML component is the generic term for building blocks that comprise the abstract data model of the schema. An XML Schema is a set of schema components" [W3C Schema]. In "20010502" version, XML Schema consists of 13 kinds of component that are classified under *primary*, *secondary* and *helper* categories. The figure 2.3 shows all components in a hierarchy structure.

In the helper group, *wildcards* component is a special component. It "matches element and attribute information items dependent on their namespace name"[W3C Schema].

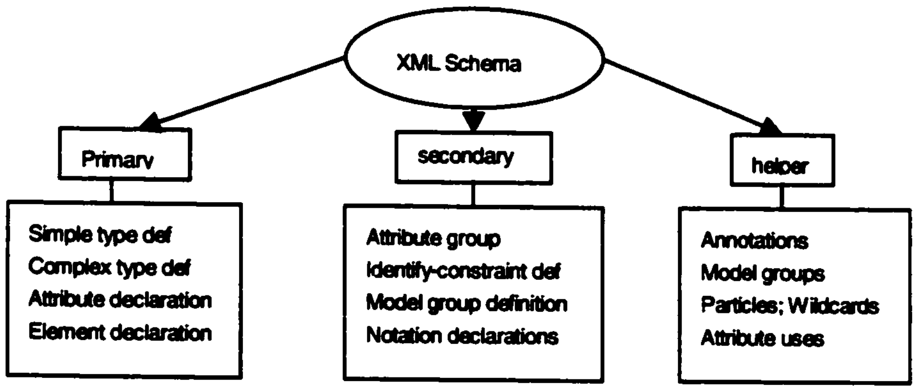


Figure 2.3 Schema Components [modified Wrox Java 2001]

Primary components are basic building blocks of XML Schemas. With the secondary components, the schema document can be optimized to a different structure for the purpose of the well-organized readable document, though the constraints defined in the schema may be the same. In addition, every secondary component must have a name associated with it. The helper components by literal can help to form parts of other components. These three groups are related very tightly. For example, we know that "the model group, particle, and wildcard components contribute to the portion of a complex type definition that controls an element information item's content"[W3C Schema]. Thus, a complex type, a primary schema component, can consist of a model group definition, a secondary schema component, with a name referenced to the real content of the complex type which embodies in a model group, a helper schema component. The figure 2.3 shows a hierarchical architecture, which "provides a very natural sequence with which to conceptualize a data structure and to visualize how this data will be broken down" [WroxJava, 2001].

XML Schema has 44 built-in data types and each data type associated with a set of applicable facets. There are 36 elements and 34 attributes in XML Schema used to specify the constraints of XML document instances. For the complete reference to XML Schema, please see [W3C Schema]. The focus of this work is tightly related to the structure of the XML document and the data types that have a great impact on VoiceXML applications.

Understanding how a schema is constructed with all of XML Schema components is very helpful for the further discussion of transforming XML documents to VXML documents.

Chapter 3

SPEECH APPLICATIONS

Speech technology has been an ongoing research topic since the 1950s, and it has improved significantly over the past few years. With the explosive growth of WWW and recent advances in wireless communication, speech is going to become a viable alternative to traditional methods of interacting with computers using keyboards, mice, and graphical user interfaces. Speech supports a multi-modal user interface, which provides a more natural way for humans to communicate with computers.

In this Chapter, we will first discuss why do we need speech and some basic concepts of speech technology. Then, we talk about four phases of speech user-interface design and compare speech with graphics. Next, we give a brief introduction to Java Speech Grammar Format – JSGF, which will be used in next Chapter to serve the VoiceXML.

3.1 The Need for Speech

Speech has been around many years. The first reason for voice application to be created was for visually impaired people. However, speech applications were not widely used until recent years, when wireless communication technology moved forward, and speech recognition and text to speech synthesis technologies breakthroughs. All of these made it possible to access information from any place, at any time by only using a mobile phone. The needs for speech also comes from:

- Restrictions imposed by circumstances. For example, when you are traveling, and you need to access some information residing on a remote server, and you don't have a laptop or desktop to access the Internet. However, in most cases, you do have access to a telephone or mobile phone. The voice application is perfect fit in with this situation.

- The needs required in hands-free and eyes-free environments. Hands-free, eyes-free is unlike applications that rely on telephone key-press input. In this environment, speech will enhance productivity. For example, mobile users can dictate notes into a handheld recorder, keeping their hands free for driving. Later, the notes are transferred to the desktop computer and automatically converted to text.
- Multi-modal user interfaces and customer services. The rapid pace of business today requires employees and customers to have fast, constant access to information. Speech-enabled applications can eliminate the constraints of the telephone keypad and allow easier-to-use-automated systems to provide a more economical service. A speech-enabled application can replace frustrating "Press 1 for choice x" interfaces with simpler, "Say the name of the person you'd like to reach" responses. For example, if you have used "Ontario road test automated booking system", you must know how frustrating it is. When you want the computer to know where you live such as "Windsor", you have to listen, follow the instruction, and press the keys several times to confirm one required item.

Speech-based user interfaces are growing in popularity. Unfortunately, the technology expertise required to build speech user interfaces prevents many individuals from participating in the speech-interface design process. Furthermore, the time and knowledge costs of building even a simple speech system make it difficult for designers and developers to build speech applications.

As we have already seen in Chapter 1, when constructing a speech application, we have some choices. One choice is to build a special-purpose voice application system such as SpeechWeb [Frost2, 1999] – constructed as a set of executable attribute grammars [Frost, 1994] [Frost, 1992]. Second, use programmatic methods – Java Speech API with CGI, Java Servlet in a multi-tier architecture to build an interactive speech application. These two approaches can be used to construct a typical dialog system, such as [Aust, 1995] the Philips train timetable system, and [Zue, 2000] Jupiter weather report system. Such systems are particularly tailored to fit user's requirements and conform speech application constraints. Third, use an automatic translation system, which is what this thesis report is concerned. In this work, we map XML data to VXML, which means that the element's content and attribute's value and metadata stored in XML documents are translated into VXML interactive dialogs. The user gets the VXML document that can be deployed and executed right away in a computer or telephony environment. Through the automated translation, we will avoid some difficulties in the process of building speech applications.

These approaches are going to find wide acceptance in different scenarios. In theory, special-purpose systems should work best. However, it is going to take a long time before such systems are mature and scale well. In addition, with large speech components, special-purpose system needs the special knowledge and experience in programming and speech interface design. In this work, we are trying to provide a way to map XML documents to VXML documents and meanwhile to considering speech user interface issues to ease the development of voice applications.

3.2 Speech User Interface Design

The field of speech user interface design has been studied and results have been effective. "For computer input, the user interface design addresses what a user can say and when it can be said. For computer output, the design specifies computer prompts, feedback to users, and other spoken and non-speech output. In combing input and output, user interface designs also consider dialog flow, timing, help, disambiguation, recovery for errors and more"[Hunt, 2000]. Like developing a graphical user interface, speech user-interface design always involves the following iterative phases: design phase, prototype phase, test phase, and refinement phase [IBM, 2000].

Design Phase The goal of this phase is to identify the functionality of the application. The activities engaged in this phase including analysis of user requirements, designing the user interface to match the task and operating conditions.

Prototype Phase The goal of this phase is to build a trial system with the user interface designed in the last phase. A method called 'Wizard of Oz' [Klemmer, 2000] is often used for this purpose. The method involves two people in which one person plays the role of the machine and the other plays the role of the user to test the initial script in order to fix problems in the script and the task flow.

Test Phase In the test phase, we need to test the system with real users to find and correct the problems could not be found in Wizard of Oz scenarios.

Refinement Phase In the refinement phase, we evaluate the overall system performance, make improvements in the UI design based on the results of evaluation and previous phases.

Note that above four phases are iterative that means we need to go back to each phase, detect and correct problems in the previous round until the application reaches the system requirements.

It is a challenge to create an intuitive graphical user interface, but it is more extremely difficult to design an intuitive speech user interface. There are some fundamental differences between speech and graphic user interface [Hunt, 2000]. *Visibility*: graphical interface is by literal visually available and its functionality is very easy to know for the user. In contrast, speech is unseen, which makes it problematical to communicate the functional boundaries of an application to the user. For example, users do not know when and how to order a command to the computer, and if something goes wrong, the user may have a hard time to noticing the problem. *Transience*: Speech is transient, which means that whenever you hear it or say it, it is gone. On the contrary, a graphical interface may leave the information to the user for further reference. *Bandwidth asymmetry*: speech input is typically much faster than typed input whereas speech output can be much slower than reading graphical output [Hunt, 2000].

3.3 Java Speech Grammar Format

"The Java Speech Grammar Format (JSGF) is a platform-independent, vendor-independent textual representation of grammars for use in speech recognition" [Sun JSGF]. JSGF can be used with the VoiceXML <grammar> element to indicate what the recognizer should listen for, and to describe what a user may say at a certain time. A complete description of JSGF is available at [Sun JSGF], so only a brief review of JSGF syntax is presented here. JSGF uses traditional grammar notations, and a JSGF grammar source file (sometimes referred to as a grammar) contains a header and a body. The JSGF grammar filename must end in the extension ".gram". Each statement in the grammar file, whether part of the header or a rule definition in the grammar body, must be terminated with a semicolon.

Every JSGF grammar file contains a grammar header as the first line, a string that identifies it as a JSGF grammar. A self-identifying header line which begins with the string "*#JSGF*", contains additional version information (V1.0 refers to the version 1.0 of the JSGF specification)s, and may contain locale or character coding information. The "#" must therefore be the first character of each JSGF grammar source file. In most cases, the first line of the grammar file will be:

```
#JSGF V1.0;
```

Each grammar file header should contain the grammar name declaration, which consists of the grammar statement followed by the name of the grammar file with the "*.gram*" extension omitted. The grammar file *mygram.gram*, for instance, would contain the statement:

```
grammar mygram;
```

Additionally, using the import statement, the header may declare rules to be imported from other grammar files. The rulenames to be imported must be specified such that they are unambiguous (i.e. containing the full grammar name and rule name). The special rule "*" can be used in an import statement to import all public rules from a grammar. The following statement imports rule `rule1` from grammar file *myimports.gram:*

```
import <myimports.rule1>;
```

The body of each JSGF grammar is composed of one or more rule definitions or expansions. A JSGF rule specifies how a single rulename (i.e., non-terminal) can be expanded in terms of words (i.e., terminals) and/or other rulenames.

By default, every JSGF rule is defined as a private rule, that is, it may only be referred to within the context of the grammar in which it is defined. Rules may be declared as public by prefacing the rule definition with the word `public`. In order for a rule to be imported into another grammar, or for the recognizer to activate a rule, the rule must be declared as public. The following list contains examples of JSGF syntax, which may be used in rule expansions to combine tokens (either terminals or non-terminals). A complete and detailed explanation of JSGF syntax can be found in version 1.0 of the JSGF specifications [SUN JSGF].

|	alternatives
( )	grouped tokens
*	zero or more occurrence of a token
+	one or more occurrences of a token
[ ]	optional grouping of tokens
{ }	associates the enclosed character string tag with a token or group of tokens

Figure 3.1 JSGF syntax

In addition, JSGF grammars can contain C, or C++ style comment declarations. JSGF grammar examples can be found in the Chapter 5 example 5.1, example 5.2, and example 5.3.

Chapter 4

VOICEXML

VoiceXML – Voice Extensible Markup Language, is based on the World Wide Web Consortium's industry-standard eXtensible Markup Language (XML). VoiceXML 1.0 specification released in March 2000 provides a high-level programming interface to speech and telephony resources for creating distributed voice applications, much as HTML is a markup language for creating distributed visual applications.

A brief overview in terms of architecture and components of the VoiceXML system is presented first in this Chapter. Then VoiceXML features and comparison with SpeechWeb are presented.

4.1 Overview of VXML

"VoiceXML is designed for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and DTMF key input, recording of spoken input, telephony, and mixed-initiative conversations" [VxmlOrg, 2000]. The explosive growth of the Internet and World Wide Web technologies has shifted the landscape for providers of traditional telephone services to a new set of customers accessing information and services through the Web. Although excellent speech recognition engines are available, speech recognition applications are still difficult to build. VoiceXML is an attempt to remedy this problem, and it allows building and deploying voice applications easier for both telephone services providers and developers.

4.1.1 Architecture of a VoiceXML System

The basic structure of a VoiceXML system is shown in figure 4.1. The typical scenario is a customer using a conventional landline or cellular phone to call a designated phone number. The call is answered by a computer system at a Voice Browser site.

The Voice Browser [W3C Voice] (also called voice gateway) retrieves the initial VoiceXML script from a VoiceXML server, which can be local to the browser or located anywhere on the World Wide Web. A portion of the Voice Browser called the interpreter parses and executes this Script playing prompts, hearing responses and passing them on to a speech recognition engine that is also part of the Voice browser system.

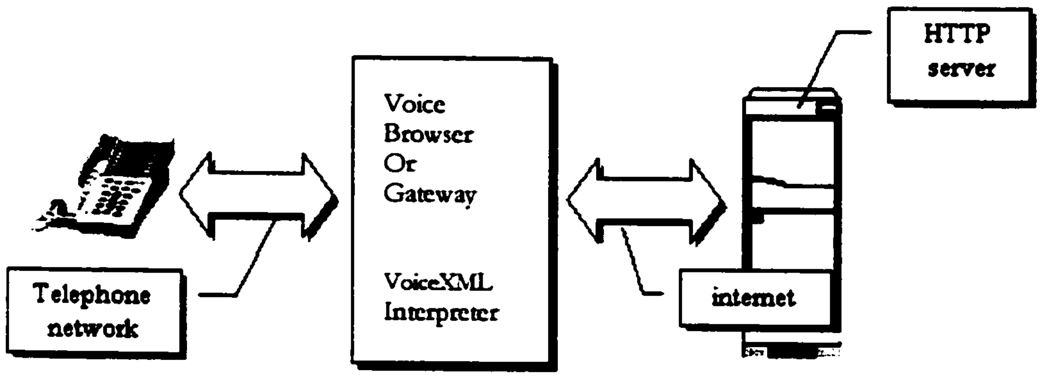


Figure 4.1 Structure of VoiceXML System[Martin, 2000]

When the script has collected all the necessary responses from the user, the interpreter assembles them into a request to the VoiceXML content server through HTTP protocol. The HTTP server responds with a dynamically generated VoiceXML page containing the information requested by the user. The process can be repeated indefinitely to produce the appearance of a conversation between the user and the VoiceXML server.

4.1.2 Voice Browser

A VoiceXML system typical includes the following components (figure 4.2): VoiceXML interpreter, telephony services, speech recognition, text-to speech generation and HTTP client services. Those components combine and cooperate together to become what we call a *voice browser* which resides in between the human and the computer to fetch voice, DTMF (touch-tone) input, pre-recorded speech or synthesized speech.

Among the above components, the VoiceXML interpreter begins parsing through and executing the instructions in the VoiceXML script. When the script indicates that user input is required, the interpreter hands off control to a speech recognition engine that "hears" and converts the spoken response to text.

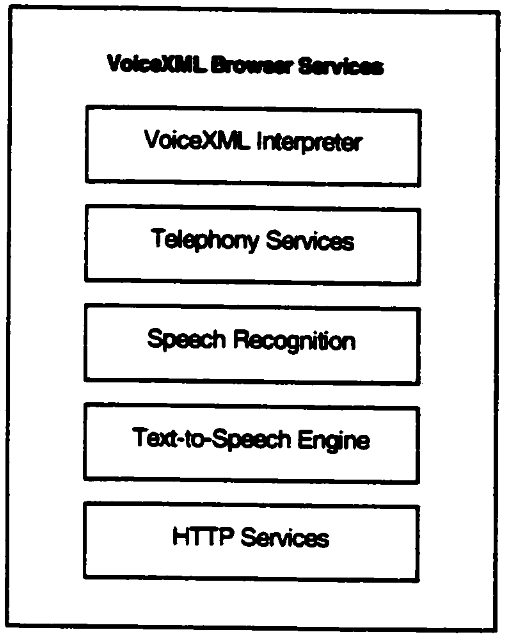


Figure 4.2 Architecture of VoiceXML System

Grammars for speech recognition are stored with the content on the VoiceXML server. They may be in a custom format specific to the recognition engine used, or be written in standard Java Grammar Specification Format (JGSF).

VoiceXML currently provides two mechanisms for generating speech or other audio output. Recorded sound in WAV or similar formats can be used to speak information or prompts. A Text-To-Speech (TTS) engine can perform the same function. Like the recognition engine, any compatible TTS engine can be used to power a voice browser. Recorded audio is delivered by specifying the URL of the WAV file. The audio files may be located on remote web servers or cached locally.

Finally, the gateway also serves as an HTTP client sending messages and receiving VoiceXML pages from the web server. Communications between the gateway and the content server follows standard HTTP protocols. Outgoing requests are in the form of an HTTP "get" or "post" command.

4.1.3 VoiceXML – Web-based Application

VXML is used at the same stage as HTML is in the application development process, but the difference is that HTML is used for Web-based applications (visually), whereas VXML is used for developing speech applications. The following diagram illustrates it:

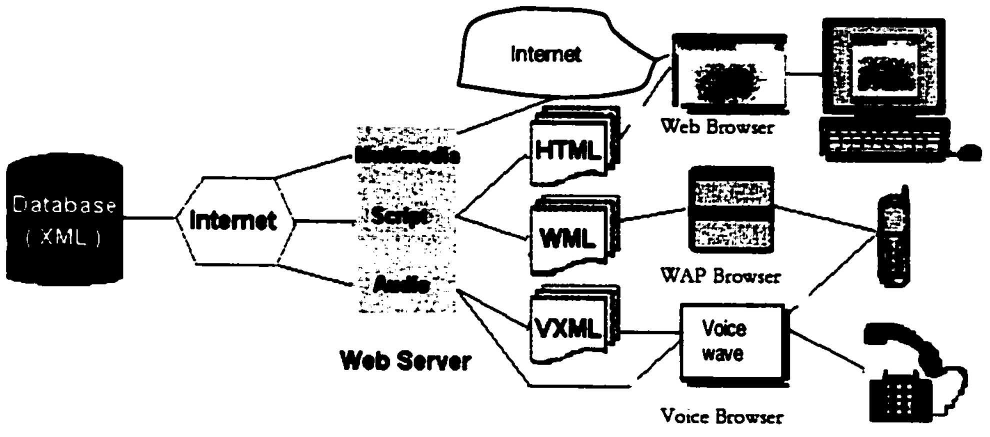


Figure 4.3 VoiceXML as a Web Application [modified MotorolaMix]

In a typical multi-tier Web-based application, the communication process between the client and server can be described as follows:

- The user sends a request, for example, to check the weather, by inputting the city, province and country and clicking on a link in the web browser through a terminal or PC.
- The web server gets the request and forwards it to data repository.
- Business logic is invoked and a result (XML format) is returned from the database to the web server
- Web publishing server manages XML data with specified XSL stylesheet to produce a HTML web page and sends it back to the client

VoiceXML provides an alternative presentation of data, which is platform-independent, amenable to tools and techniques familiar to web developers, and accessible to anybody with a telephone. Here is the route for a VXML application having the similar application logic with the above infrastructure:

- The user sends a request, for example, to check the weather, by speaking the name of the city, province, and country.
- The speech recognition engine matches the utterance against the active grammar, then uses the action tags to add semantic to the city, province and country holders.
- The data is then sent to the web server and forwarded to the data repository.
- Business logic is invoked and a result (XML format) is returned from the database to web server.
- Web publishing server manages XML data with specified tools or a XSL stylesheet to produce a VXML page and the TTS engine sends the voiced result back to client.

As you can see, a GUI and a VUI with two different means to represent data to users might use the same business logic and application logic at the back end. Therefore, VXML is relatively easy for web developers to grasp and use it right away after a few examples. Next section we will take a close look at VoiceXML and its building blocks.

4.2 Usability, Features & Drawbacks

In this section, we will emphasize the issues about how to use VoiceXML in practice, advantages of VoiceXML and its drawbacks.

4.2.1 Ease of Use of VoiceXML

The main goal of VoiceXML is to "bring the full power of web development and content delivery to voice-response applications" [VxmlOrg, 2000], and to free the authors of such applications from low-level programming and resource management.

The VoiceXML language has a control flow mechanism and a well-defined semantics that preserves the author's intention regarding the behavior of interactions with the user. This point will be demonstrated in the example 4.1. Those features make VoiceXML straightforward and easy to implement for developers.

The basic spoken dialogues of VoiceXML are *<forms>* and *<menus>*.

- *Forms and Form Items*

Forms allow the user to provide voice or DTMF input by responding to one or more *<field>* elements. Each field can contain one or more *<prompt>* elements that guide the user to provide the desired input. Another type of form item is the *<subdialog>* element, which creates a separate execution context to gather information and return it to the form. The following is a simple VoiceXML "page" with *<form>* element:

```
1 <?xml version="1.0"?>
2 <vxml version="1.0">
3    <form id="welcome">
4       <block>
5         <prompt> Welcome to University of Windsor.
6           <audio src="http://speechlab/welcome.wav">
7              This is the University of Windsor
8              automated inquiry system.
9           </audio>
10          </prompt>
11      </block>
12   </form>
13 </vxml>
```

Example 4.1 VXML - <form> welcome

The example above consists a single *<form>*, which prompts users a welcome. As we can see, the first line is an XML file declaration. Since a VXML document is an XML document, this line should always appear at very beginning of any VXML document. The second line is a VXML document declaration. Line 3 to 12 is a *<form>* dialog in which the *<block>* element is a form item that denotes a container for executable code. For example *<prompt>* is an executable element, which means that text strings in between open *<prompt>* and close *</prompt>* will be output to the user by the voice browser using TTS. Inside the *<prompt>*, the element *<audio>* includes an audio clip and the audio tag can have alternate text in case the audio sample is not available. *<form>* has an attribute called *id* that is an ID data type defined by VoiceXML DTD, which means that *id="welcome"* for this *<form>* is going to be unique in this VXML document. By doing this we can manage transitions between forms.

<field> is one of the most important form items, which gathers user input based on active grammars or DTMF. See the following example:

```
<?xml version="1.0"?>
<vxml version="1.0">
  <form id="course">
    <field name="courseId" type="digits">
        <prompt>Please say a course id.</prompt>
       <help> course id has 7 digits number </help>
       <filled>
         <if cond="courseId.length!=7">
            <throw event="help"/>
            <assign name="courseId" expr="undefined"/>
         </if>
       </filled>
    </field>

    <field name="title">
       <prompt> What is the course title?</prompt>
       <grammar src=http://speechlab/title.gram
                type="application/x-jsgf"/>
    </field>

    <field name="credit" type="number">
       <prompt> How much credit of the course
                <value expr="courseId"/>
       </prompt>
       <filled>
         <submit next=http://speech/place_record.jsp
                   namelist="courseId title credit"/>
         <clear namelist="courseId title credit"/>
       </filled>
    </field>

  </form>
</vxml>
```

Example 4.2 VXML – course.vxml <field> input record

When VoiceXML browser executes this page, the following dialogue between a computer and a user may occur:

```
Computer:  Please say a course id.
User:      036022
Computer:  course id has 7 digits number
Computer:  Please say a course id.
User:      0360100
Computer:  What is the course title?
User:      Key concept in Computer Science.
Computer:  How much credit of the course 0360100?
User:      3
```

Above example has one <form> that includes three <field>, and each <field> takes one user input. This is *directed* form, which means that each field or other form item is executed once and in a sequential order, as directed by computer. Another form is *mixed initiative* form, which means the user could direct the conversation. It is achieved by using form-level grammars or document-level grammars (field-level grammars are only active in the scope of fields) and allows the user to full in multiple fields from a single utterance. For above example, some explanation are given below:

The first `<field>` accepts `courseId` that is the name assigned to this `<field>` in line 4. An attribute `type` is used to "specify a *built-in grammar* for one of the fundamental types, and also specifies how its value is to be spoken if subsequently used in a value attribute in a prompt" [VxmlOrg, 2000]. In this case, the `digits` is stored as a string, and rendered as digits, i.e., "four-five-six", not "Four hundred fifty-six". Line 7 is the `<filled>` element, which specifies an action to perform when the field is filled by user input. At line 10, `<throw event="help">` throws a help event when users input does not match the requirement of the length of `courseId`. At line 6 is the catch for help event, which is a shorthand for `<catch event="help">`. The VXML interpreter provides implicit default catch handlers for the `noinput`, `help`, `nomatch`, `cancel`, `exit`, `and cancel` and `error` events if the author did not specify them.

The second `<field>` accepts a title for the course from line 15 to 19. In this field, a `<grammar>` element is used to specify an *external* grammar file via a URI. The grammar file can be local or remote. Grammars can also be specified *inline*, for example using JSGF (Java Speech Grammar Format) [SUN JSGF] we can specify the title should be either of the following two between open and close `<grammar>`:

```
<grammar> Key concept in Computer Science |Background
          Reading </grammar>
```

The third *`<field>`*, from line 21 to 30, is used to collect the credit of the course. The credit is specified with the grammar type number. In the *`<prompt>`*, we use *`<value expr="courseID">`* as part of prompt to guide the user to provide the desired input, which is the value of the first *`<field>`* input. From line 25 to 29 is a *`<filled>`* element, appearing inside this field, which specifies an action to perform after this field is filled in by user input. The action is to submit all the field inputs denoted by a *`namelist`* attribute to a JSP page that could either store this data into a database or pass it to another process.

One important note is that when this VoiceXML page is rendered by a voice browser, the form interpretation algorithm (FIA) [VxmlOrg, 2000] is executed. The FIA has a main loop, which is responsible for repeatedly selecting form items e.g. a field item, an event handler, and then visiting the first suitable one. In the example 4.2, the voice browser will visit the first form item whose variable is undefined, which is the first *`<field>`* *`courseId`*. When the input is captured by the field variable *`courseId`*, this field is no longer undefined. The FIA enters *`<filled>`*, if the condition of the *`<if>`* statement is not true, we need to re-enter the courseId. In VXML, this situation does not need a *`<goto>`* element to jump to the beginning of this field, we only need to assign the field variable courseId to undefined in line 10 shown below, the FIA will automatically select and visit this field again for the next round. Of course, the FIA is also responsible for other tasks during the form rendering, for details of FIA, please see the document [VxmlOrg, 2000].

```
10      <assign name="courseId" expr="undefined"/>
```

Another thing to point out is the VXML built-in grammars, which determine valid spoken and DTMF input and affects how the TTS engine, will pronounce the value in a prompt. For example, if the input *`"one fifty two"`* as the type *number*, it would be *`"152"`*, in contrast, as the type *currency*, it would be *`"1 dollar 52 cents"`*. Similar for the number *`"1.52"`* for *number* type is pronounced as *`"one point fifty two"`*, for *currency* type, however, is pronounced as *`"one dollar and fifty two cents"`*. From these examples, we can see the grammar type has a deep impact on representing what the people really mean when they say something. The VoiceXML 1.0 specification defines seven built-in grammar types, and they are *boolean*, *currency*, *date*, *digits*, *number*, *phone*, and *time*.

Field items can be used with option lists:

```
1 <?xml version="1.0"?>
2 <vxml version="1.0">
3    <form>
4        <field name="welcome">
5          <prompt>
6          <audio src="http://speechlab/welcome.wav">
7          </audio>
8           Welcome to University of Windsor
9           automated inquiry system, we're featuring
10          <enumerate/>
11         </prompt>
12
13         <option dtmf="1" value="library"> library info</option>
14         <option dtmf="2" value="course"> course info</option>
15         <option dtmf="3" value="service"> campus services </option>
16
17        <filled>
18             <goto expr="welcome+'.vxml'"/>
19        </filled>
20
21      </field>
22   </form>
23</vxml>
```

Example 4.3 VXML - <option> & <goto>

It is more convenient to use an <option> when a simple set of alternatives is all that is needed to specify the input values for a field. In the example above, three *<option>* elements with the contents and *dtmf* attributes (line 13 to 15) are used to generate two kinds of grammars: speech grammar and DTMF grammar. Speech grammar is for the spoken input, whereas the DTMF (Digital Tone Multiple Frequency) grammar is for the touch-tone key input. This example will accept both spoken input and touch-tone key input. The *value* attribute of the *<option>* is used to assign a value to the field item variable, which in this case is *'welcome'*. For example, when the user input is *"course info"* or the DTMF touch-tone is *"2"* (line 14), the <goto> element (line 18) will form the target VXML file dynamically based on the user input, so the transition will go to the *"course.vxml"*. The contents of *<option>*s are used to form the *<enumerate>* (line 10) prompt strings.

- *Menus*

See an example below:

```
<?xml version="1.0"?>
<vxml version="1.0">
    <menu>
         <prompt>
           <audio src="http://speechlab/welcome.wav">
           Welcome to University of Windsor
           automated inquiry system, please choose
           register, course or facility for information.
           <enumerate/>
         </prompt>

         <choice next="http://speechlab/register.vxml">
              register information
         </choice>
         <choice next="http://speechlab/course.vxml">
              course information
         </choice>
         <choice next="http://speechlab/facility.vxml">
              facilities information
         </choice>
     </menu>
 </vxml>
```

Example 4.4 VXML - <menu> & <choice>

A menu is essentially a simplified form with a single field, which offers choices to the user and transition to another dialog based on the user input. The example above is similar to the example 4.3. In *<menu>*, a *<choice>* is used to specify a speech grammar and/or a DTMF grammar that provide the input options to the user. A DTMF grammar can be defined by a *dtmf* attribute in *<choice>*. This example is another convenient way to make a choice and transitions to different destinations based on user's choice.

VoiceXML provides a number of ways managing flow control. For example:

```
<link event="help">
<link next=http://speechlab.uwindsor.ca/test.vxml"/>
<goto nextitem="login"/>
<submit next="/servlet/login"/>
<choice next="http://speechlab.uwindsor.ca/test.vxml">
<throw event="exit">
```

In this section, we have introduced the basic uses of VoiceXML elements and related algorithm etc. It is essential to know how to use VXML to build a voice application that features the ease of use of speech user interface design and fully take advantage of the powerful functionality of VoiceXML. Especially, it is important for the goal of transforming XML to VXML since VXML is the system output.

4.2.2 Features of VoiceXML

VoiceXML has gained a large industry support since its 1.0 release, and many are heavyweights such as Motorola, Oracle, Lucent and IBM. VoiceXML is an XML application because it adheres to the XML standard. This feature yields some important benefits. The most important is it allows the reuse and easy retooling of existing tools for creating, transforming, and parsing XML document. It also allows VoiceXML to make use of other complementary XML-based standards.

VoiceXML provide several important capabilities:

- The only standard way of developing a portable speech activated applications.

- Separates user interaction code from service logic. VoiceXML applications can use the same existing back-end business logic as their visual counterparts. Although new VoiceXML pages may need to be created, the underlying infrastructure, including database design, stored procedures, and CGI scripts can be reused with little or no modification. This brings the benefits of rapid development of VoiceXML applications.

- Implements a client/server paradigm. VXML minimizes client/server interactions by specifying multiple interactions per document. VoiceXML allows you to request documents and submit data to server scripts using URIs. VoiceXML documents can be static, or dynamically generated by CGI Scripts, JavaBeans, ASPs, JSPs, JavaServlets, or other server-side logic.

- Promotes service portability across implementation platforms. VoiceXML is a common language for content providers, tool providers, and platform providers.

- Supports networked and Web-based applications to enable rapid delivery of applications, which also means a user at one site can access information or an application provided by a server at another distant location.

4.2.3 limitations of VoiceXML

The VoiceXML is not intended for heavy computation, database operations, or legacy system operations. These limitations assumed to be handled by resources outside the interpreter of VoiceXML Browser.

4.3 Comparison with Speech Web

SpeechWeb[Frost1, 1999] provides natural-language speech access to large knowledge bases. Parts of SpeechWeb are developed using Java Speech API [SUN JSAPI] and executable attribute grammars [Frost, 1992]. Both SpeechWeb and VoiceXML are accessible via Web, and they provide natural-language speech interfaces, though comparing SpeechWeb to VoiceXML, it has some limitations.

- Consists of a collection of speech-accessible hyper-linked objects called sihlos. Sihlos are distributed over a network. Each sihlo contains speech-activated hyperlinks to other sihlos on the network. However, the developer for the SpeechWeb has the responsibility to locate and provide links to other sihlos. In contrast, VoiceXML provides several ways to handle links among grammars, which isolate the application logic from server logic.

- SpeechWeb allows only one grammar file is active at any time of the interaction process. On the other hand, VoiceXML specification allows more than one grammar that is active at the same time.

- SpeechWeb is initially difficult for some users to control the dialogue flow because users are not familiar with the questions that could be asked at the beginning. VoiceXML is easy to provide prompts to users about what can be said or asked in the dialogue flow.

- No reusable dialog components can be used in SpeechWeb. VoiceXML has mechanism for reusable dialog components.

- For SpeechWeb it is hard to modify implemented prototype to create and delivery of web-based, personalized interactive vice-response applications. VoiceXML is easy to use and can quickly voice-enable existing Web-based applications.

On the other hand, SpeechWeb enables the construction of very complex language processors that would be difficult to construct in VXML unless a related server-side processor is used in conjunction with VXML. Perhaps a useful approach when very complex language-processing capability is required would be to use VXML as a front-end speech-to-text interface, and to use W/AGE [Frost, 1994] (part of SpeechWeb for building complex language processors) as the back-end.

Chapter 5

XML TO VOICEXML TRANSLATION

5.1 Overview

XML is rapidly becoming a standard format for data exchange over the Internet. One of the most important reasons to use XML is because it separates data from the presentation, and allows us to focus on producing much more useful XML data. A generic architecture for translating these data to VoiceXML, with data access and data input functions, ease of use interface and customized specification, is introduced in this Chapter.

When the data that may represent some objects from an object-orientated program, or from a relational database, or being converted from other applications, is presented in an XML format, we might not know how that data will eventually be used. However, XML data provides a more potentially broad use to many different types of users. For example, if a web site produces its main interface using XML. It can use a transformation layer to represent itself as HTML for the traditional Web users, WML for mobile clients, or VXML for both telephony and computer-based voice application customers. Suppose the site is updated from time to time, this would usually create much workload to maintain the data consistent in each format. Fortunately, in this case only the underlying XML data need to be changed, and these XML data can be transformed into different presentation formats (other forms of XML).

It is also important to notice that this translation enables the applications to use their own representation of data, while being able to communicate with other applications, customers and partners in the format they prefer. Converting different XML applications or switching to/from other forms of data representation to enable information exchange between applications is a very common situation in business applications.

We know there is much need for speech as well as for translating between XML applications. So far, we have looked at the data structures that XML documents can encapsulate, and we have talked about how DTDs and XML Schemas can define those structures and contents. In addition, we have introduced the ease of use of VXML and how VXML use its components and control-flow mechanism to construct a speech application, and main differences between graphical user interface design and speech user interface design. The generic architecture introduced in this Chapter is a guideline to building VXML applications from XML documents. The diagram given below is an overview of the system design:

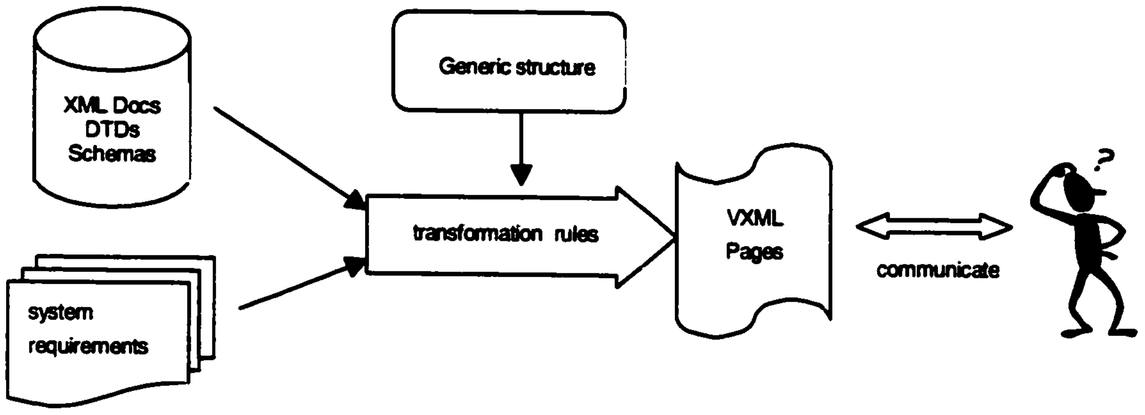


Figure 5.1 An Overview of the System Design

In this chapter, a generic architecture is presented. We will begin by introducing the expressions used in the transformation, giving the system requirements for the translation first. We will then look at the generic architecture – especially with respect to the issues of speech user interface and customized specifications, and next move on to look at the architecture of the generic architecture. In the next chapter we will focus on the structural and data type's transformation from XML documents to VXML documents based on DTDs and XML Schemas with two forms – data input and data access.

5.2 Notation Used in the Transformation

In the translation architecture and mapping algorithm described in this chapter and the next chapter, we use some notation to identify which data in the XML document is to be accessed. In order to do this and be consistent with existing work by other researchers, it was decided to use a notation. This notation is closely related to the well-defined expressions in XPath

[W3C XPath], and XQuery [W3C XQuery] that have already been developed by W3C for XML data access. The notation used in this thesis report ultimately may also facilitate the implementation of the translation algorithm as it may be possible to make use of tools already developed for XPath and XQuery. Therefore, principle notation and expressions used in this paper were developed by the author of this thesis report and were adopted and inspired from XPath and XQuery specification from W3C to locate a particular XML component in an XML tree structure. XPath is a notation for navigating along "paths" in an XML document, and is used in several XML-related applications including XSLT [W3C XSLT] XPointer [http://www.w3.org/]. XQuery is an XML query language that "provides flexible query facilities to extract data from real and virtual documents on the Web" [W3C XQuery].

In order to clearly explain notations used in the remaining of this paper, let us look at the example 2.2 courses.xml and its tree structure presented in figure 5.2 (next page). In this diagram, elements and attributes are considered as the children of their associated element. The circle denotes the root of the document, squares represent elements, and shadowed squares presents attributes. The element pointing to an element node or an attribute node is a branch. Other element nodes with a text content, and attribute nodes are leaf nodes.

Locate the node To locate a node that can be an element or an attribute node in the XML tree structure, we use the `'$'` followed by the path from the root element to this particular node delimited by `'.'` character. In the path expression, only the root can be omitted.

```
1 $root                          //the root element
2 $courses                       //the root element
if the current context node is courses, $element stands for element's name "courses"

3 $course[1].title               //title of the first course node
  the path can be simplified as: course1.title

4 $root.course[2].section[1]     //first section element of second
                                   course node of the root
5 $course[1].*                     //select all the children elements
                                   of the first course node
6 $courses.course1.id            //first course's id attribute
```

We use the number inside square brackets, e.g., *[1]* to indicate the position of the element with respect to its parent node, and this format will be used in the mapping algorithm later. In

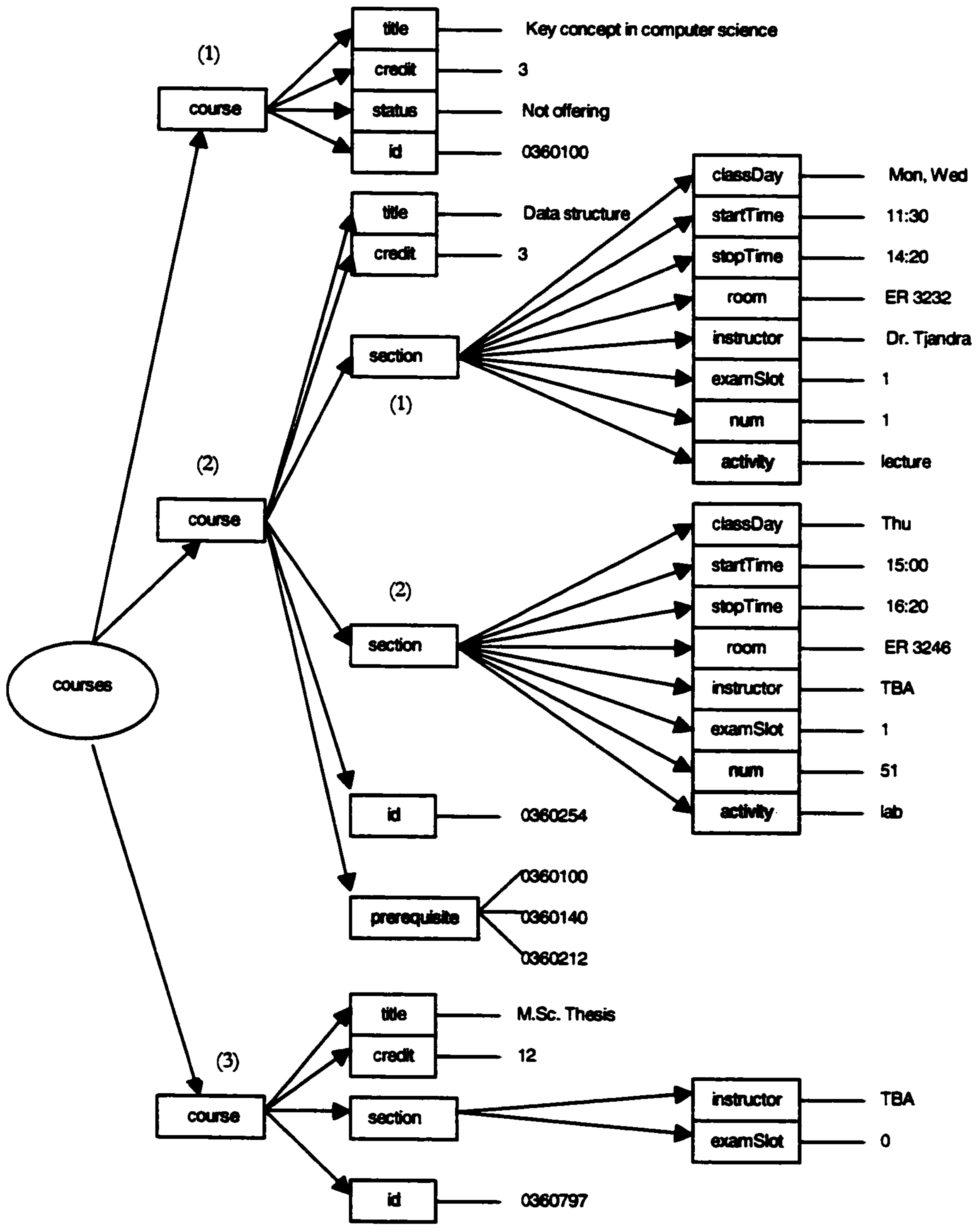


Figure 5.2 Tree structure for example 2.2 courses.xml

the next chapter, there are many sample VXML documents which use the path expressions to locate a particular element node, and in these samples, simplified path expressions, for example, `"course1.title"`, are used. Char `'*'` is used to denote all the subelement nodes under their parent node, such as shown in the sample 5 denotes the title, credit, and status node of the first course node.

Locate the attribute In the transformation process, we treat both elements and attributes the same way as they are the subelement nodes of their parent node. If for some reason we want to denote a particular node as an attribute rather than an element, for example, in figure 5.2 the id attribute is a child of course element, we use `@'` right before an attribute name.

```
1 $course[1].@id
  if the current context node is id, @attribute stands for the
  attribute's name "id"

2 $course[2].section[2].@num
3 $course[2].@*
4 $course[@id="0360254"] //locate course element with id=0360254.
```

Locate the contents Contents here refers to the leaf nodes for both elements and attributes. 'text()' is used to denote the content of the leaf node. E.g.:

```
1    $course[1].credit.text()       //credit of the first course node
2    $course[2].@id.text()          //the value of id of course 2
```

Miscellaneous Notations introduced here are going to be used in the translation process. For example, if we want to denote the number of courses the courses.xml contains, `'count()'` is used and inside the parenthesis is the target of the count. The 'query()' is used to denote a query pattern.

```
query($courses[1].title)    //return the title of course 1
count($course[2].section)//return the number of sections of course 2
count($course[@id="0360254"].section)  // return the number of
                                          sections of course 0360254.
```

Denote a set of children elements in sibling order using '[RANGE num to num]' format:

```
$root.course[RANGE 1 TO 2]
```

To denote all of the descendants we use double slash notation '//', for example, to express all the title element nodes in the document (two forms):

```
$root//title
$//title
```

5.3 System Requirements

Our goal is to make the outputted VXML as useful as possible in a general basis, so it is important and necessary for us to specify system requirements first. These design requirements should be stated clearly because they will have a great impact on the outputted VoiceXML structure.

- The system should provide a generic architecture to formalize the process of translating XML to VXML. The architecture should take care of the details in terms of speech user interface during the transformation.
- The system should provide the mechanism to navigate the XML.
- The system should be able to access the current node's direct parent and children in a sense of tree structure of an XML document.
- The system should be able to accept user's input to an XML document according to its DTD or Schema.

Above are application requirements for querying the information residing in XML documents, which indicates the possible routes that may occur during the user and computer interaction process. Some general requirements regarding the speech user interface design for better communicate between the human and computer are listed as follows:

- The system should provide directed dialog for user to proceed, which indicates that the application directs the user to make an utterance at most cases.
- The system should allow full-duplex (Barge-in) implementation
- The system should provide simple grammar for speech recognition. As we know that the VXML is going to be created from a very diversity of XML document, at this point, it is impossible to provide mixed initiative forms with natural command

grammars for the VXML application. However, the natural command grammars and mixed initiative form for natural language recognition can be achieved by user participating the specified design, expressing what fields and forms need to have customized design.

- Besides VoiceXML built-in commands (help, quite/cancel and repeat), the system should provide other 'always-active' commands to make it easy for user to navigate in the VXML application.

These system-design requirements are related to the issues: what kind of information (or queries) can be answered in terms of complexity of queries. They involve XML nodes, attributes, and position. The user-defined specification deals with operations similar to XQuery Language, such as filtering, group, sort, join, select etc., which is very much like SQL for relational databases.

5.4 Generic Architecture of Translating XML to VXML (GATXV)

To design and build the generic translating architecture is actually a process of developing a speech user interface. This process requires an iterative process of four phases for a specified task required by an application client. According to the system requirements stated in 5.3, we will express the features included in the generic architecture and taken them into account to design such an XML to VXML translating architecture in this section.

The generic architecture is described from three different points of view. First, the system is divided into data access and data input applications. Second, the system provides ease of use speech interface in many ways such as using directed dialog, providing the introduction messages, providing handy commands etc. Third, the system uses a helper file that is an XML conformed document. This helper file is a customized specification to enrich the functional queries that the system can have to help people get the information they want.

5.4.1 Data Access and Data Input

According to the system requirements, the focus of our work has been to develop a generic architecture and algorithms. The architecture formalizes the transformation of XML to VXML. The algorithms, terminating and deterministic rules, are used to translate XML to VXML with two kinds of applications: *data access applications* and *data input applications.*

A *data access application* is based on both XML structure (DTD or schema) and XML instances. The DTD or schema is used to build the skeleton of VXML document, and XML instances are used to obtain the specific information for voice responses. Data access applications allow users to get into the value of attributes and content of elements. Data access applications provide us to access a wealth of information through the voice.

A *data input application* is mainly based on XML DTD or schema, so we can obtain and analyze the structure of XML documents that the user to going to create through speech. According to the structure, a VoiceXML document can be created for gathering information. Furthermore, the gathered information can be stored in the database, presented in a Web browser, or viewed in other formats such as PDF (Portable Document Format) or through XML after translation using the process above. For instance, such a data input application can be a VoiceXML document functioned a voice inputting course information system. This system is generated to serve collecting data for all the courses offered in a semester that conform the same structure, which is defined by a DTD or schema file.

5.4.2 Speech User Interface

We are trying to improve the generic architecture's usefulness and friendliness from the following aspects: *grammar preparation*, *introductory messages*, *body*, and *ending message* of the architecture.

5.4.2.1 Grammar preparation

The preparation section is an essential part of the application process, which mainly generates and obtains (if applicable) all the necessary *grammars* for the use of the application. According to the system requirements, the grammars for the speech recognition engine to recognize the user's utterance are listed as follows:

1. A grammar for a collection of all the elements and element's attributes: eleAttr.gram
2. A grammar for a collection of always-active command list: command.gram
3. A grammar for random accessing XML documents by speaking a node's path: access.gram, which combines the eleAttr.gram with VXML built-in grammar *number*.
4. A grammar for a huge collection of domain-related vocabularies for inputting the contents of elements and values of attributes: dictation.gram
5. A grammar for the user specified query pattern: queryRule.gram.

- A grammar that could be automatically generated from the DTD or XML Schema, which is a collection of the element names and element's attribute names. Let's take a close look at the example 2.2 (`course.xml`), for example. The `eleAttr.gram` is the combination with all the elements and attributes of the document. This grammar file includes three rule definitions: <eleAttr>, <element> and <attribute>. In the rule <element>, we take each element's name as a token, between them is "|" a vertical bar which mean alternatives. Similar to <element>, in <attribute> rule, we take each attribute's name with its related element's name together with the form "`[element's] attribute`" as a token with the "|" in between, and "`element's`" is optional. In this form, the '`element`' stands for the name of the element, and '`attribute`' stands for the name of the attribute. The rule <eleAttr> is defined as a public rule that means this rule can be imported into another grammar, and the rule <eleAttr> is in the form of `<eleAttr> = <element> / <attribute>`. The grammar is possible in the following form:

```
// This is the eleAttr.gram for courses.xml (example 2.2)

#JSGF V1.0;
grammar eleAttr;

   <element> = courses | course |title |credit | status |
            section | classDay | startTime | stopTime |
            instructor | examSlot

   <attribute> = [course's] id {id}|
              [course's] prerequisite {prerequisite}|
              [section's] num {num}|
              [section's] activity {activity}

```

```
15      public <eleAttr> = <element> | <attribute>
```

Example 5.1 Grammar - eleAttr.gram

The only note to the example 5.1 is that from line 10 to 13, for each attribute we use an action tag – curly braces to indicate the attribute's name [Lucas, 1999]. In this way user can say, for example, *course's id* or *id*, no matter which one the system will take "*id*" as the input for this attribute.

- A grammar for a collection of always-active command list named *command.gram*. [IBM 2000] has a list of recommended always-active commands. Because we are developing a generic architecture for translating XML to VXML for data input and data access, we are considering two sets of commands for help, and they are listed in the figure 5.3.

The "direction" commands are used to find a way in the document. For example, "backup" goes to the previous form item prompt, "list commands" lists all always-active commands. On the other hand, the "navigation" commands are used to find elements and attributes in an XML tree structural view. For example, "parent" goes to

Command Name	Always-active commands list
Direction	Backup, exit, help, list commands, repeat, start over, what can I say now
Navigation	Parent, first child, last child, previous, next, attribute, next attribute, current

the parent node of the current context element.

Figure 5.3 Always-active commands list

The function of "navigation" commands can be seen as part of DOM [W3C DOM] methods, but the difference is in our scenario it is a voice call compared to DOM where it is a program call. The *command.gram* is given as follows:

```
        // This is the command.gram
        #JSGF V1.0;
```

```
grammar command;

<direction> = backup | exit | help | list commands |
              repeat | start over | what can I say now
<navigation> = parent | first child | last child |previous |
               next | attribute | next attribute |current

public <command> = <direction> | <navigation>
```

Example 5.2 Grammar - command.gram

- A grammar for random accessing XML documents by speaking a node's path: access.gram. In the body of the application, the system needs to exchange the information with the human via voice. It is essential to define what kind of utterances that can be recognized by the computer. In data access applications, some specific XML document instances are required to which we provide several different ways to access. One of them allows the user to speak a node's path at any given point in the application to access the XML data, so we call this method as "random access", and the grammar file engaged with it is "access.gram". In section 5.2, we introduced the notation to locate an element or an attribute from which we can see that the path of a node consists a set of names of elements, attributes and some position numbers. These numbers give the location of the elements in the context of their parent. For example (figure 5.2):

```
Path:                                  vocabularies for grammar:

(1)courses.course2.title           //courses, course, 2, title
(2)courses.course2.section1.@num //courses, course, 2, setion, 1, num
```

When the user said: "courses, course2, title", the system should be able to response to "the title of this course is Data Structure". Therefore, we need to construct a generic grammar, which will be able to accept any available paths of a specific XML instance. This grammar is the combination of the "eleAttr.gram" and VXML built-in grammar *number*. In the following sample, we show a specific example "courses.course2.title" to illustrate how to construct such a grammar. In order to present it more clearly, we limit the range of a number within 1 to 9. In this example, we use action tag [Lucas, 1999] to get the actual path string we needed for retrieving the request XML data in data access applications:

```
// This is the a specific example for access.gram: path.gram

#JSGF V1.0;
grammar path;


<num> = 1 { this.$value="1";}
      | 2 { this.$value="2";}
      | 3 { this.$value="3";}
      | 4 { this.$value="4";}
      | 5 { this.$value="5";}
      | 6 { this.$value="6";}
      | 7 { this.$value="7";}
      | 8 { this.$value="8";}
      | 9 { this.$value="9";}
      | 0 { this.$value="0";};

<path> = courses {this.$value="courses"}
         course 2{this.$value=this.$value+".course" + $num;}
         title {this.$value=this.$value+ ".title";};
```

Example 5.3 A specific example of random access grammar: path.gram

- A grammar for the huge collection of domain-related values vocabularies for inputting the contents of elements and values of attributes, which is used for the data input application. Such a grammar, for instance, every city and province name grammar can be obtained from some companies. The dictation grammars are inexpensive at current market and the speech recognition engines in recent years claim to achieve accuracy of 95% and more. The bottom line is we can always construct such grammars by ourselves.
- The user specified query pattern grammar: queryRule.gram will be introduced in 5.4.3.

5.4.2.2 Introductory Messages

At the beginning of our generic architecture, there is always an introductory section, which includes the messages when the user starts the application [IBM, 2000]. These messages usually contain three parts whose function and example are listed in the table below:

Name	Function	Example
Welcome	May include subject, function introduction	Welcome to the automated course information query system
Always-Active List	Give user the hint what can say, what commands are always available	You can always say "repeat" to previous prompt, or "exit" to quit
Initial Prompt	The first prompt to user asking user input	Please say a course number to start, for example: 0360100

Figure 5.4 Generic Architecture Introduction Part

The introductory section has an important role during the interactive process between the human and the computer, and it reminds the user the application's subject and functions. A good set of introduction messages will give the application a good start.

5.4.2.3 Body of the Generic Architecture

The body of the generic architecture is the main section where all the conversations, exchanging information are mixed together. In this section, proper prompts in terms of application logic and context should be sent to the user at an appropriate time with a clear synthesized speech, which means at least the following things need to be considered:

1. Design consistent prompts pattern for every requested user input
2. Get the destinations for each always-active command
3. Mark up the synthesized speech output for better understanding

Now, we give a detailed explanation and solutions for the above concerns in body section:

- Design consistent prompts pattern for each requested user's input. The goal is to clearly indicate to users what they can say at any given point besides always-active commands, which is applicable for both data access and data input applications. For instance, in a data input application we want the user to input the course's id in a <field>, the code might like this:

```
1    <field name="course.id" type="digit">
2       <prompt> Please say the course's id </prompt>
3       <prompt count="2"> Course's id, please? </prompt>
4       <help> course's id is a 7 digit number </help>
5       <noinput> I could not hear you.
6        <reprompt/>
7       </noinput>
8    </field>
```

Example 5.4 Prompt Pattern

For one user input, we use <prompt> <reprompt> with <help> and <noinput> event to remind the users what they can say. The <prompt> that will be played to response to user input is controlled by the Form Interpreter Algorithm (FIA) of voice browser. In the above example after the first system prompt, if there is no input from the user, <noinput> event (line 7) will be caught. When the <reprompt/> is executed, the FIA will increase the <prompt> counter, so the second <prompt> will be played. The typical conversation for above prompts pattern would be:

```
Computer:    Please say the course's id
User:        (silence)
Computer:    I could not hear you. Course's id, please?
User:        help
Computer:    course's id is a 7-digit number
User:        0360100
```

- Analyze and get the command *destination* for each always-active command [IBM, 2000]. In order to explain the command's meaning and destination, let's look at the example 2.2 and its tree structure presented in figure 5.2. In the tree structure, we have the root of the document – in circle, element – in square, and attribute – in shadowed square, the branches and leaves. Suppose we are at the *second course node* of the example 2.2 whose course id is 0360254, at this point, the destinations of each always-active commands are shown in figure 5.5.

Command	Meaning	Explanation (e.g.)	Destination
backup	Go back to previous <form> item or <menu>	Assume now system has a menu prompt asking for user input, previous <form> item is the first course node's id attribute	$course[1].@id
exit	To exit the system	Also see the section "Ending of the application"	Ending message, and Quit
help	Catch <help> event	Go to associated <help> within current prompt pattern	<help> say title, credit, or section </help>
repeat	Repeat the prompt associated with current form item, menu	Repeat the menu prompt asking for user input	$course[2] menu prompt
Start over	Go to the very first dialog	Restart the whole dialogs of the document	Welcome message
parent	Go to parent node	Now at course[2], whose parent is courses	$courses
First child	Go to first child node	Order: subelements first and attribute followed	$course[2].title
Last child	Go to the last child node	Directed children only (elements, attributes)	$course[2].prerequisite
previous	Go to previous sibling node	If the node is a branch, the destination is the node itself. If the node is a leaf, the destination goes to the content or value of that leaf (either a element or an attribute)	$course[1]
next	Go to the next sibling node		$course[3]
attribute	Go to the first attribute	Same to commands: *parent, first child and last child*	$course[2].@id
Next attribute	Go to the next attribute		$course[2].@prerequisite
current	Return current node name	Either an element node or an attribute node	$course[2]
what can I say now	What the user can say at a given point in the interaction	If, e.g., the current note is courses, there would not have a *parent, previous, next, attribute, next attribute* command available	All the commands listed above and command "*list commands*".
List commands	List all the always-active commands to user	All the commands listed above plus itself	

Figure 5.5 Directions and destinations

- Mark up the synthesized speech output for better understanding of the synthesized speech [SUN JSML]. Although TTS cannot mimic the complete naturalness of human speech, the TTS markup tags can improve the speech quality in various ways. For example, to improve the TTS engine's structural analysis by using <div> tag, to improve how a word or phrase is spoken by using <sayas> tag, etc. We consider to use speech markup tags <div> <sayas> <break> and <emp> in the following way:

Markup tag	Meaning	Example

<div>	Identify the enclosed text as sentence or paragraph	`<prompt>`
		`    <div> the course id is <break size="medium" />`
		`       <emp> <sayas class="digit">0360100</sayas> </emp>`
		`    </div>`
		
		`    <break size="medium" />`
		
		`    <div> the course title is <break size="medium"/>`
		`        <emp>key concept in computer science</emp>`
		`    </div>`
		`</prompt>`

<sayas>	Specify how a word or phrase is spoken	
<break>	Specify a pause in the speech output	
<emp>	Specify emphasized text	

Figure 5.6 Speech Markup

Use *<div>* tag to specify each sentence. Use the *<break>* in two ways: (1) separate the main contents that are actually picked up in the XML document from others in a sentence. (2) place it in between sentences. Use the *<emp>* tags to emphasis the content of the element or value of attribute. Use the *<sayas>* if we can identify the type of text, in between open*<emp>* and close*</emp>* tag, whose type is one of VXML built-in grammar types, for instance, *"digit"* in above example.

Besides using the preparation grammars, introductory messages, created necessary grammars, and speech markups, it is essential to create reusable dialog components and use them as often as possible in the voice-related applications. It is important for modularity, portability, and high performance to the system, especially in VoiceXML applications. In our generic architecture, for example, the grammars, built-in field-types, and some help commands can always be reused.

5.4.2.4 Ending Message of the Application

Compare this part with body of the application, it is very simple and easy to handle. Users can use "exit" command form always-active commands to quit the application at their desire. In data input application, we have another way to exit the application by confirming not to continue inputting data when the system requires an answer from the user. A typical ending message, for example, can be something like " thanks for using automated course information input system, goodbye".

5.4.3 Customized Specification

Since the generic architecture is used to handle the general cases of translating XML to VXML, the default structure and XVMA generated VXML might not be sufficient for all user's needs. The special needs from the user is always coming down to the ground no matter how much you pay attention to designing the speech user interface, or how careful you handle every detail in the mapping process. To solve this problem, we can always modify the generated VXML directly because it is easy to learn and easy to use. However, this would add some burden to the end users, and it would limit the system's functionality. Therefore, we create a queryRule schema in the generic architecture that may be extended by the user. Any XVMA implementation should provide the necessary code to accept and process such a query definition. At the most basic level, a queryRule document must conform to the DTD shown in the example 5.7.

In this section, we will introduce the customized specification, defining query rules using specification, and the format of specification file. Before we go to detail of any user specified query rules, let us look at what kind of questions cannot be handled without such specification. For example, in data access application can we ask the questions like: (for courses.xml):

```
Q1.  What is the first child of the second course node?
Q2.  How many courses and sections does this document have?
Q3.  How many sections does course 0360254 have?
Q4.  Answer the title and credit of the course 0360797?
```

So far, to examine the above questions against our generic architecture, the answer is that we can not handle these questions. One of the reasons for that is that even through we can use appropriate XML API or various XSL processors to compute and get the answer for each question, we do not have available grammars to recognize these sentences. Moreover, there is no way to recognize the semantics of each sentence. For instance, in the question Q3, how can we let the recognizer know that the "`sections`" here means the `section` element; *6010254* stands for one of the attributes `id`'s value for the `course` element; and "*how many*" stands for the count we need? We have to deal with those "unusual questions" while we are trying to build any voice applications.

Question one, however, we could access first child of the second course node indirectly by through the path courses →course 2 → child 1, so we will not consider it in user specified rules. Questions like Q2 and Q3 are to compute the count of appearances of a particular element under the document root or a node with its ID type attribute equals to a specified value. Question Q4 is to request the content of an element searched by an attribute value with type ID of an element. In our customized specifications, we only allow (1) to identify a unique node. This node can be the document element (root) or a node with its one of unique children. For example, in Q3 and Q4, id attribute is used to identify the second course node and the third course node shown in figure 5.2. (2) To request the number of count of an element under the identified node; (3) to request the contents of elements, values of attributes under the identified node.

Each XML document can be given a customized specification, called *queryRule.xml*, an XML conformed document, in which user-specified query rules is specified. Let us look at the example 5.5 in which four query-rules including a default query are defined.

```
<?xml version="1.0"?>
<!DOCTYPE queryRule SYSTEM "queryRule.dtd">

<queryRule xmlSource="courses.xml">
  <query id="1">
   <queryNode keyItem="root" />
   <countItem item="course section" />
  </query>

  <query id="2">
   <queryNode keyItem="course.id" condition="0360254" />
   <countItem item="section" />
  </query>

  <query id="3">
   <queryNode keyItem="course.id" condition="0360797" />
   <queryItem item="title credit" />
  </query>

  <query id="default">
   <queryNode keyItem="course.id" />
   <queryItem item="all" />
  </query>
</queryRule>
```

Example 5.5 queryRule.xml for courses.xml

The second line above indicates that the above XML document must conform to the structure and constraints defined in *queryRule.dtd* that will be introduced later in this section.

The line 4 is the root element `<queryRule>` with its attribute `xmlSource` that specifies to which XML document instance this rule specification related. From line 5 to 18 are three query rules defined in `<query>` element with `id` attribute to identify the order of the queries.

The default query rule (line 20 to 23) is used to indicate that we can directly access a node and its children by specifying its unique ID type attribute. The default query rule is only for the highest level node with an ID type attribute in the XML tree structure. The example above shows that the user can query each course element's leaf children by specifying the course's id attribute. For example, in the figure 5.2, if we search by `id='0360254'`, the system should provide the node `course2`'s title, credit, etc. If the user does not provide the queryRule.xml for courses.xml, the default rule should still be generated automatically by the system and be implemented in the data access applications. The example of default query rule will be illustrated in Chapter 7, and the algorithm will be presented in Chapter 9.

Now, let us look at the three queries specified in the example above. In the first query, `queryNode`'s `keyItem` is `"root"`, which is the root element and the only one, so we call it keyItem, and it is used to express "query in this document". The query one has a `<countItem>` whose attribute item includes two elements name: `course` and `section` that are descendent of root element `<courses>`. The `<countItem>` means to count the occurrences of each element included in the `item` list. Therefore, the first query basically wants to know:

```
    "How many course elements and section elements does this document have?"
Q2: How many courses and sections does this document have?
```

This is the Q2 we faced before where we could not handle it in default system design. Similar to the first query, the second and third queries use `keyItem` as `"course.id"` whose data type is `"ID"` which is guaranteed to be unique in courses.xml, and condition attribute specify the value of the `id`. In query three, instead `<countItem>`, a `<queryItem>` is used to express that this query rule is to query the contents of elements listed in `item` attribute rather than to compute the count. So these two queries specified in queryRule.xml are trying to ask:

```
    "How many section elements in course node with id equals to 0360254?"
    "What are the title and credit of the course with id equals to 0360797?"
Q3.  How many sections does course 0360254 have?
Q4.  Answer the title and credit of the course 0360797?
```

As we can see from above examples, those rules specified in queryRule.xml are exactly the questions we wanted to ask before. By specifying the query rule in this way, we can depend on the elements, attributes and their value in the specification document to create the query patterns for these three queries:

```
Query1: count($root//course, section)
Query2: count($course[@id="0360254"].section)
Query3: query($course[@id="0360797"].title, credit)
```

Example 5.6 Query patterns

The expressions used in the query patterns have been introduced in 5.4. Knowing the semantic of query pattern, we can code to get the result of each query. Furthermore, from the query patterns we can translate them into correspondent grammar files needed for the purpose of speech recognition. By having the query result and the grammar files we can start coding the VXML fragment for each query pattern and adding it to our existing prototype application. The possible grammar for query two can be constructed as follows:

```
  How many section [s]|[elements]
  does course [element][with id [attribute [equals to ]]0360254 have?
| [are]in[the]course [element][with id[attribute[equals to]]0360254?
```

From the process described above, we can see that a query pattern is associated with *three* phases. First phase is an implementation of query pattern called *queryPatternImpl* phase. Second phase is a translation of grammar called *grammarTranslator* phase. The third phase is a coding of VoiceXML to combine query pattern results and grammars to feature the voice accessible queries to XML document called *generateVxml* phase. These three phases cooperate to obtain the functionality specified in customized specification. The customized specification queryRule.xml must obey the following DTD:

```
<!ELEMENT queryRule (query)*>

<!ELEMENT query(queryNode, (countItem | queryItem ))>

```

```
<!ELEMENT queryNode EMPTY>
<!ATTLIST queryNode
      keyItem    CDATA    #REQUIRED
      condition  CDATA    #IMPLIED>

<!ELEMENT countItem EMPTY >
<!ATTLIST countItem item NMTOKENS #REQUIRED>

<!ELEMENT queryItem EMPTY >
<!ATTLIST queryItem item NMTOKENS #REQUIRED>

<!ATTLIST query id ID #REQUIRED>

<!ATTLIST queryRule xmlSource  CDATA #REQUIRED>
```

Example 5.7 Customized specification: queryRule.dtd

One interesting thing is that based on the above DTD file, we can build a voice input application using generic architecture and algorithms presented in this work. With this voice-input application, we can feature voice inputting query rules if we have appropriate grammars for a specified domain.

In queryNode, we use "*keyItem*", the purpose is to guarantee that the *queryNode* can be easily and uniquely located in an XML document. This makes the query and implementation of query pattern to be relatively very simple, however, it limits the query patterns available for a specified XML document. In our example, the "*keyItem*" is either a root element or a node with the type ID attribute, which are unique in the DTD and XML Schema. However, XML Schemas allow the uniqueness to be elements, attributes or their combinations. At the current stage, we limit the *keyItem* to be either an attribute or an element, no combinations are allowed in the customized specification. In addition, "*keyItem*" requires the full path (the root element can be omitted) of the element or attribute in the form from the root to the branches and to the leaf delimited by dot ".", such as in the query 2 and 3, "*course.id*", "*courses.course.id*", or "*root.course.id*" are presented the same key item.

Any other kinds of query cannot be handled so far such as: "list all course's title containing the word 'computer' ", or "courses' id does not begin with '0360' ". However, we could always extend the existing specification by allowing more elements in queryRule.xml to accept more query patterns. The focus of this work is to translate XML to VXML under the general system requirements rather than to handle how to query XML documents. XQuery – W3C's

XML query language [W3C XQuery], XQL[Robie, 1999] and XML-QL [Deutsch, 1999] are dealing with XML querying.

5.4.4 Architecture of GATXV

We have introduced the generic architecture – especially with respect to the issues of the functionality, speech user interface and customized specifications. To recap the points of the generic architecture, from the examples given above we summarize the generic architecture in the following diagram.

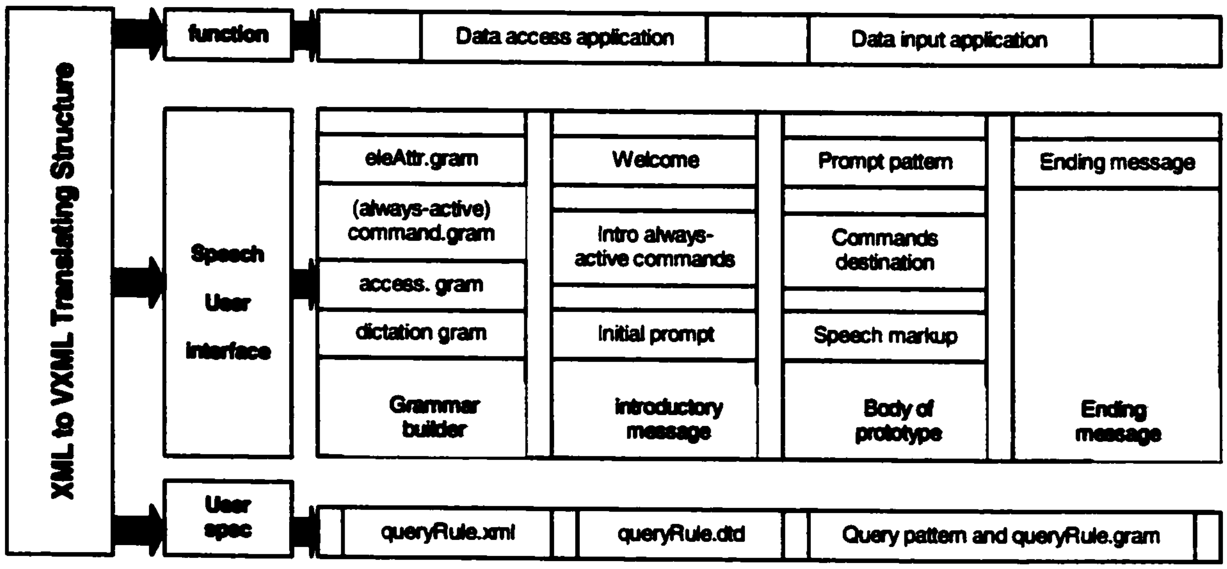


Figure 5.7 GATXV Architecture

The Generic architecture of Translating XML to VXML shows that when we are trying to transform XML documents to VXML documents, system operations, speech user interface and user specified query rules should be considered. In this Chapter, we emphasized the last two issues, which is more close to the speech-related topic than structure mapping. In the following chapters, we will explore the first aspect about methods of constructing data access and data input applications from XML specifications.

Chapter 6

ANALYSIS OF RELATIONSHIP BETWEEN XML AND VXML

Starting from this chapter, we will focus on the structural transformation from XML documents to VXML documents based on DTDs and XML Schemas, as well as mappings from XML Schema data types to VXML built-in grammars. This translation is depicted with two forms – data input applications and data access applications, these two forms should follow the system requirements described in 5.3. Moreover, the transformation process should cooperate with the generic architecture by adopting the structure from the welcome message to the ending message, as well as grammars, prompt patterns, speech markup, etc. The goal of this process is not only to make the transformed VXML application perform the functions required, but also to improve the ease of use of speech interface.

We will investigate XML, VXML and the relationship between them in this chapter. Following it we will look at the possible mapping approaches that can be applied in both data input and access applications in Chapter 7. Then we move on to look at mapping from XML Schema data types to VXML built-in grammars in Chapter 8. We present the XML to VoiceXML Mapping Algorithm (XVMA) for data access and data input applications in Chapter 9.

6.1 Analysis of XML Documents

XML Information Set Specification [http://www.w3.org/TR/xml-infoset/] defines eleven different types of information items that may exist in an XML document:

- The Document Information Item.
- Element Information Items.
- Attribute Information Items.
- Processing Instruction Information Items.

- Unexpanded Entity Reference Information Items.
- Character Information Items.
- Comment Information Items.
- The Document Type Declaration Information Item.
- Unparsed Entity Information Items.
- Notation Information Items.
- Namespace Information Items.

We have seen some of the above information items in Chapter 2 and their use in an XML context. Among these information items, the document, element, attribute and character information items are the most important components to represent XML data in an XML document. The others are most often optional. However, some of them are essential in certain cases. For instance, when an XML document instance needs a DTD to be validated by an XML parser, the DOCTYPE declaration (the document type declaration information item inside an XML instance) becomes required. The guideline of handling these information sets during translation from XML to VXML are given below:

- The Document Information Items – since VXML is an XML application, this item will keep the same.
- Element Information Items and Attribute Information Items – see Chapter 7. The attributes are treated with no difference to elements in the mapping process.
- Processing Instruction Information Items and Comment Information Items – ignored in the mapping.
- Unexpanded Entity Reference Information Items – items will be expanded in document's content and attribute's value, so they are not a problem in the mapping.
- Character Information Items – see Chapter 7.
- The Document Type Declaration Information Items – in the initialization phase, these items may be used to get DTDs for a specified XML documents.
- Unparsed Entity Information Items and Notation Information Items – an unparsed entity is only allowed to appear as a token in the value of an attribute of declared type ENTITY or ENTITIES, and with its associated notation which is used to locate a

helper application capable of processing data in the given notation. So they are not considered in the mapping.

- Namespace Information Items – it is assumed that the elements and attributes are in the default namespace.

We can accept three options in the input file: an XML document instance, a DTD corresponding to this instance, or an XML Schema corresponding to this document instance. From the previous examples, we can see that the structure of an XML document has an important role in the transformation process. The reason we should allow three choices of the input file is in that from any of them we can build the same tree structure. The simplest one to accurately reflect such a structure is the DTD from which the tree is constructed as a JSGF grammar rule is expanded until every branch of the tree reaches its leaf node. Examples of such tree structures are given in figure 2.2, and figure 5.2. In the next section we will introduce the transformation based on DTDs.

The XML family consists of a large amount of branches. See the following diagram:

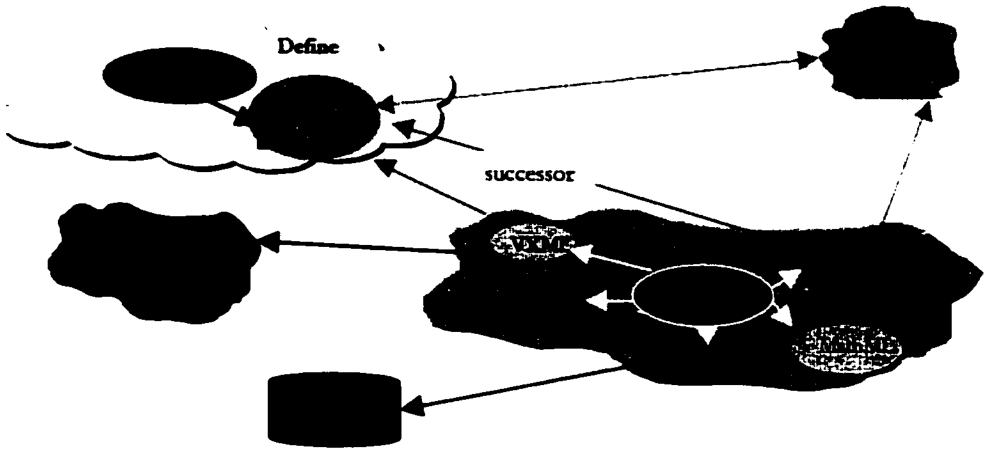


Figure 6.1 XML-based Applications

As we introduced in Chapter 2, XML and SGML are meta-markup languages. XHTML, WML and VXML are actually XML-defined applications. XML itself can be seen as an abstract base language whereas e.g. VXML is a specialized form of XML with limited vocabularies defined by XML. These XML applications through different interfaces present data to different type of users. For instance, VXML through voice browser to communicate with users in a speech basis, XHTML and HTML through Web browsers to represent data to traditional Web clients, and DBML through e.g. Oracle database to express data relations to database users. Therefore, XHTML, WML VXML, MathML are domain specific XML applications, each of which has a set of domain specific vocabularies. For example, VXML documents must conform to the VXML DTD and all the legal elements and attributes for certain element are fixed, so the user cannot add their own tags to those XML-based applications.

In this paper, XML documents stand the general XML documents that are at the center of XML applications in figure 6.1. For example, book records for a bookstore, employee records for a university, purchase orders for an e-commerce company, yellow page information for a city, etc.

6.2 Investigation of VoiceXML DTD

Using VoiceXML to represent XML document content is not as easy as it looks. The reason is that VoiceXML, though it is an XML application, defines its own mechanisms to enable dialogs between human and computer. Any VoiceXML document must conform to the VoiceXML specification in terms of the use of tags, grammars, audio file format and so on.

VoiceXML is not just about the storage of data in a particular XML format, as a matter of fact, it is used to created speech-based applications. VXML is designed to help the quick development and deployment of human-computer dialog systems. "A VoiceXML document (or a set of documents called an applications) forms a conversational finite state machine. The user is always in one conversational state, or dialog, at a time" [VxmlOrg, 2000]. The core of a VoiceXML document is the dialog and our goal is collecting and querying of data, which is mainly achieved by the VoiceXML dialog mechanisms: *forms* and *menus*.

Within the VoiceXML implementation, the form interpretation algorithm (FIA) is built, which drives the interaction between the user and a VoiceXML form or menu. A menu can be viewed as a form containing a single field whose grammar and whose <filled> action are constructed from the <choice> elements. The FIA must handle: form initialization, prompting, grammar activation and deactivation, leaving the current form, processing multiple field fills from one utterance, selecting the next form item to visit, and choosing the correct event handlers etc. For details of FIA, please refer to the VoiceXML 1.0 specification.

VoiceXML DTD files have definitions about: a set of entity declarations, root, dialogs, prompts, fields, event, audio input/output, call control and control flow definitions. Except for some necessary components used for describing the document structure, the other elements in a VoiceXML DTD are closely related to dialogs (such as the elements with which the dialog structure is built). Elements handle the user's response including no input or no match with the grammar, and some action elements with which a dialog can perform, etc. The challenge is how to apply these dialog-related elements to match on the diverse content and structure in a huge collection of XML documents.

From the VXML DTD, we can see that the top dialog containers for a VXML application are <form> and <menu>. Forms are key component of VoiceXML documents, and they specify a dialog for presenting and gathering information. A form contains a set of form items, which include field items and control items; declarations of non-field item variables; event handlers and filled actions. These items are blocks of procedural logic that execute when certain combinations of field items are filled in. The aim of two methods of transformation from XML to VXML is trying to fully make use of these two items combining with prompts and grammars to provide an easy to use speech interface. The main task of our work can be illustrated by the figure 6.2 as follows:

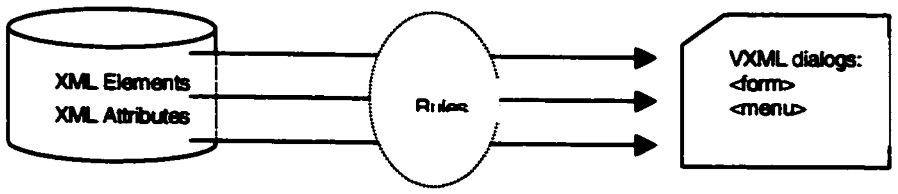


Figure 6.2 The Main Task of the System

Chapter 7

TRANSFORMATION BASED ON DTDS

In this chapter, the mapping from XML to VXML based on DTDs will be introduced in great detail through examples. For data access application, the mapping process needs the help of document instances to complete the transformation and it is eligible to use all the always-active commands. On the other hand, data input applications are limited to use only some direction commands.

7.1 Examples of Data Input Applications

For simplicity, we will not apply the VXML examples to the generic architecture introduced in Chapter 5. For example, there is no welcome, ending messages showing in the samples, and we do not follow the prompt patterns for each form item, etc. The generic architecture components should show up, however, at the formal version of the transformation results. We only show the skeleton mappings that are necessary to express the functions – input, and access to the information. Some other matters, for instance, to submit the collected data input to a server to redistribute to other sources or databases, to apply the customized specification to query document in data access applications are not shown in the samples as well.

7.1.1 Basic Mappings

To illustrate the data-input mapping, it is easiest to first look at a simple example. The example is from the Chapter 2 (example 2.1):

```
<?xml version="1.0" ?>
<!DOCTYPE courses1 SYSTEM "courses1.dtd">
<courses>
  <course id="0360100">
     <title> Key concepts in computer science </title>
     <credit> 3 </credit>
  </course>
  <course id="0360797">
    <title> M.Sc. Thesis </title>
    <credit> 12 </credit>
  </course>
</courses>                    "courses1.xml"
```

```
<!ELEMENT courses ( course* ) >
<!ELEMENT course( title, credit ) >
<!ELEMENT title ( #PCDATA ) >
<!ELEMENT credit ( #PCDATA ) >
<!ATTLIST course id ID "REQUIRED">


                    "courses1.dtd"
```

Example 7.1 courses1.xml and courses1.dtd

There is an obvious mapping to the VXML document shown as follows:

```
<?xml version="1.0">
<vxml version="1.0">
 <form id="course">
    <block><prompt>You are going to voice input..</prompt></block>
    <field name="title">
       <prompt> what's is title? </prompt>
       <grammar src="title.gram type="application/x-jsgf" />
    </field>
    <field name="credit">
       <prompt> credit? </prompt>
       <grammar src="credit.gram" type="application/x-jsgf" />
    </field>
    <field name="id">
       <prompt> what's the id? </prompt>
       <grammar src="credit.gram" type="application/x-jsgf" />
    </field>
 </form>
</vxml>
```

Example 7.2 Data input mapping

Example 7.2 is a very simple VXML application in which a dialog <form> includes three <field> items and three grammar files. These three grammar files belong to the dictation of the grammar builder in the generic architecture. According the system requirements, data input applications should be built to accept the user input repeatedly, and this can be done by clearing the form item variables, the FIA will automatically select the dialog again. Another way is to use a <goto> element. From this example, a simple tree structure to describe the dialog flow can be constructed as follows:

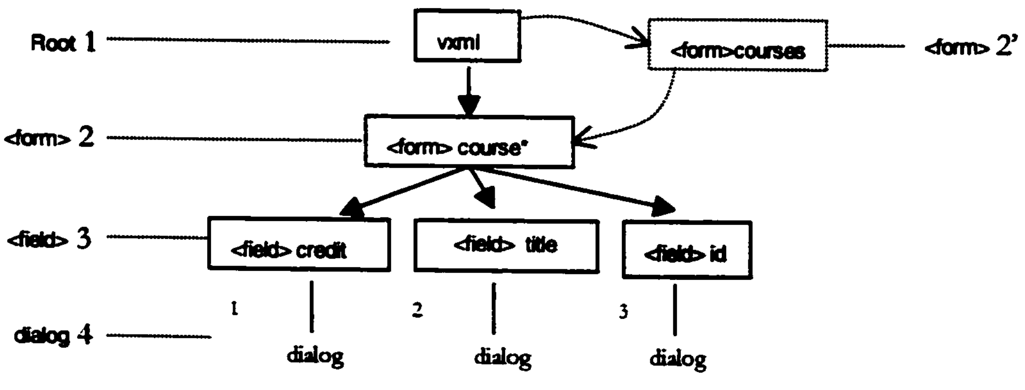


Figure 7.3 Input data dialog tree structure

This tree has a four-level hierarchy. The first level is the root element <vxml>, the second level a <form> named `course`, the third level are <filed> items with their name labeled at the nodes, and fourth level are the dialogs for collecting the user input. One note for the dialog tree is that in this special case, the document root element <courses> has only one repeated child <course>, so we can omit it from the data input VXML file, thus <courses> is around by the dash-dot line. However, if we added <courses> node, it would be represented by a <form> in VXML, and inside <form> would be either a <subdialog> or a <field> with a <goto> element (see mapping sequence for detail 7.1.2).

From the example above, we can see that essential vocabularies used in the example 7.2 VXML file such as <form> `id`, <field> `name` we can get from the DTD file we listed in example 7.1. The dialog tree structure is identical to the DTD tree structure shown below:

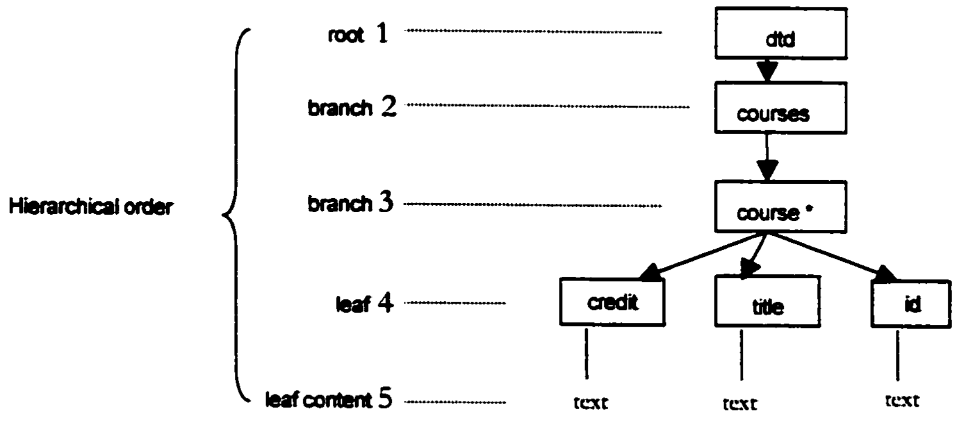


Figure 7.4 DTD Tree structure

The root of a DTD tree is always the <dtd> element. Constructing a DTD tree is very easy, just as JSGF grammar rules are expanded until every branch of the tree reaches its leaf node.

As we can see that the DTD tree has the identical structure to the dialog tree. However, they specify different domains, and each node at the same position in two trees has two different meanings. In the DTD tree above, the first level of the hierarchy is the root element; the second and third levels are the branches; the fourth level is the leaf; and the fifth level is the leaf content which, in DTD, is the text. The *leaf* is the node containing only the content node, which may be either an element node or an attribute node. The *branch* can be any other node except for the root node <dtd>.

Here we should point out that the DTD tree is slightly different from the XML document tree that is shown in figure 2.2. In figure 2.2, we have two <course> nodes, and every leaf node has its own content. On the other hand, in the DTD tree, only one <course *> node and text content of leaf node are presented. The DTD tree is the skeleton of the document instance, so it presents the structure of the document in a more general way, which means it presents a class of the document instances being validated against this DTD.

Another note for the DTD tree: if a leaf node, for example, title, allows repeated occurrences with multiple content like <course*>, we denote such a node as <title*>. In the DTD tree, there is only one <title*> node. However, this node is not a leaf node, so we count it as a branch node in mapping process. Leaf node only denotes the *'pure'* leaf under any circumstances.

Now, we can get the basic mapping ideas for the data input application from the dialog tree and DTD tree structures. That is:

- Map the branch to <form>, and element name maps to <form> `id` attribute
- Map the leaf to <field>, element name maps to <field> *name* attribute
- Map the content to the dialog items then apply prompts, grammars, <help> etc.

7.1.2 Mapping Complex Content Models

Example 7.1 is relatively simple in terms of its content model reflecting a simple DTD structure. The methods to handle various parts of a more complex content model will be introduced in this section. We are not trying to be exhaustive, only to consider important variants.

- *Mapping Sequences*

Sequence nodes are mapped to squares in the DTD tree such as `title` node in figure 7.4.

```
<!ELEMENT course(title, credit, section)>
<!ELEMENT section(classDay, …, instructor, examSlot)>
```

As has already been seen, the elements contained in other element, for example, title, credit, classDay, .., instructor, and examSlot are the sequence leaf nodes in the DTD tree. We map them to <field>; the section is a branch node, and we map it to <form>, which is the same as the course node. However, the connection between the `section` node and its parent node *course* is depended on a <goto> element, which we have seen some examples in Chapter 4, or a <subdialog> element. The following is a sample of a <subdialog> element used in this case:

```
<form id="course">
    <block><prompt> you are going to voice input..</prompt></block>
    <field name="title"> .. .. .. </field>
    <field name="credit"> .. .. .. </field>
    <subdialog name="section" src="#course.section"> </subdialog>
</form>
<form id="course.section">
    <field name= …> </field>
    .. ..
     <return>
</form>
```

Example 7.3 Mapping sequence

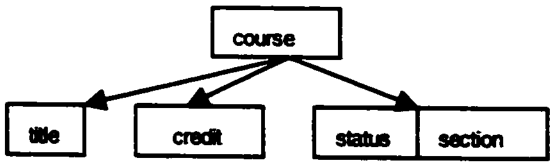
The *<subdialog>* is similar to a function call with that a set of parameters can be passed along to the subdialog. After the subdialog is executed, a *<return>* element can make the dialog flow return to the calling dialog along with a set of return variables. Note that in the example above, we use the expression: `course.section` to represent the path to the section node.

- *Mapping Choices*

Choice nodes are mapped to connected squares in the DTD tree. For example:

```
<!ELEMENT course(title, credit, (status|section) )>
```

will map to the following tree structure, where status and section are the choice nodes.



Each element in a choice may have two options, which is mapped either to a <form> or to a <field>. The choice itself will be mapped to a <field> inside its parent <form> with <option> element, which is similar to the example 4.3. The following is the skeleton of the VXML:

```
1    <form id="course">
2       <block><prompt>.. .. </prompt></block>
3       <field name="title"> .. .. .. </field>
4       <field name="credit"> .. .. .. </field>
5       <field name="choice1">
6          <prompt> Please choose <enumerate /> </prompt>
7          <option dtmf="1" value="status"> status </option>
8          <option dtmf="2" value="section"> section information</option>
9          <filled>
10            <if cond="choice1 == 'status'">
11               <goto nextitem="status"/>
12            <elseif cond="choice1 == 'section'">
13               <goto next="#course.section"/>
14            </if>
15           </filled>
16      </field>
17      <field name="status"> .. .. .. </field>
18   </form>
19
20   <form id="course.section">
21      <block><prompt>you are going to voice input..</prompt></block>
22      <field name= ...>
23      .. ..
24   </form>
```

Example 7.4 Mapping choice

A <field> named *choice1* is used to accept user's input. The contents of *<option>*s with *dtmf* attributes specify the spoken grammar and DTMF grammar for the user input. If the input is *'status'*, dialog goes to the <field> *status* (line 17). If the input is *'section'*, dialog goes to the <form> *course.section* (line 20).

- *Mapping Repeated Children*

Repeated children means that the children can occur multiple times in their parent, which are mapped to '*' at the upper-right corner of element square in DTD tree, and see the example 7.4 of *course** node. In the data input application, dealing with repeated children allow the user to voice input information continuously based on the structure of the XML document. For example:

```
1 courses(course*)
2 course(title, credit, (status | section*)
```

The first line indicates that multiple <course> elements may appear in the <courses>. The second line indicates that in <course> may have several <section> elements. No matter where the repeated elements appear, for instance, in a leaf or a branch, they will always be mapped to a <form>. In the data input applications, we can make a decision on whether or not to input another child based on the user's choice. The following code can illustrate this:

```
.. ..
<form id="course.section">
   <field>.. .. </field>
   <field name="repeat">
      <prompt> more sections? </prompt>
      <filled>
       <if cond="repeat">
         <goto next="#course.section"/>
       </if>
      </filled>
   </field>
</form>
..
```

Example 7.5 Mapping repeated children

One thing should be noted is that if we have a content model, for example:

```
<!ELEMENT book( color*)>
<!ELEMENT color (#PCDATA)>
```

The color element is a repeated leaf element, which is not a 'pure' leaf. So `color` will be mapped to a <form> with a <field> to accept the user's input and followed by another <field> to obtain user's choice on if the book has multiple colors.

7.2 Examples of Data Access Applications

Data access applications differ from the data input applications in four respects. First, data access applications have to be constructed based from XML document instances since we need to get some information from these documents (compare with, data input applications which can fully depend on a DTD file). Second, data access applications can fully take advantage of the always-active commands and customized specifications of the generic architecture to navigate and query the XML documents. In data input applications, however, the navigation commands and some direction commands are not eligible to use. Third, the users in data access applications are more active than in data input applications, in the sense that users have a capability to control where the dialog goes. Fourth, in data access applications, we provide four different ways to access XML documents: default access – directed dialog access, random path access (introduced in 5.4.2.1), search by unique key – default query rule (introduced in 5.4.3) and query rule access (introduced in 5.4.3). On the contrary, there is only one way to voice input data for data input applications.

Due to the above reasons, data access applications may have different formats for different users. We will examine mappings rules discovered in data input applications to find out whether they are still applicable for data access applications. In the following sections, we will focus on exploring and examining the approaches of the default directed-dialog access. We will also show some examples on other access methods.

7.2.1 Mapping Complex Content Models

Data access applications allow users to 'browse' XML documents. Mapping the complex content models in data access applications goes to one structure: <menu>. For example, a sequence can be mapped to a <menu>. Similarly for the repeated children, as in the following example (for example 2.2 courses.xml and figure 5.2):

```
<menu id="courses">
   <prompt> please choose <enumerate/></prompt>
   <choice next="#courses.course1" >
       [course] <sayas class="number">60100</sayas> </choice>
   <choice next="#courses.course2" > [course] 60254 </choice>
</menu>

<menu id="courses.course1">
   <prompt> please choose course 0360100's <enumerate/></prompt>
   <choice next="#courses.course1.all" > all information </choice>
   <choice next="#courses.course1.title" > title </choice>
   <choice next="#courses.course1.credit" > credit </choice>
   <choice next="#courses.course1.id"> id </choice>

   <help> please say all, title, credit, id </help>
   <choice event="backup"> back up | parent </choice>
   <catch event="backup"> <goto next="#courses" /> </catch>
```

```
18
19   <choice event="repeat"> repeat </choice>
20   <catch event="repeat"> <goto next="#courses.course1"/> </catch>
21
22   <choice event="firstChild"> first child </choice>
23   <catch event="firstChild"> first child is title
25      <goto next="#courses.course1.title"/>
26   </catch>
27
28   <choice event="lastChild"> last child |attribute</choice>
29   <catch event="lastChild"> last child is attribute identification
30      <goto next="#courses.course1.id"/>
31   </catch>
32
33   <choice event="next"> next </choice>
34   <catch event="next"> next is course 2, 254
35       <goto next="#courses.course2"/>
36      </catch>
37
38   <choice event="current"> current</choice>
39   <catch event="current"> current node path is courses, course 1
40   </catch>
41
42   <choice event="now"> what can I say now</choice>
43   <catch event="now"> you can say all, title, credit,
44        identification.the direction commands are: start over,
45        exit, help, backup, repeat. The navigation commands
46        are: parent, first child, last child, next,
47        attribute, current, list commands
48   </catch>
49
50   <choice event="list"> list commands</choice>
51   <catch event="list">
52       direction commands are: start over, exit, backup, repeat
53       navigation commands are: parent, first child, last child,
54       previous, next, attribute, next attribute, current,
55       what can I say now, list commands,
56       and you can always say help
57   </catch>
58
59   <choice event="noSuchCommand"> previous |next attribute</choice>
60   <catch event="noSuchCommand">
61       Sorry, No such command is available at current point
62   </catch>
63
64</menu>
```

Example 7.6 Data access <menu> and always-active commands

Data Access applications are associated with document instances. In the example above, line 1 to 6 map the repeated nodes *courses(course*)* to a <menu>. From line 8 to 13 map a sequence *course(title, credit, id)* to a <menu>. From line 15 to 62 illustrate one possible implementation for 14 always-active commands for *course1* node except for the commands: start over and exit. These two commands are document scope commands, which means that for all of the nodes in an XML document, they have fixed destinations. Examples are available in Appendix.

The <choice> defines grammar of the user input. The attribute *next* of the <choice> indicates the target of the transition when the user input matches the grammar defined in <choice>. The attribute id of the <menu> and the attribute next of the <choice> are paths of nodes. For example, `<menu id="#courses.course1">` stands for that the current context node is *course1*, and its path is `courses.course1`. If a node has leaf nodes such as node course1, we provide an additional access to all its leaf nodes by specifying path to, for example, `courses.course1.all` (line 10). This "*all*" path by the way is also a part of search by unique key access in data access applications, the example of which will be presented later.

The always-active command grammars in the example above are also defined by <choice>s, which make the transition very simple. When the user input matches a specific command, we throw an event using attribute *event* of the <choice>, and a <catch> to hold the event. The status of active-commands for a specific node can have three cases in the <catch> element. One, the command is valid and the dialog will go to another target node, for example, *backup*; two, the command is valid but the dialog will not go to another node, for example, *list commands*; three, the command is invalid that means to the current context node, it has no destinations, for example, *previous*.

Another note for mapping complex content models is that XML document instances have already made decisions on choice children, for example, in figure 5.2, the course1 has status, course2 has sections on the choice of `(status|section)`. So there is no need to deal with choice children in the data access applications.

7.2.2 Basic Approaches

In this section, we introduce three different mapping approaches for directed-dialog access for the leaf nodes and their parent node. As we can imagine, these mapping approaches will not cover all the possibilities the user may have. However, the mapping rules described here are simple, easy to implement and they illustrate the typical cases in data access applications.

- *Approach 1*

Assume we get to the node *course1* in the file, and already have all the necessary grammars for data access. Let us see an example of a data access application (the XML and the DTD is from the example 7.1):

```
<?xml version="1.0"?>
<vxml version="1.0">
   <form id="courses">
      <field name="course1">
       <prompt> the title is 'key concept in computer science'.
       </prompt>
       <prompt> the credit is 3. </prompt>
       <prompt> the id is 0360100. </prompt>
      </field>

      <field name="course2">
       <prompt> the title is 'MSc. Master Thesis'</prompt>
       <prompt> the credit is 12 </prompt>
       <prompt> the id is 0360797.</prompt>
      </field>
   </form>
</vxml>
```

Example 7.7 Approach one for data access

This translation is simple: the document maps to a <form>; a branch maps to a <field> with the name combined with branch element node name and the order number of the branch; the leaf node with its contents maps to <prompt>.

In this case, the VXML simply "reads" through the whole file and the browser speaks the contents of the whole file.

As we can see the structure is clear, and the VoiceXML file brings the whole information about a branch to the user once, which will satisfy the users who require doing so. However, this kind of transform does not make use of VoiceXML's capability of enabling interactive dialogs between human and computer. Based on this approach, it is not easy to implement all the active commands because of the destinations of some commands are not physically labeled, in addition, users have to wait for the sequential audio until the information they wanted is given.

This approach makes the user take a passive role in the conversation, and the user has to pay much attention because of once the information is prompted by the TTS engine, it is gone. If you missed it, you have to start over. Though, this mapping may arise some problems, we cannot deny the possibility that some users do need this kind of access.

- *Approach 2*

An alternative approach is to allow the user choose the data that he/she wants to access. This approach needs to bring up the dialogs between human and computer, and it takes two steps to locate an element node in DTD tree or document tree. This approach uses the mapping rules from the data input applications and converts the same XML document to the following form:

```
<?xml version="1.0"?>
<vxml version="1.0">
   <form id="course1">
      <field name="select">
         <prompt> Please select <enumerate /> </prompt>
         <option dtmf="1"> title </option>
         <option dtmf="2"> credit </option>
         <option dtmf="3"> id </option>
         <filled>
            <if cond="select=='title'"> <goto nextitem="title"/>
            <elseif cond="select=='credit'" /> <goto nextitem="credit"/>
            <elseif cond="select=='id'" /> <goto nextitem="id"/>
            </if>
         </filled>
      </field>
      <field name="title">Title is key concept in CS </field>
      <field name="credit"> credit is 3 </field>
      <field name="id"> id is 0360100 </field>
   </form>
   <form id="course2">  .. .. </form>
</vxml>
```

Example 7.8 Approach two for data access

The example above shows that branch maps to <form>, and each <form> has an `id` attribute that is of type ID, uniquely in the whole document. Leaf maps to <field>, leaf element name maps to <field> *name* attribute. <option> is used to get the user's input from either a spoken or a touch-tone key; and a <goto> element is used to choose the target based on the input.

The purposes of this mapping are three fold: first, this format is a step towards building a more interactive, useful speech interface using VoiceXML's conversational features, it directs user's input and gets the information for the user. Second, every named node in the XML document tree or DTD tree has an associated item with a name or id in VXML. In this way the user can directly jump to the information he/she wants without waiting for the whole file to be processed sequentially like approach 1. Third, with this structure, VXML is easier to use the always-active command to locate the command's destination for a better result combining with the generic architecture.

It seems that this mapping approach has everything we want. The representation format is useful in many cases and is consistent with the current system requirements. However, if we want to extend the system later, for example, giving users ability to browse any element node at any given point, with this structure, we have to locate the branch <form> first, then locate the leaf <field> item. Another optional mapping to enable one step to access document data is introduced next.

- *Approach 3*

In order to access data more directly and make them available for all possible VoiceXML flow control tags to access, we can map every leaf element to a <form> with a unique id attribute to identify it. The potential strength of this approach for one-step access XML data is achieved by speaking the path of the node, which is random access. First, we will give an example to show the mapping rules of this approach. Then we offer an example of random access.

```
<?xml version="1.0"?>
<vxml version="1.0">
```

```
    <menu id="course1">
       <prompt> Please choose <enumerate /></prompt>
       <choice next="#course1.title"> title </choice>
       <choice next="#course1.credit"> credit </choice>
       <choice next="#course1.id"> id </choice>
    </menu>
    <form id="course1.title">
       <block> title is key concept in computer science </block>
    </form>
    <form id="course1.credit">
       <block> credit is 3 </block>
    </form>
    <form id="course1.id">
       <block> id is 0360100 </block>
    </form>
    <menu id="course2"> .. .. </menu>
    <form .. > .. </form>
</vxml>
```

Example 7.9 Approach three for data access

In the above example, every leaf element maps to a <form>, and its path expression is as <form>'s *id* attribute. A branch maps to a <menu>, each child of the branch maps to a <choice> with next attribute to point to the target either a <form> or another <menu>.

As we mentioned before, this approach has all the advantages of approach two. Moreover, it brings a possibility to user to navigate any node in an XML document at any give point provided the appropriate grammars and transition mechanisms are available. In the following example, we use access.gram introduced in Chapter 5 to accept the path of a node:

```
<form id="random">
   <field name="path">
   <prompt>please say random search path </prompt>
   <grammar src="/random.gram" type="application/x-jsgf"/>
      <filled>
       <goto expr="'#'+path"/>
      </filled>
   </field>
</form>
```

Example 7.10 Random access example

In approach 3, every branch is mapped to a <menu>, and every leaf is mapped to a <form>. Both <menu> and <form> have an attribute *id* to uniquely identify the corresponding node, and the value of the *id* is the path expression of the node, so with the example above, we are able to locate the specific node by speaking its path.

Of course, the approaches described above are not the only ways to translate XML to VXML for data access applications. However, they are simpler and more direct than other mapping possibilities. The structure is simple and easy to construct. Generally speaking, the third approach is recommended, which is the default option in the XML to VXML Mapping Algorithm that is presented later.

7.2.3 Search by Unique Key Access

In this section, we will give an example of searching by unique key access the XML data. As a matter of face, this method is an implementation of the default query rule that is introduced in the last section of Chapter 5.

```
<form id="search">

      <block>
         <prompt> please search by course identification,
            please say an identification number </prompt>
      </block>

      <help> For example, say course id 0360100, 0360254 </help>

      <field name="search" >
         <filled>
         <if cond="search=='0360254'">
               <goto next="#courses.course2.all"/>
         <elseif cond="search=='0360100'" />
               <goto next="#courses.course1.all"/>
         <else /> <throw event="noSuchCourse" />
         </if>
        </filled>
      </field>

      <catch event="noSuchCourse">
            no such course included in the document
      </catch>
</form>
```

Example 7.11 Search by unique key access

In the example 7.6 mapping complex content models, we mentioned that if a node has leaf nodes such as node course1, we provide an additional access to all its leaf nodes by specifying path to, for example, *courses.course1.all*. The search by unique key access basically accept the course *id* in the example above, then the system will transit the dialog to the corresponding "*all*" <form> based on the user input (line 13 and 15).

The main parts of data input application and data access application for the courses.xml (example 2.2 and figure 5.2) are documented in Appendix.

One important point for this chapter to reiterate is that, so far, the XML to VXML mappings are based on the DTDs since the DTD is widely used to validate the XML documents. When mapping from a DTD, every element or attribute is mapped to a string, since there is no way to predict the target data type from a PCDATA-only element type. However, when mapping from an XML Schema, the target type can be captured and used in the mapping process since XML Schemas have over 44 data types. In the next chapter, we will see how to map XML Schemas to VXML applications.

Chapter 8

MAPPING XML SCHEMAS

We studied XML Schemas in Chapter 2. The XML Schema notation has become a W3C recommendation since May 2001. It has the same syntax as XML instance documents have, supports more than 44 data types, and allows users to define their own data types. Its extensibility and modularity make it possible to reuse and describe the content model in great detail.

Our primary reason for examining XML Schemas is because it is likely that XML Schema specifications will replace the DTD in the near future. Therefore, it is crucial for us to consider and provide a way to map the XML documents to VXML based on the XML Schema. After investigating DTD, XML Schema, VXML and going through the mapping process based on DTDs, We found:

- The mappings from XML to VXML are focused on the structural transformation and data transformation. The structural transformation is mainly based on the tree structure.
- The mapping based on DTDs cannot predict the target data type from a PCDATA-only element, which will affect the way that the TTS engine interprets and pronounces the data in a prompt.
- It is essential to take advantage of XML Schemas data types, and map them to the VXML built-in grammar types.
- There is no need for mappings for every XML Schema components, such as attribute group and model group since there is no counterpart of these components in VXML.

From these observations, the mapping issues based on XML Schemas involve the following aspects: which components of XML Schema will be used for mappings; how to construct a similar DTD tree structure from an XML Schema; and how to map XML Schema data types to VXML built-in grammar types.

8.1 Schema Tree

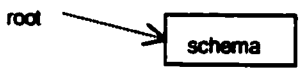
First of all, we need to make it clear that the *Schema Tree* is used to show the structure of a set of XML instances like the DTD tree does for DTDs rather than to express the structure of the schema itself. So the schema tree cannot be directly constructed from the schema, it needs some steps to form. For example, the following fragments of XML Schema are from Appendix-2 for courses.xml, and an XML instance and DTD are shown in the example 7.1:

```
<xsd:schema>
 <xsd:element name="courses" >
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element name="course" type="courseEntry"
                       minOccurs="0" maxOccurs="unbounded">
       </xsd:sequence>
       </xsd:completype>
   </xsd:element>

 <xsd:complexType name="courseEntry">
       <xsd:sequence>
           <xsd:element name="title" type="xsd:string"/>
           <xsd:element name="credit" type="creditType"/>
       </xsd:sequence>
       <xsd:attribute name="id" type="idType" use="required"/>
 </xsd:complexType>

 <xsd:simpleType name="creditType">
       <xsd:restriction base="xsd:integer">
         <xsd:minInclusive value="0"/>
         <xsd:maxInclusive value="12"/>
       </xsd:restriction>
 </xsd:simpleType>

 <xsd:simpleType name="idType">
       <xsd:restriction base="ID">
         <xsd:length value="7"/>
         <xsd:pattern   value="\d{7}"/>
       </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```

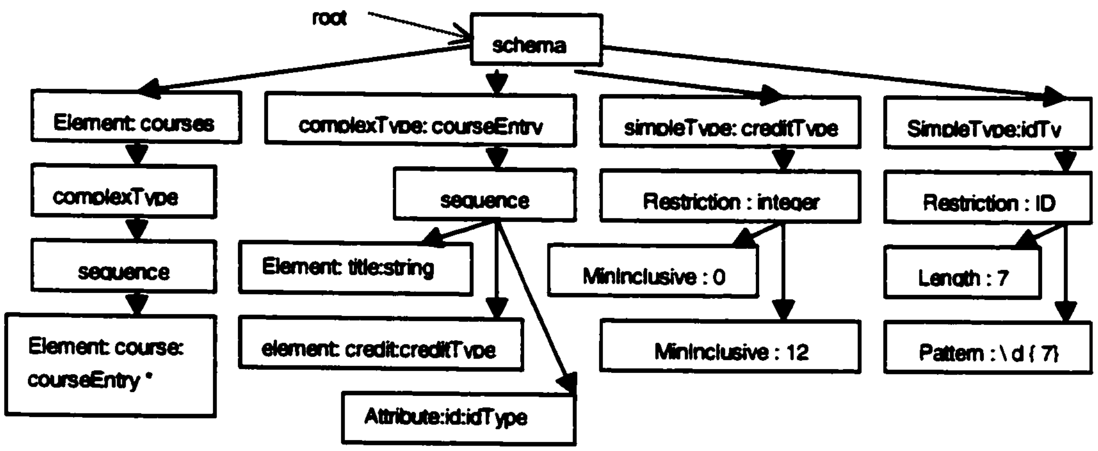
Example 8.1 Mapping schema sample

Remember that the core components of an XML document is the elements and attributes, so no matter what schema components are used in the schema file, what kind of schema formats (inline, ref or group) to decorate the elements, attribute, data types and their relationship, we should focus on the core items: elements and attributes. To construct a Schema tree from above schema, we give the following rules:

1 Creating the root of a schema tree, which always is the "schema".



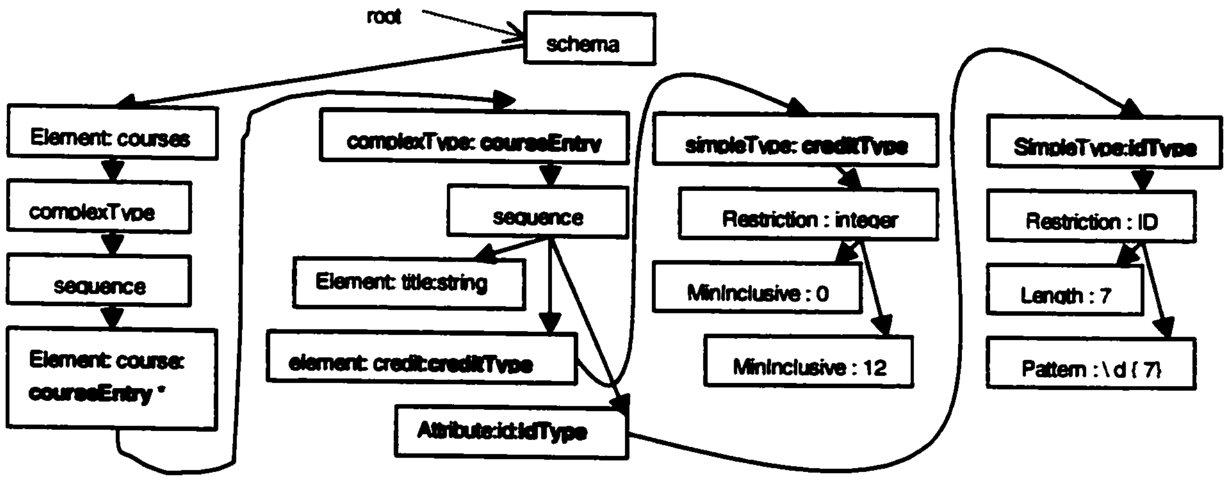
2 Building the children of the schema: following the hierarchical order of the schema document and using the local name followed by its attributes' value if any, delimited by ":" to denote the node in the tree for clarity and simplicity.



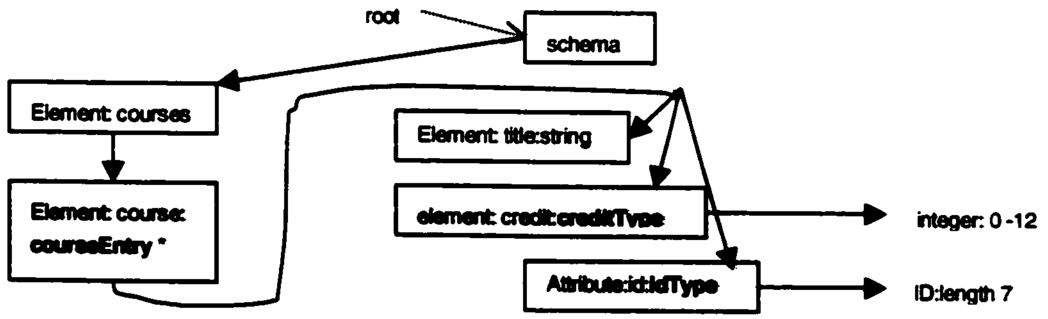
'*', '+', or '?' is used to denote the element occurrence with the following meanings:

minOccurs	maxOccurs	uses
"0"	"unbounded"	*
"1"	"unbounded"	+
"0"	"1"	?

3 Combining model group, attribute group, complex type, and simple type to their owner element or attribute, which includes the changing the hierarchical relation, moving and appending them to the parent that they are related.



4 Except for the elements and attributes, remove all other schema elements: groups such as *group model* and *attribute group*, compositors such as *choice*, *sequence* and *all*, constraints such as *key* and *unique*, derivations such as *restrictions* and *extension*, *annotations*, etc. While removing a simpleType element, leave the base type and facets and use them as content constraints to the element or attribute it refers.



5 Use only the element or attribute name to denote a node. If an element or attribute has a built-in data types, we use that type denote the content constraint of this node. Reshape the structure and get the schema tree.

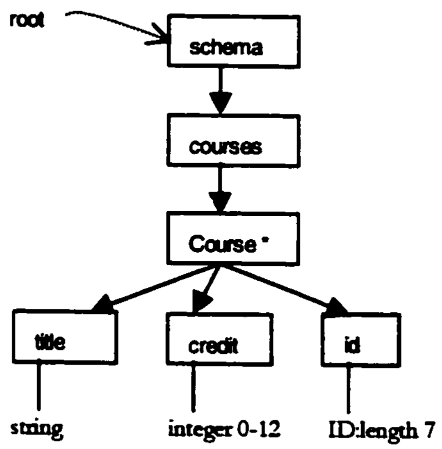


Figure 8.1 Schema tree

Comparing figure 7.4 DTD tree and figure 8.1 Schema tree, we have exactly the same structure except that the types of the leaf nodes are more specific in Schema tree, as discussed later. It is not a surprise for us because the purpose of the both DTD and Schema is to define a class of XML documents. As a matter of fact, this result is just what we have expected: mapping XML to VXML based on a tree structure by which the translating process has consistent rules to apply. In addition, this structure is close to the natural form of XML document instances with which it is easier to combine the structure with the instances to build the data access applications. Furthermore, this transition avoids ambiguity descriptions in the mapping process.

Nevertheless, the Schema tree does have a difference to the DTD tree: every leaf's content has a specified data type with facet constraints rather than a 'text'. That is one of the reasons why the XML Schema is being developed and adopted. Having a specified data type for every element and attribute makes a significant impact on the process of translating XML to VXML because data type will affect the recognition engines on how to recognize the user's utterances, and TTS engines on how to pronounce a word, which in turn has a great effect on the voice application system. The next section we will focus on how to take data types into account to make a difference in mapping process.

8.2 Data type Mappings

XML Schema has 44 built-in data types, while the VXML has 7 built-in grammar datatypes, such as *boolean*, *date*, *digits*, *currency*, *number*, *phone*, and *time* and the default is *string*. The data type mappings between them is shown as follows (the slash '/' stands for the default type *string*):

Schema Datatypes	VXML Types	Schema Datatypes	VXML Types
string	/	time	time
normalizedString	/	dateTime	date and time
token	/	date	date
boolean	boolean	gMonth	date
unsignedByte	number	gYear	date
byte	number	gYearMonth	date
integer	number	gDay	date
positiveInteger	number	gMonthDay	date
negativeInteger	number	Duration	/
nonNegativeInteger	number	Name	/
nonPositiveInteger	number	QName	/
int	number	NCName	/
unsignedInt	number	anyURI	/
long	number	language	/
unsignedLong	number	ID	/
short	number	IDREF	/
unsignedShort	number	IDREFS	/
decimal	number	ENTITY	/
float	number	ENTITIES	/
double	number	NOTATION	/
hexBbinary	digits	NMTOKEN	/
Base64Binary	digits	NMTOKENS	/

Figure 8.2 Data type mapping table

We map the XML Schema data types in three ways:

1 In data input application, a leaf maps to a <field>. If the leaf has a data type in Schema tree, a *type* attribute will be appended followed by the *name* attribute, and the value of the *type* attribute will be the mapped VXML built-in grammar type.

2 In data input application, if a leaf element has a specified data type with facet constraints, we can add a verification block in the <field> to verify that user's input is legal and insert these constraints in the <help> event of prompts pattern. For example, in schema tree figure 8.1, credit is the type *'integer'* – mapped to *'number'* with constraints value between 0 and 12. The following code illustrates case 1 and 2 together:

```
<field name="course.credit" type="number">
   <prompt> Please say the course's credit </prompt>
   <prompt count="2"> Course's credit, please? </prompt>
   <help> course's id is a number from 0 to 12</help>
   <noinput> I could not hear you.
    <reprompt/>
   </noinput>
</field>
```

3 In access data application, when the TTS engine will prompt the requested information of a particular element or attribute, use <sayas> to markup the output speech to have a more natural way to communicate with users.

```
<!--before:-->
<prompt>
    <div> the course credit is <break size="medium" />
       <emp> 12 </emp>
    </div>
</prompt>

<!-- after -->

<prompt>
    <div> the course credit is <break size="medium" />
       <emp> <sayas class="number">12</sayas> </emp>
    </div>
</prompt>
```

Chapter 9

XML TO VXML MAPPING ALGORITHM (XVMA)

So far, we have presented the XVMA at a physical level with various examples of miscellaneous aspects of translating XML to VXML, many of them are typical samples that can be used as templates in the mapping process. In this chapter, we will introduce two phases of XVMA, and finally give the XML to VXML Mapping Algorithm for Data Input Application (XVMA4DIA), and XML to VXML Mapping Algorithm for Data Access Application (XVMA4DAA).

9.1 Initialization Phase

The initialization phase has the following tasks:

- Obtain the XML source: XML document, DTD or XML Schema that will be used to build the associated tree structures.
- Get all the grammars generated by grammar builder to be ready for use, including the customized specification queryRule.gram for data access.
- Initialize the variables used to track the destinations of always-active commands for data access applications.

9.2 Body Phase

The body phase consists of two function models and several processes:

The select model: is responsible for selecting the next XML node for translation either based on the tree order from the left to the right (especially in data input applications) or based on the user's commands in data access applications. If a command is invalid at a point, the select model should return an appropriate message to the user.

The assemble model is in charge of using mapping rules in the different situations that we have discussed to construct the VXML code. It includes:

- Construct the introductory messages for different applications
- Map the current node(s) to dialog <menu>, <form>, or <field> according to the content model of the XML source , and consider the following cases:
    - o Branch, leaf, sequence, choice, repeated children.
    - o If necessary, construct the assistant items such as <subdialog>, <goto>, <option>, <filled>, etc., for the content model.
- Setup the prompt <prompt>, <help> and <grammar> for the each node(s) according to the content model and data type.
- Markup the output speech while considering the data type markup.
- Update the destination variables of always-active commands.
- Construct the user-specified queries.
- Handle the ending messages.

9.3 XVMA for Data Input Application (XVMA4DIA)

```
main(){
   <?xml version="1.0"?>
   <vxml version="1.0">

      initial();
      welcome();
      process();
      end();

   </vxml>
}

//If a direction command has a fixed destination, we use <link> to define the
  command  grammar, and set document scope help.
initial(){
   <link next="#welcome">
        <grammar> start over </grammar>
```

```
    </link>

    <link next="#end">
        <grammar> exit |goodbye </grammar>
    </link>

    <help> you can always say "list commands", "start over", or
           say "goodbye", "exit", to quit the system.
    </help>
}

//Construct the introductory messages
welcome(){
      <form id="welcome">
         <block>
            <prompt> Welcome to the voice input $rootElement
                     information system </prompt>
            <prompt> You can always say start over to start input
                     again, or say goodbye to exit. </prompt>
            <goto next="#$rootElement"/>
         </block>
      </form>
}

//Process each element in the source
process(){
   set currentNode = rootElement;
   path=$currentNode.path();

   if (currentNode == leaf){
      <form id="$path"> leafProcess(path); </form>
   }elseif (currentNode == branch) {
      brahchProcess(path);
   }
}

//Construct the ending messages
end(){
   <form id="end">
      <block>
       <prompt> Thanks for using $documentElement voice input
                system. Goodbye.
       </prompt>
       <exit />
      </block>
   </form>
}

leafProcess(path){
   currentNode=$path.current();

   if ($currentNode.dataType can map to grammarType)
      store grammarType;
   get $currentNode.properties;
```

```
    if currentNode is optional{
       optionalProcess(path);
    }
    <field name="$currentNode" type="&grammarType">

       //grammar ready;
       <grammar src="url/dictation grammar"
                type="application/x-jsfg" />

       promptPatternProcess(path,$currentNode.properites);
       commandProcess(path);
    </field>
}

branchProcess(path){
    currentNode=$path.current();
    <form id="$path">
       if (currentNode is root){
          <block> Now, you are going to voice input $currentNode
                  information </block>
       }

       while (has more subElements){
          currentNode=$subElement;
          path=$currentNode.path();

          if (currentNode==leaf) leafProcess(path);

          elseif (currentNode==choiceBranch){
             choiceProcess(path);
          }

          elseif (currentNode== B or B*)
          (where B: subelements branch; B* repeated branch){
             if (currentNode == B*) optionalProcess(path);
             <subdialog name="$B" src="#path">
             </subdialog>
          }//if
       }//while
       if (repeated branch) do repeatedProcess($B.path());
    </form>
    branchProcess(branch B);
}

choiceProcess(path){
    choiceElements C1, C2 .. Ci=$path.current();

    <field name="choice&C1">
       <prompt> Please choose to input the following:
                <break /> <enumerate/> </prompt>
       <help> please say (enumerate each choiceElement)</help>
       for (each choiceElement Ci){
        <option dtmf="&i" value="$Ci"> $Ci information </option>
       }
        <filled>
```

```
         Field variable F = current field name;
         For (each Ci){
            <if cond="&F == 'Ci'">
               <goto nextitem="&Ci"/>
               //assign other choiceElement Cj (j <> i ) expr to true;
               <assign name="&Cj" expr="true" />
            </if>
         }
      </filled>
      commandProcess(path);
   </field>

   for (each Ci){
      if (Ci is a leaf) {
          leafProcess($chooseElement.path());
          inside leafProcess() <field>, insert:
          <filled>
               if ( (Next = Ci.next() ) == invalid)
                     <goto nextitem="confirm" />
               else  <goto nextitem="$Next" />
          </filled>
      }elseif (Ci is a branch){
          <subdialog name= "&Ci" src="$path">
               <prompt> this is &Ci </prompt>
               <filled>
                  if ( (Next = Ci.next() )=invalid)
                        <goto nextitem="confirm" />
                  else  <goto nextitem="$Next" />
                  commandProcess($path);
               </filled>
          </subdialog>
      }//if
   }//for
}//end of choice
```

```
optionalProcess(path){
   <field name="option" type="Boolean">
       <prompt> $currentNode is optional, do you want to input?
       </prompt>
       <help> say yes to input, say no to skip </help>
       <filled>
         <if cond="option">
            <goto nextitem="$currentNode" />
         <else/>
            goto next available form, form item, or return.
         </if>
       </filled>
       commandProcess(path);
   </field>
}

repeatedProcess(path){
   repeatedNode=$path.current();

   <field name="confirm" type="boolean">
```

```
    <prompt> more $repeatedNode? </prompt>
    <help> please say yes or no </help>
       <filled>
        <if cond="confirm">
          store user's input; //not included in XVMA4DIA
          <goto next="#$path"/>
          <else /> <return />
        </if>
       </filled>
       commandProcess(path);
    </field>
}

commandProcess(path){

    // Construct "list commands" - always-active command;
    <link event="list">
       <grammar> list commands </grammar>
    </link>

    <catch event="list">
       Available commands are: &eachAvailableDirectionCommands
    </catch>

    //Except for "start over", "exit", "list commands, construct other available commands;
    for (each such command C){
       update destination variable V for command C;
       <link next="#&V">
          <grammar> &C </grammar>
       </link>
    }

    //Process unavailable direction commands: C1, C2, ..;
    <link event="error.noSuchCommand">
       <grammar> C1 | C2 |.. </grammar>
    </link>
    <catch event="error.noSuchCommand">
       Sorry, no such command.
    </catch>
}

promptPatternProcess(path,$currentNode.properties){

    currentNode=$path.current();
    parentNode=$currentNode.parent();
    $currentNode.dataType=$currentNode.properties.dataType;
    $currentNode.typeConstraint=
                    $currentNode.properties.typeConstraint;

    if (parentNode is valid){
       <prompt> what is the $currentNode of $parentNode? </prompt>
    else {
       <prompt> what is the $currentNode? </prompt>
    }
    <prompt count="2"> $currentNode, please?</prompt>

    If (dataType, typeConstraint are valid ){
       <help> $currentNode is $currentNode.dataType,
                 and is $currentNode.typeConstraint </help>
    else {
       <help> please say $currentNode. </help>
    }

    <noinput> Sorry, I could not hear you.
         <reprompt/>
    </noinput>
}
```

Example 9.1 XVMA4DIA

9.4 XVMA for Data Access Application (XVMA4DAA)

```
This algorithm describes 4 ways to access XML documents:
  1. Approach 3 - Directed Dialog Access
  2. Random Path Access
  3. Search by Unique Key Access: Default Query Rule
  4. Query Rules Access

main(){
     <?xml version="1.0"?>
     <vxml version="1.0">

        initial();
        welcome();
        randomAccess();
        searchByProcess();
        queryRuleProcess();
        end();

     </vxml>
  }

  initial(){

     // declare always-active commands variables: used to store and update
        the changed destinations for such commands, e.g.:

     <var name="backup"/>

     // declare variable "formID" to store the path of form and menu: attribute id

     <var name="formID" />

     //if a direction command has a fixed destination and return, we use <link> to
```

```
        define the command grammar. For commands: start over, exit.

    <link next="#welcome">
        <grammar> start over </grammar>
    </link>

    <link next="#end">
        <grammar> exit |goodbye </grammar>
    </link>

    //Links to the randomAccess, searchByProcess, queryRuleProcess, and Process;

    <link next="#random">
        <grammar> random access | random search </grammar>
    </link>

    <link next="#search">
        <grammar> search by |query by </grammar>
    </link>
    <link next="#query">
        <grammar> query pattern | query rule </grammar>
    </link>

    <link next="#$rootElement">
        <grammar> default [access] </grammar>

    //set document scope help
    <help> you can always say start over, goodbye,
           help and other active commands to proceed.
    </help>
    <help count="2"> you can say random access, search by,
           query pattern and default access to access information
    </help>
}

//Construct the introductory messages
welcome(){
    <form id="welcome">
       <prompt> Welcome to the automated voice
              query $documentElement system. </prompt>
       <prompt> you can always say start over to start again, or
              say exit to quit the system. </prompt>
       commandProcess(welcome);
       set currentNode = rootElement;
       path=$currentNode.path();
       process(path);
    </form>
}

//Construct the ending messages
ending(){
    <form id="exit">
       <block>
         <prompt> Thanks for using voice access $documentElement
```

```
                 information system. Goodbye! </prompt>
          <exit />
        </block>
    </form>
}

//for random path access XML
//random.gram is a combination with eleAttr.gram and built-in grammar number
randomAccess(){
    <form id="random">
        <assign name="formID" expr="random"/>
        <field name="path">
            <prompt> please say random search path </prompt>
            <prompt count="2"> path, please? </prompt>
            <help> For example, say courses, course1, a path please.
            </help>
            <grammar scr="url/access.gram" type="application/x-jsgf"/>
            <filled>
                <goto expr="'#'+path" />
                commandProcess(formID);
            </filled>
        </field>
    <form>
}

//an implementation of the default queryRule, search information by (if any) unique ID
  at most top level of XML document instance.
searchByProcess(){
    get keyItem of query Node;
    if ($keyItem.dataType can map to grammarType){
        store grammarType;
        typeConstraint=$keyItem.properties.typeConstraint;
    }

    <form id="search">
        <block>
            <prompt>search by $keyItem, please say a $keyItem</prompt>
            <assign name="formID" expr="search"/>
        </block>
        <help> a $keyItem please? </help>
        <field name="search" type="grammarType">
            <prompt> please say a $keyItem </prompt>
            <help> the $keyItem is &grammarType,
                     and has &typeConstraint </help>
            <filled>
                for (every $keyItemi appears in XML document){
                    parentNodei=$keyItemi.parent();
                    parentPathi=$parentNodei.path();
                    <if cond="search=='$keyItemi'">
                        <goto next="#$parentPathi+'.all'" />
                    <elseif cond="search=='$keyItem(i+1)'"/>
                        <goto next="#$parentPath(i+1)+'.all'" />
                    ..
                    <else /> <throw event="noSuch$keyItem" />
                <\/if>
             }
          </filled>
       </field>
       <catch event="noSuch$keyItem">
          no such $keyItem included in the document.
       </catch>
       commandProcess(formID);
       flowControl(formID);
    </form>
}

// implementation of customized queryRules
queryRuleProcess(){
    get queryRule.xml
    <form id="query">
       <field name="query">
          <prompt>This is query rule section, please say a query
          </prompt>

          <prompt count="2"> For example,$queryRule1?</prompt>
          <help> query rule, please? </help>

          <filled>
          </filled>
          for (each queryRulei){
             <link next="#query&i">
                <grammar> grammar queryRulei in queryRule.gram
                </grammar>
             </link>
          }
          commandProcess(query);
       </field>
    </form>

    for (each queryRulei){
       <form id="query&i">
        <block>
          <prompt> &resultOfQueryRulei </prompt>
          <assign name="formID" expr="query"/>
          commandProcess(formID);
          flowControl(formID);
       </form>
    }
}

process (path){
    courrentNode=$path.current();
    if (currentNode == leaf){
       leafProcess(path);
    }elseif (currentNode == branch) {
       brahchProcess(path);
    }
}
```

```
leafProcess(path){
   currentNode=$path.current();
   parentNode=$currentNode.parent();
   if (currentNode!=all){
      if ($currentNode.dataType can map to grammarType)
       get grammarType otherwise grammarType=null;
      <form id="$path">
       speechMarkupProcess(currentNode, parentNode, grammarType);
       commandProcess(path);
       flowControl(path);
      </form>
   }else{
      <form id="$path">
         for(each leaf currentNode=parentNode.subElement()){
            if ($currentNode.dataType can map to grammarType)
               get grammarType otherwise grammarType=null;
            speechMarkupProcess(currentNode, grammarType);
         }
         commandProcess(path);
         flowControl(path);
      </form>
   }
}

branchProcess(path){
   if currentNode is repeated branch
      repeatedProcess(path);
   else {
      currentNode=$path.current();
      subElement=currentNode.subElement();
      if (number(subElement)==1)
         process(subElement);
      elseif (number(subElement)>1){
         <menu id= "$path">
            <prompt> please choose <enumerate /></prompt>
            <choice next="$path+'.all'"> all information </choice>
            after the <menu> process("$path+'.all'");
            for (each subElement){
               path=$subElement.path();
               <choice next="#$path"> $subElement </choice>
            }
            <help> please say all, each $subElement </help>
            commandForMenuProcess(path);
       </menu>
       for (each subElement) process(subElement.path());
      }
   }
}

repeatedProcess(path){
   currentNode=$path.current();
   num=1;
   <menu id= "$path">
      <prompt> please choose <enumerate /></prompt>
      while (has more repeatElement){
            <choice next="#$path&num"> $currentNode&num </choice>
            num++;
         }
         commandForMenuProcess(path);
      </menu>
      process(currentNode.subElement());//for each subElement;
}

commandProcess(path){
      update every destination variable V for each command C;

      for(commands: backup, repeat, parent, first child, last child
         previous, next, attribute, next attribute){
         if (V!=null){
            <link event="&C>
               <grammar> &C </grammar>
            </link>
            <catch event="&C>
               <goto expr="&V"/>
            </catch>
         }else {
            <link event="noSuchCommand">
               <grammar> &C </grammar>
            </link>
            <catch event="noSuchCommand">
               Sorry, such command is unavailable at current point.
            </catch>
         }
      for (commands: current, list commands, what can I say now){
         <link event="&C">
            <grammar> &C </grammar>
         </link>
         <catch event="&C"> &C is/are: &V </catch>
      }
}

// same with commandProcess, use <choice> instead of <link>
commandForMenuProcess(path){
      update every destination variable V for each command C;

      for(commands: backup, repeat, parent, first child, last child
         previous, next, attribute, next attribute){
         if (V!=null){
            <choice event="&C>
               <grammar> &C </grammar>
            </choice>
            <catch event="&C>
               <goto expr="&V"/>
            </catch>
         }else {
            <choice event="noSuchCommand">
               <grammar> &C </grammar>
            </choice>
            <catch event="noSuchCommand">
               Sorry, such command is unavailable at current point.
```

```
          </catch>
       }
    for (commands: current, list commands, what can I say now){
       <choice event="&C">
          <grammar> &C </grammar>
       </choice>
       <catch event="&C"> &C is/are: &V </catch>
    }
}

speechMarkupProcess(currentNode, grammarType){
    parentNode=currentNode.parent();
    <div> $parentNode's $currentNode is
       <break size="medium" />
       <emp>
       if (grammarType != null){
        <sayas class="&grammarType"> $currentNode.text()</sayas>
       }else {
        $currentNode.text()
       </emp>
   </div>
   <break size="medium" />
}

flowControl(path){
    name=$path.current()
    parentName=name.parent()
    if (parentName != null){
       parent[0]=parentName;
       parent[1]=1;
       parentPath=$parentName.path()
    }else {
       parent[0]=name;
       parent[1]=0;
    }

    <form id="continue">
       <field name="confirm" type="boolean">
       <prompt> Do you want to continue accessing &parent[0]?
       </prompt>
          <filled>
             <if cond="parent[1]==1">
                <if cond="confirm"> <goto next="#$parentPath"/>
                <else /> <if cond="parent[0]=='$rootElement'">
                            <goto next="#end"/>
                         <else /> flowControl(parentPath);
                         </if>
                </if>
             <else /> <if cond="confirm">
                         <if cond="parent[0]=='$rootElement'">
                            <goto next="#$rootElement"/>
                         <else/> <goto expr="'#'+parent[0]"/>
                         </if>
                      <else /> <if cond="parent[0]=='courses'">
                                  <goto next="#end" />
```

```
                                <else /> flowControl($rootElement);
                                </if>
                          </if>
                   </if>
                 </filled>
               </field>
            </form>
}
```

Example 9.2 XVMA4DAA

9.5 Functions and Notation Used in the Algorithms

The uses of functions and notation in XVMA are summarized as follows (also see 5.2):

- VXML elements, appearing in the open and close tag for example, <menu>, <form>, and <field>, denote the direct output. The contents need to be parsed.
- $currentNode – current node name. For example, `$title` stands for `'title'`. It has two properties – dataType, constraint and several methods.

We use, for example, $title.dataType, $title.typeConstraint and the following methods to describe the node title's properties - $title.properties. Properties of a node are a set of name/value pairs.

Methods	Function	Methods	Function
path()	Return the full path	parent()	Return its parent node if any
previous()	Return previous sibling if any	firstChild()	Return the first child if any
next()	Return the next sibling if any	firstAtt()	Return the first attribute if any
lastChild()	Return the last child if any	lastAtt()	Return the last attribute if any
subelement()	Return all the subelements if any		

- $path – the full path of current node. For example, $courses.course.title

Methods	Function	Methods	Function
current()	Return the target of the path	root()	Return the root element

- variable, for example variable `grammarType`, use *&grammarType* to denote the content of this variable in between" " and ' ' quotation marks

As can be seen from the algorithms described above, XVMA4DIA depends on the XML DTDs or Schemas, maps each node to a <form> or to a <field>. Therefore, the running time for this algorithm is O(N), where N is the total number of the elements and attributes defined in XML DTD. Data access application algorithm XVMA4DAA depends on the XML document instances, maps each node to a <menu> or to a <form>. So the running time is O(N) too except that N is the total number of the elements and attributes appear in a specific XML document instance. Both algorithms are linear. The process for always-active commands attached to each node is insignificant in both cases.

THE SPEECH SYSTEM THAT WAS CREATED AS PART OF THIS WORK

10.1 A data input system

The generic structure for speech system can be illustrated in the following diagram:

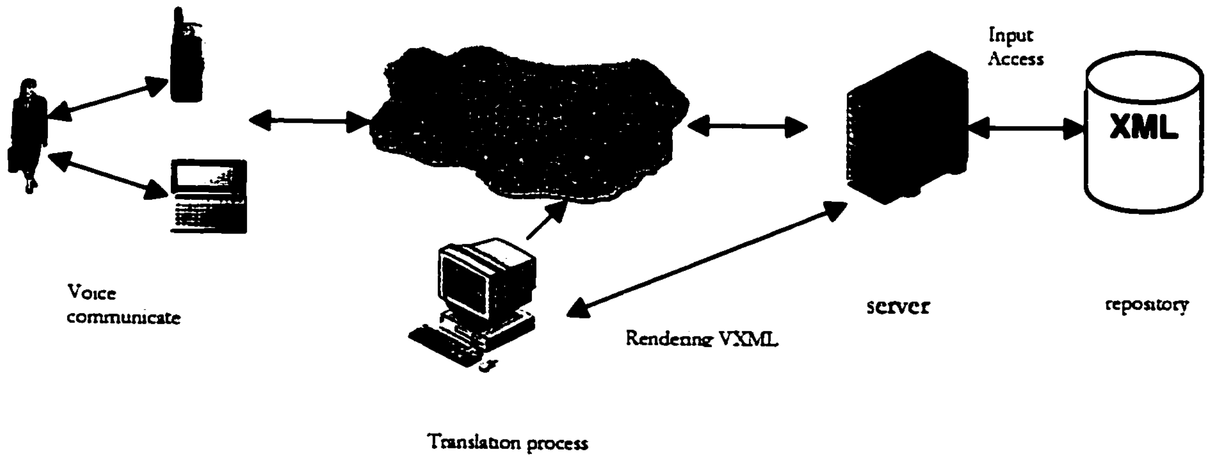


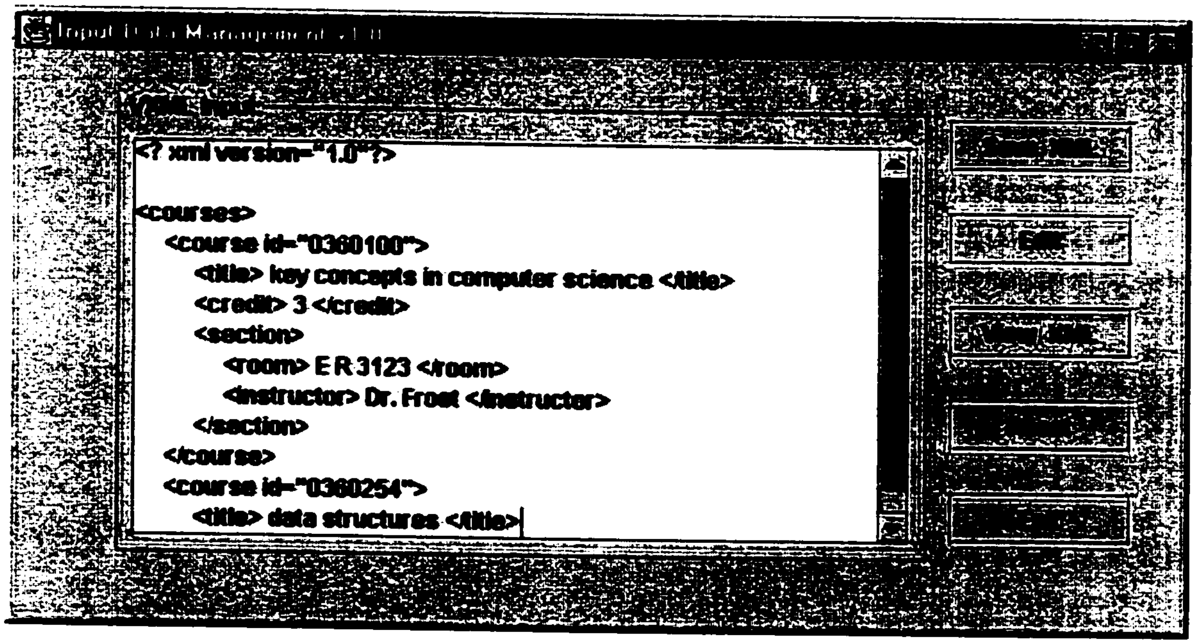
Figure 10.1 Structure of demo application

In data input system, the user can choose to enter data in courses or books. The dialog might go something like this:

```
Computer:  Please choose application: input books, input courses.
User:      Input courses.
Computer:  Welcome to the voice input course information system.
           You can always say start over to start input again, or say
           Goodbye to quit.
Computer:  Now, you are going to voice input courses information.
Computer:  What's the title of the course?
User:      Key concept in Computer Science.
Computer:  What's the credit of the course?
User:      3.
Computer:  Please choose to input: status, section information.
User:      Section information.
Computer:  What's the room of the section?
User:      E R 3123
Computer:  What's the instructor of the section?
```

```
User:       Dr. Frost.
.. ..
Computer:   More sections?
User:       No.
Computer:   What's the identification of the section?
User:       0360100.
Computer:   More courses?
User:       Yes.
Computer:   What's the title of the course?
User:       Data structures.
.. ..
```

After the above dialog, the input data manager in the server side will store the user input into an XML document according to the XML specifications. The following is the interface of XML view for user input:



```
<? xml version="1.0"?>

<courses>
   <course id="0360100">
      <title> key concepts in computer science </title>
      <credit> 3 </credit>
      <section>
         <room> E R 3123 </room>
         <instructor> Dr. Frost </instructer>
      </section>
   </course>
   <course id="0360254">
      <title> data structures </title>
```

Figure 10.2 Screen shot for data input demo application

The XML view of the user input can be saved, and it can be represented in other formats, for example in HTML or in PDF.

10.2 A data access system

The user can choose to access data in courses or books.

The dialog might go something like this:

```
Computer:  Please choose application: access books, access courses.
User:      access courses.
Computer:  Welcome to the voice access course information system.
           You can always say start over to start input again, or say
           Goodbye to quit.
Computer:  Please choose course 60100, 60254.
Computer:  60100?
User:      please choose 60100's ail information, title, credit,id.
Computer:  The title of course 60100 is key concept in computer
           science.
.. ..
User:      Search by.
Computer:  Please search by course id, please say an id number
User:      help
Computer:  For example, course 60100 say 60100.
User:      60254
Computer:  The title of the course is data structures,
           The credit of the course is 3,
           The section information, please choose section later for
           Detail. The id of the course is 60254.
           The prerequisite of the course is 100,140,212.
.. ..
Computer:  Do you want to continue?
User:      No.
Computer:  Thanks for using voice access course information system.
           Goodbye.
```

The demo source code for data input and data access can be found in Appendix.

Chapter 11

DISCUSSION AND FUTURE WORK

The ultimate goal of this work is to achieve the automated transformation from any given XML document to VoiceXML application. Through the practice and design process, we discovered that translating XML to VXML is challenging, so at the current stage, we limited the XML source to be non-domain-specific documents. This Chapter will provide suggestions on implementation of XVMA, discuss the observations we have made through the work and interesting issues that surfaced while designing the system. We also give directions for future work.

11.1 The Ease of Conversion

The ease of conversion is described from two parts: firstly, the algorithm is easy to understand; secondly, the VXML is easy to use. The VXML provides an easy interface for both developers and users since it separates the speech components from the speech engines. Consequently, it reduces the complexity of building voice applications. The ease of use of VXML has been discussed in detail in Chapter 4, therefore we will concentrate on the first point: the ease of implementation of the algorithm in the following:

We conducted many experiments on some particular aspects such as to test the speech markup output, prompt pattern, command list etc. We also presented the algorithm in a more VXML-style, thus many parts of algorithm can be picked up and used right in the coding. The algorithm is straightforward and can be implemented using many available tools. With a limited learning of VXML presented in Chapter 4, the user can even manually construct a simple translation of XML to VXML by him/herself.

From the coding techniques of the conversion, there is no difficulty. However, to implement an automatic translation system, not only the translating itself but also the output of VXML needs to be tested. Moreover, the latter needs an iterative process to refine the design, correct the prototype, and conduct tests to obtain a better speech interface. This process is time consuming and needs a lot of time to complete. From this point of view, it seems not easy at all for the conversion. On the other hand, some aspects of the prototype are based on previous work. The generic- architecture has already been tested in many facets, and our goal is to provide a generalized voice access/input system for a given XML document. Therefore, general speaking the algorithm is useful in most cases.

11.2 Available Tools

One great thing about XML is that there are numerous available open-source tools available. Our suggestion is to fully examine all the options, and combine them if possible to make the best decision. This decision is important because we want it to be as extensible as possible for enabling the future expansion of system requirements. The following lists tools available to manipulate XML data:

DOM: (the Document Object Model) [W3C Dom], which describes the document as a data structure, a tree of nodes, and allows retrieval and manipulation of the tree and its nodes.

SAX: (Simple API for XML parsing) [SAX, 1999] offers the programmer an event-driven view of an XML document as it is parsed. It lays out the document as a sequence of events and associates an event with each tag. SAX can be used on much larger documents than a DOM parser can.

XSLT: (eXtensible Stylesheet Language Transformation) [W3C XSLT] is primarily designed for transforming one XML document into another. XSLT supports the following document manipulations:

1. Selection of elements or portions of element content using the XPath syntax;
2. Rearrangement or transformation of extracted information (including not only text content but also element names) in the target document.
3. Addition of information in the target document.

JAXP: (Java API for XML parsing) (http://www.sun.com) from SUN Microsystems. It tries to standardize some aspects of Java XML parsing that DOM and SAX do not cover.

11.3 Analysis

The user interface design consists of the iteration of design, prototype construction, experimentation and evaluation steps. The generic architecture we presented is actually built on this paradigm, and some aspects of the structure are based on previous works such as [IBM, 2000], [Allen, 1995], and [Klemmer, 2000] that has been developed.

We hoped that the translating structure and mapping algorithm were able to produce the system easily. In order to give a basic test on the system, we manually constructed some examples based on XVMA. One is the example we used through the paper, a course-offering document, and another is the bibliography document from the http://www.bn.com/ (the sample XML document and DTD can be found in Appendix).

The value of the manually translation is not so much in the particular resulting applications (i.e. the course voice query/input service, or a bibliography voice query/input system), but in the proof-of-design of the applied approaches. Translation based on the XVMA can allow users voice query and input information to an XML document using a VoiceXML engine that are currently impossible or very difficult for most non-speech experts. Each of the examples demonstrates the ease of the conversion and ease of use of the algorithms.

Tests were conducted using IBM WebSphere Voice Server SDK [VoiceServerSDK], which includes a voice browser, a speech recognition engine and TTS engine and a telephony acoustic model. The transformed VXML achieved the system requirements specified in 5.3. The following are the observations made through the test:

1. In data input application, we use a limited vocabulary to test the result, and it turned out the recognition accuracy went very well in terms to response to different accents at no training situations. The speech recognition engines in recent years claim to achieve accuracy of 95% or more, at least they can reach 85% accuracy ratings [VRS]. Although, they do require up to 30 or 60 minutes of training time to accomplish speaker independence feature. In this case, we have to take further tests when we use a dictation grammar for data input applications.

2. In data input applications, we observe that there is a need to replay the user's input at a certain point to let the user hear what he/she has input before the data goes to any other repository. The replay has a verifier role in a sense of review of data and correction of incorrect recognition. The words that can be accepted by the recognition engine must exist in the grammar files thus there may not have misspelled-errors like in GUI applications. However, the word mistaken by users or confused with other available vocabulary errors was common.

3. In data access applications, the user who is unfamiliar with the document is more willing to follow the computer's prompt to access the data rather than to use the always-active command list we provided. However, for a well-known user of the system, is likely to be more active to navigate the document and query data using the command list especially the *direction* commands.

   In the test, we found that in some cases, the user might become lost after using several consecutive *navigation* commands. Therefore, the *'current'* command was added in the navigation command list to deal with the lost case. We also discovered that *"what can I say now"* command was very useful for help the user out in this kind of situations.

4. In data access applications, there is a need to add a function when the children of a parent node are more than a certain number such as 5 or 10. The system should automatically bring up this information to the user, and allow the user to select the directions, or remind the user of options at this point. The current model is to list all the children in a <menu>, then let the user make a choice. When the number of children is beyond five, it is hard for the user to memorize all the options, and it is not practical.

5. In Chapter 7, in the mapping approaches for data access application section, we mentioned that we choose *approach 3* for data access mappings because it's mapping format will allow users to browse any element node at any given point. This function was added in data access applications during the iterative process of design. We can speak the node's path to get the information we want. It is achieved by using the node

path delimited by "." as the value of attribute `id` in a *<form>*. This way we only need to generate the grammar to get to the point. The reason we use "." is because in *<form>* the `id` must be the ID type, which must conform to the constraints defined in [W3C XML].

6. In use of customization rules specified in queryRule.xml file, we discovered that the user who was not aware of query patterns and did not follow the query rule grammar structures we designed, found it hard to use efficiently. On the other hand, the designer of query rules, can get the result without any trouble.

   These findings inspired us: first, the system should provide a way to let the user know the query rule functions. Second, the sample queries may present to the user somehow. Third, when an unspecified query is asked by the user, the system should provide further message to assist the user or to exit or return to the previous point at circumstances.

7. In the test sample bibliography document (see Appendix-3), a *<book>* has a `<price>` element. The type of `<price>` in DTD is *#PCDATA*, and it would be type `decimal` or `float` in Schema. However, these types do not stand for the real type of `<price>`, which is *currency*.

   Even though we use the data types defined in XML Schema to map to the VXML grammar types, there are still some gaps between the mappings. Some data types in XML Schema have no appropriate match in VXML. Therefore, we managed to map them to *string*. However, in some cases, for example, the type *duration* is specified in a special format – a six-dimensional space, PnYnMnDTnHnMnS, where each character stands for a different dimension, the *string* for this type without knowing its semantics is meaningless. On the other hand, XML Schema does not have counterpart types such as *currency* and *phone* defined in VXML. From figure 6.6, we can see clearly that the schema types only map to five grammar types. The type *currency* and *phone* that are widely used in conversations can not be mapped at all in our system.

Examples in Chapter 4 have demonstrated that the grammar type in voice applications will affect recognition engine and TTS to represent data in totally different format. This indicates that using inappropriate grammar types could increase some mistakes or miscommunication with the human. Therefore, it is necessary for us to provide a way to allow the user to identify the data types if the schema and DTD cannot provide such information.

11.4 Future Work

Besides the above points needed to make adjustment and improvement in the future work, we still face numerous issues. For example, future work needs to implement XVMA, and tests the result fully and formally. The current mapping is still limited on non-domain-specific XML documents. And the area of user-customized specification needs to be investigated further.

There are many ways query an XML document. However, the query patterns defined in a customized specification are limited. If we could create the rules to generate match grammars for the associated query patterns, in theory, we could achieve very flexible and efficient voice query. How can we bring more productive access methods to the user? The user specified rule is the answer. A thorough exploration should be made on how to define a query rule, construct a flexible grammar from such a rule, and make use of the rule in a seamless integration with other directly accessible information.

It would be an interesting research topic to examine the possibility and methods needed to build a mapping schema between the XML XQuery language [W3C Query] expressions and VoiceXML grammars to create an advanced voice query interface Voice XQuery using VXML.

A brief introduction to XQuery language is introduced in the following:

XML Query language "provides flexible query facilities to extract data from real and virtual documents on the Web, collections of XML files will be accessed like databases" [W3C XQuery]. XQuery language is designed to support the above statement. XQuery for XML documents is very much like SQL for querying databases. An XQuery expression has a certain pattern, for example [W3C Query]:

List the titles of books published by Morgan Kaufmann in 1998.

```
FOR $b IN document("bib.xml")//book
WHERE $b/publisher = "Morgan Kaufmann"
AND $b/year = "1998"
RETURN $b/title
```

The above XQuery statement will return all the titles published by Morgan Kaufmann in 1998. With XQuery, we can easily add new element, attribute to the query output. The great thing about XQuery is that the input and the output are both in XML format, so we can continuously query. If we can map an XQuery expression to a VXML grammar and assign the semantic to the utterance received by voice browser, we can query anything we want.

At the time of writing this paper, the XML Query Language [W3C XQuery] is still in the draft form. There are several XQuery-related specifications underway. It is very likely that XQuery is going to be an extremely powerful and important query language due to the widely adopted XML.

GATXV and XVMA provide an excellent basis for transforming in a general setting. However, they also exhibit certain limitations, many interesting issues arise, while building the system, which are relevant to future efforts in the area of user-customized specifications with XML format used in speech user interface design, especially in VXML. GATXV and XVMA only begin to address limited issues and there are clear opportunities for the future research.

We can conclude that our work is only one initial step to translating the XML to VoiceXML. More extensive research and study must be undertaken in order to fully analyze the behavior and extend the functionality of the algorithms developed.

Chapter 12

CRITICAL ANALYSIS OF WORK DONE

Through design of the generic architecture, development of the mapping algorithms, implementation and testing the demo system, and analysis the current work, some issues need to be clarified; some points need to be emphasized. In this chapter, we will discuss these aspects in detail.

12.1 Investigation of Accuracy of Recognition for Data-Input Applications

In the last chapter, we analyzed that for a limited vocabulary, the recognition accuracy for data input applications went very well. It is necessary for us to examine the recognition accuracy for a "larger" grammar. Therefore, we conducted an experiment. In this section, we discuss how the experiment was designed and conducted. We then show the results and some conclusions we have made.

We created a dedicated data input VXML application for input of course instructor names based on a "larger" grammar. The VXML application repeatedly asks the user to input a name, and replays the recognized name on the computer. We recorded recognition accuracy while the participants are taking the test. These numbers are:

1. N1 – The number of names that were recognized first time when the participants speak them.

2. N2 – The number of names that were recognized after the participants repeat 2 to 3 times or change the pronunciation of those names.

3. N3 – The number of names not recognized after the effort above.

The instructor grammar used in the test is listed as follows:

```
#JSGF V1.0;
grammar instructors;

public <instructors> =

 Akaka, Daniel    |  Allard, Wayne     | Allen, George   |
 Baucus, Max      |  Bayh, Evan        | Bennett, Robert  |
 Biden Jr, Joseph  | Bingaman, Jeff    | Bond, Christopher  |
 Boxer, Barbara  | Breaux, John        | Brownback, Sam  |
 Bunning, Jim    | Burns, Conrad       | Byrd, Robert  |
 Campbell, Ben Nighthorse  | Cantwell, Maria  | Carnahan, Jean  |
 Carper, Thomas  | Chafee, Lincoln     | Cleland, Max  |
 Clinton, Hillary | Cochran, Thad      | Collins, Susan  |
 Conrad, Kent    | Corzine, Jon        | Craig, Larry  |
 Crapo, Mike     | Daschle, Thomas     | Dayton, Mark  |
 DeWine, Mike    | Dodd, Christopher   | Domenici, Pete  |
 Dorgan, Byron   | Durbin, Richard     | Edwards, John  |
 Ensign, John    | Enzi, Mike          | Feingold, Russell  |
 Feinstein, Dianne  | Fitzgerald, Peter  | Frist, William  |
 Graham, Bob     | Gramm, Phil         | Grassley, Chuck  |
 Gregg, Judd     | Hagel, Charles      | Harkin, Tom  |
 Hatch, Orrin    | Helms, Jesse        | Hollings, Ernest |
 Hutchinson, Tim | Hutchison, Kay Bailey  | Inhofe, James  |
 Inouye, Daniel  | Jeffords, James     | Johnson, Tim  |
 Kennedy, Edward | Kerry, John         | Kohl, Herb  |
 Kyl, Jon        | Landrieu, Mary      | Leahy, Patrick  |
 Levin, Carl     | Lieberman, Joseph   | Lincoln, Blanche  |
 Lott, Trent     | Lugar, Richard      | McCain, John  |
 McConnell, Mitch| Mikulski, Barbara   | Miller, Zell  |
 Murkowski, Frank| Murray, Patty       | Nelson, Bill  |
 Nelson, Ben     | Nickles, Don        | Reed, Jack |
 Reid, Harry     | Roberts, Pat        | Rockefeller IV, John  |
 Santorum, Rick  | Sarbanes, Paul      | Schumer, Charles  |
 Sessions, Jeff  | Shelby, Richard     | Smith, Bob  |
 Smith, Gordon   | Snowe, Olympia      | Specter, Arlen  |
 Stabenow, Debbie| Stevens, Ted        | Thomas, Craig  |
 Thompson, Fred  | Thurmond, Strom     | Torricelli, Robert  |
 Voinovich, George | Warner, John      | Wellstone, Paul  |
 Wyden, Ron ;
```

Figure 12.1 Instructors grammar for data input applications

The grammar listed above includes 100 names, and some of them have very close pronunciation. We tested the recognition accuracy when the names were randomly input by the user. Five people attended in the experiment with no training at all. We gave participants a list of the names that they can speak to the computer. The relevant data and the test results are given below:

Participants	Sex	Accent	N1	N2	N3
P 1	Male	No	95	4	1
P 2	Male	Yes	80	11	9
P 3	Male	Yes	82	10	8
P 4	Female	Yes	92	7	1
P 5	Female	Yes	89	6	5

Figure 12.2 Results of recognition accuracy test

From the figure above, we can see that the recognition engine of IBM VoiceServer SDK has relatively-high recognition accuracy. The accuracy ratings of the first-time speaking for all five participants are over 80%, and the average is at 88%. The overall recognition-accuracy ratings are at 95%. We have the following conclusions:

1. The recognition accuracy with a "larger" grammar for data input applications could be accepted at reasonable ratings.

2. The data input applications with a "larger" grammar are able to understand people who have accents with high accuracy. However, the inaccurate pronounce of some words and the accents are still considered as a major factor to affect the recognition accuracy.

3. We found that the only one name which was unrecognized by participants P1 and P4 was not the same, which means that we can improve the accuracy ratings by simply training the users in advance.

4. Some names that the system can not hear or misunderstanding the user in the testing, which is usual in the voice applications due to the noise in the testing environment, or the quality of the microphone used in the testing.

12.2 How to Scale up the Grammar

One approach can be used to scale up the grammar is that we can use "semantics" to restrict the grammar, for example, using the department information to restrict the user input of instructor's name:

```
Computer:  instructor department?
User:      Computer Science.
Computer:  Instructor name?
User:      Richard A. Frost.
```

From the example above, we can see that the instructor grammar can be divided into pieces by "semantics", so we can reduce the size of the instructor grammar file.

12.3 Building Distributed Applications

VoiceXML is an emerging open standard that brings the Web development paradigm to voice applications. This means that existing HTTP protocols for enterprise services can be seamlessly extended with voice. In the figure 4.3 (VoiceXML as a Web application), we illustrated that VXML was a perfectly fit with the Web development paradigm. It indicates that voice applications built using VXML can share the same business logic and application logic at the back-end. VXML enables developers to quickly develop and deploy voice-enabled e-business solutions. It allows companies to take advantage of their existing Web infrastructures for the delivery of voice-enabled Internet applications to both wired and wireless devices.

Some companies have already built the entire Web pages using XML documents. For example, The Cocoon project from the Apache aims to "change the way web information is created, rendered and served" (http://xml.apache.org). It has been claimed that XML is the future of the Web. The long-term goal of our research is to achieve automate transformation from XML applications to VXML applications. In a distributed network system (see the figure 10.1), the translation process can be deployed in a server to dynamically render XML documents to VXML documents. The XML documents can locate in the local network or the remote sites. In addition, the produced VXML documents that could reside over the network will be collected and fetched to the user. The entire process can be transparent to the end-user. In this way, we can manage to access/input large amounts of information in XML format on the fly through speech.

The Web development paradigm brings vendor and network independence to distributed voice applications, and drastically reduces the cost for the development and deployment of traditional voice applications required to quickly deliver powerful voice-based solutions. Our work provides the possibility to achieve this goal, and the demo application we developed is a step towards the objective.

12.4 Adding Flexibility to Data Access Applications

In chapter 5, we discussed the need for customized specification, the query rule schema, defining query rules based on the schema, and use of the query rules.

We defined the query rule schema (see the example 5.7) which was an XML DTD file. The user needs to specify the query-rule instances through the available XML tools or directly create XML documents that are required to be valid with respect to the schema. The query rule can be described in the notation we developed. In chapter 5, we also described that the query pattern is associated with three phases: *queryPatternImpl* phase, *grammarTranslator* phase, and *generateVxml* phase (please see chapter 5 for details).

Here, we want to emphasize that it is necessary to design customized specifications for data access because use of customized query rules is a viable alternative to extend the functions for data access applications.

CONCLUSION

The aim of our work has been to develop the mapping rules from XML to VXML in order to build a speech user interface from an XML specification. This work has addressed the development of the GATXV generic architecture, XVMA algorithms and many issues we faced during its design, particularly in the ways to manipulate speech-related components. This work is inspired by the fact of the explosive growth of the Web, the widely adopted uses of XML, and the much-needed speech applications in the real world.

We focused on exploring the underlying structural relations and data-type mappings between XML and VXML via the document instances, DTDs and XML Schemas. The generic architecture and algorithms we have developed not only address the mapping rules but also proposes a generalized model for constructing a voice application using VXML. We also made some efforts to discover the user-customized means of accessing the XML data on mixed-initiative type dialogue.

13.1 Benefits of This Work

The approach we provided in this thesis report to transforming XML applications to VoiceXML applications has the following characteristics:

- Developed algorithms and generic architecture that can be used by others to manually translate XML applications to VXML applications.
- Supported the viability of automatic construction of major part of speech applications.
- Ultimately lead to make speech applications.

- Provides an easy way for voice accessing a wealth of information.
- Takes advantage of both XML and VXML's benefit and make use of XML Schema built-in data types to gain the strength of speech interface.
- Offers the user choices to easily specify the query patterns to become involved in building the voice applications. The extensibility of this approach makes it potentially useful to expand the functionality of the system.
- Explores the underlying XML structure to capture and make use of the skeleton to build voice access/input applications.
- Transformed VXML can be deployed immediately in both Web-based and telephony-based environment.

13.2 The Defense of the Thesis

The goal of our work was to prove the thesis statement:

"It is possible to automatically translate XML applications to VXML applications".

By using the translating architecture and mapping algorithms provided in this work, we can translate XML documents to VXML documents easily. The work has shown that:

- Voice accessing and inputting information in the XML documents can be solved by automatic translation of XML documents to VXML documents, which is the ultimate goal of this work. At present, we state that it is possible to automate the translation of non-domain-specific XML documents to VXML documents. The non-domain-specific XML applications refer to the general XML documents such as course information of a school, employee records, purchase orders etc. Actually, we can translate some domain-specific XML applications such as XHTML and MathML to VXML applications using our approaches. However, some output is meaningless in terms of data. In this work, we only care about the data rather than the tags used to describe how the data would be presented.

- It is possible to automatically translate XML DTDs or Schemas with associate XML documents to VXML applications for data access.

  In the data access applications, all the grammars can be automatically generated by XML document instances, DTDs or XML Schemas. Moreover, we can always extend existing query patterns by defining more elements in the user-customized specification.

- It is possible to automatically translate XML DTDs or Schemas to VXML applications for data input with the exception that grammars will have to be constructed manually for many applications.

With the above conditions, the thesis statement: "it is possible to automatically translate XML applications to VXML applications" is true.

The following is to recapitulate the main topics addressed in this paper:

A Generic Translating Architecture: We presented a generic architecture of transformation from XML to VXML (GATXV) for constructing aural means of human-computer interface. This architecture covers the issues of the functionality, speech user interface and customized specifications in the transforming process. The user-customized specification is a step toward a more natural way to build a speech interface for querying XML data.

Two Mapping Algorithms: We introduced two mapping algorithms XVMA4DIP and XVMA4DDP for mapping XML to VXML to build data input and data access applications. The mapping system is built based on a tree structure, and data type mappings. The method for constructing such a tree structure from XML document instance, DTD or XML Schema was also presented. The mapping algorithm is a combination of translating architecture and mapping rules.

A Binding Method: We explored a binding method to make use of the variety of data types defined by XML Schema in VXML applications. To map the XML data types to VXML built-in grammar types greatly improves the understanding of synthesized speech, which leads to a better human-computer communication.

A Translating Model: We provided a generalized translating model for transforming between XML applications while developing our mapping system. We noted that XML is a meta-markup language that can be used to define various XML applications, and these applications use different interfaces to present data to different type of users. Along with universal acceptance of XML format to represent and exchange data in the Web, such XML-based applications will be adopted broadly as well. The need to build a mapping model to transform one type of XML into another is become more needed than ever before. This work is also trying to examine the possibility, and readily availability to develop such a translation model.

APPENDIX – EXAMPLES

1. Appendix-1 Schema – inlined style of example 2.1

```
<?xml version="1.0">
<xsd:schema
        xmlns:xsd=http://www.w3.org/2001/XMLSchema>
        targetNamespace="http://speechlab.uwindsor.ca"
        xmlns="http://speechlab.uwindsor.ca"
        elementFormDefault="qualified">

     <xsd:annotation>
        <xsd:documentation xml:lang="en">
        Offering courses information schema -inlined
     </xsd:documentation>
  </xsd:annotation>

  <xsd:element name="courses">
     <xsd:complexType>
        <xsd:sequence>

         <xsd:element name="course" minOccurs="0"
                                 maxOccurs="unbounded">
           <xsd:complexType>

              <xsd:sequence>
                    <xsd:element name="title" type="xsd:string"/>
                 <xsd:element name="credit">
                    <xsd:simpleType>
                    <xsd:restriction base="xsd:integer">
                       <xsd:minInclusive value="0"/>
                       <xsd:maxInclusive value="12"/>
                    </xsd:restriction>
                    </xsd:simpleType>
                 </xsd:element>
              </xsd:sequence>

              <xsd:attribute name="id" use="required">
                 <xsd:simpleType>
                    <xsd:restriction base="ID">
                       <xsd:length value="9"/>
                       <xsd:pattern
                          value="\d{2}-\d{2}-\d{3}"/>
                    </xsd:restriction>
                 </xsd:simpleType>
              </xsd:attribute>

           </xsd:complexType>
         </xsd:element>
```

```
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Appendix -1 Schema - inlined style of example 2.1

2. Appendix-2 Schema –of example 2.2

```
<?xml version="1.0">
<xsd:schema
   xmlns:xsd=http://www.w3.org/2001/XMLSchema>
   targetNamespace="http://speechlab.uwindsor.ca"
   xmlns="http://speechlab.uwindsor.ca"
   elementFormDefault="qualified">

   <xsd:annotation>
       <xsd:documentation xml:lang="en">
        Offering courses information schema -inlined
       </xsd:documentation>
   </xsd:annotation>

   <xsd:element name="courses" >
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="course" type="courseEntry"
                        minOccurs="0" maxOccurs="unbounded">
         </xsd:sequence>
       </xsd:completype>
   </xsd:element>

   <xsd:complexType name="courseEntry">
       <xsd:sequence>
         <xsd:element name="title" type="xsd:string"/>
         <xsd:element name="credit" type="creditType"/>
         <xsd:choice>
           <xsd:element name="status" type="xsd:string"
           <xsd:element name="section" type="sectionType"
                        minOccurs="1" maxOccurs="unbounded">
         </xsd:choice>
       </xsd:sequence>
       <xsd:attribute name="id" type="idType" use="required"/>
       <xsd:attribute ref="prerequisite" use="implied"/>
   </xsd:complexType>

   <xsd:simpleType name="creditType">
       <xsd:restriction base="xsd:integer">
         <xsd:minInclusive value="0"/>
         <xsd:maxInclusive value="12"/>
       </xsd:restriction>
   </xsd:simpleType>

   <xsd:complexType name="sectionType">
       <xsd:sequence minOccurs="0" maxOccurs="1">
         <xsd:element name="classDay" type="classDays"/>
         <xsd:element name="startTime" type="courseTime"/>
         <xsd:element name="stopTime" type="courseTime"/>
        <xsd:element name="room" type="xsd:string"/>
        <xsd:element name="instructor" type="xsd:string"/>
      </xsd:sequence>

      <xsd:attribute name="num" type="xsd:positiveInteger
                 use="required"/>
      <xsd:attribute name="activity" use="optional">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="lecture"/>
            <xsd:enumeration value="lab"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
   </xsd:complexType>

   <xsd:simpleType name="days">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Monday"/>
        <xsd:enumeration value="Tuesday"/>
        <xsd:enumeration value="Wednesday"/>
        <xsd:enumeration value="Thursday"/>
        <xsd:enumeration value="Friday"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="classDays">
      <xsd:restriction>
        <xsd:simpleType>
          <xsd:list itemType="days"/>
        </xsd:simpleType>
        <xsd:maxLength="2"/>
        <xsd:minLength="1"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="courseTime">
      <xsd:restriction base="time">
        <xsd:pattern value="hh:mm"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:simpleType name="idType">
      <xsd:restriction base="ID">
          <xsd:length value="7"/>
          <xsd:pattern value="\d{7}"/>
      </xsd:restriction>
   </xsd:simpleType>

   <xsd:attribute name="prerequisite">
      <xsd:simpleType>
        <xsd:list itemType="idType"/>
      </xsd:simpleType>
   </xsd:attribute>
</xsd:schema>
```

Appendix -2 Schema – complete schema of example 2.2

3. DTD for bib.xml – bib.dtd from http://www.bn.com

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ), publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (first, last )>
<!ELEMENT editor  (first, last, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

Appendix -3 DTD bib.dtd

4. XML document – sample for the bib.dtd from http://www.bn.com

```
<bib>
    <book year="1994">
        <title>TCP/IP Illustrated</title>
        <author><first>W.</first><last>Stevens</last> </author>
        <publisher>Addison-Wesley</publisher>
        <price> 65.95</price>
    </book>

    <book year="2000">
        <title>Data on the Web</title>
        <author><first>Serge</first><last>Abiteboul</last></author>
        <author><first>Peter</first><last>Buneman</last></author>
        <publisher>Morgan Kaufmann Publishers</publisher>
        <price> 39.95</price>
    </book>

    <book year="1999">
       <title>The economics of technology and Content for digital TV
        </title>
        <editor>
              <first>Darcy</first><last>Gerbarg</last>
              <affiliation>CITI</affiliation>
        </editor>
        <publisher>Kluwer Academic Publishers</publisher>
        <price>129.95</price>
    </book>
</bib>
```

Appendix -4 XML document bib.xml

5. VXML: data input application for courses.xml (example 2.2, incomplete due to the length)

```
<?xml version="1.0"?>
<vxml version="1.0">

  <link next="#welcome">
     <grammar> start over </grammar>
  </link>
  <link next="#end">
     <grammar> exit | goodbye </grammar>
  </link>
  <help> you can always say list commands, say goodbye or exit to
      quit the system, or say start over !</help>

  <form id="welcome">

     <block>
        <prompt> Welcome to the voice input course
                 information system.</prompt>
        <prompt > You can always say start over to start
                 input again, or say goodbye to quit</prompt>
        <goto next="#courses"/>
     </block>
  </form>

  <form id="end">
     <block>
        <prompt> Thanks for using voice input course
                 informaiton system, Goodbye!</prompt>
        <exit />
     </block>
  </form>

  <form id="courses">
     <block> Now, you are going to voice
              input courses information</block>
     <subdialog name="Course" src="#courses.course">
     <filled> <goto next="#end"/> </filled>
     </subdialog>
  </form>

  <form id="courses.course">
     <grammar scope="dialog"> backup </grammar>

     <field name="title">
        <grammar>   key concepts in computer science |
           programs and computers |
           advanced computer programming with C and Java |
           data structures
        </grammar>

        <prompt> what's the title of the course? </prompt>
         <prompt count="2"> title, please? </prompt>
```

```
    <help> please say title </help>
    <noinput> Sorry, I could not hear you
             <reprompt /> </noinput>
    <link event="error.nosuchcommand">
     <grammar> backup </grammar>
    </link>
    <catch event="error.nosuchcommand"> Sorry, no such command
           <reprompt /> </catch>
    <link event="list" >
     <grammar> list commands | what can I say now </grammar>
    </link>
    <catch event="list">
          Available commands are: start over, exit,
          what can I say now, list commands.
    </catch>
    <link next="#courses.course">
     <grammar> repeat </grammar>
    </link>
 </field>

 <field name="credit" >
    <grammar> 0 | 3 |12 </grammar>
    <prompt> what's the credit of the course? </prompt>
     <prompt count="2"> credit, please? </prompt>
    <help> please say credit </help>
    <noinput> Sorry, I could not hear you
             <reprompt /> </noinput>
    (active-commands process omitted)
 </field>

 <field name="choiceStatus">
    <prompt> Please choose to input the following:
       <break /> <enumerate /> </prompt>
    <help> please say status, section or press
            keypad to choose</help>
    <option dtmf="1" value="status"> status </option>
    <option dtmf="2" value="section">
          section information</option>
    <filled>
     <if cond="choiceStatus == 'status'">
       <goto nextitem="status"/>
       <assign name="section" expr="true" />
     <elseif cond="choiceStatus == 'section'"/>
       <goto nextitem="section"/>
       <assign name="status" expr="true" />
     </if>
    </filled>

    (active-commands process omitted)
 </field>

 <field name="status">
    <grammar> not offering </grammar>
    <prompt> what's the status of the course? </prompt>
     <prompt count="2"> status, please? </prompt>
```

```
        <help> only one status available  - not offering,
              please say not offering </help>
        <noinput> Sorry, I could not hear you  <
              reprompt /> </noinput>
        <filled> <goto nextitem="confirm" /></filled>
        (active-commands process omitted)
     </field>

     <subdialog name="section" src="#courses.course.section">
        <prompt> this is to section </prompt>
        <filled> <goto nextitem="confirm" /> </filled>
     </subdialog>

     <field id="continue" type="boolean">
        <prompt> more course? </prompt>
        <help> please say yes or no </help>
        <filled>
         <if cond="confirm"> <goto next="#courses.course" />
         <else/> <return /> </if>
        </filled>
        (active-commands process omitted)
     </field>
  </form>

  <form id="courses.course.section">
     <field name="instructor" >
        <grammar> Dr. Frost|Dr. Saba |Dr. Kao|Dr. Goodwin </grammar>
        <prompt> what's the instructor of the section? </prompt>
         <prompt count="2"> instructor, please? </prompt>
        <help> please say instructor </help>
        <noinput> Sorry, I could not hear you
              <reprompt /> </noinput>
        (active-commands process omitted)
     </field>

//classDay startTime stopTime room examSlot num activity omitted

     <field id="continue" type="boolean">
        <prompt> more sections? </prompt>
        <help> please say yes or no </help>
        <filled>
         <if cond="confirm"> <goto next="#courses.course.section" />
         <else/> <return /> </if>
        </filled>
        (active-commands process omitted)
     </field>
  </form>
</vxml>
```

Appendix -5 VXML data input application

6. VXML: data access application for courses.xml (example 2.2, incomplete due to the length)

```
<?xml version="1.0"?>

<vxml version="1.0">
  <meta name="accesscourse.vxml" content="courseOfferingSystem"/>

  <var name="formID" />
  <link next="#welcome">
     <grammar> start over </grammar>
  </link>

  <link next="#end">
     <grammar> exit | goodbye </grammar>
  </link>

  <link next="#defaultQueryPattern">
     <grammar> search by | query by </grammar>
  </link>

  <link next="#random">
     <grammar>random access |random search </grammar>
  </link>

  <link next="#query">
     <grammar> rule |  pattern </grammar>
  </link>

  <link next="#courses">
     <grammar> go to courses </grammar>
  </link>

  <help> you can always say list commands, say goodbye or exit to
        quit the system, or say start over!</help>

  <form id="welcome">
     <block>
        <prompt> Welcome to the voice access course information
                 system.</prompt>
        <prompt > You can always say start over to start input again,
                 or say goodbye to quit</prompt>
        <goto next="#courses"/>
        (active-commands process omitted)
     </block>
  </form>

  <form id="end">
     <block>
        <prompt> Thanks for using voice access course informaiton
                  system, Goodbye!</prompt>
        <exit />
        (active-commands process omitted)
    </block>
</form>

<form id="random">
    <field name="path">
    <prompt>please say random search path </prompt>
    <grammar src="/random.gram" type="application/x-jsgf"/>
       <filled>
         <goto expr="'#'+path"/>
       </filled>
       (active-commands process omitted)
    </field>
</form>

<form id="query">
    <block> this is query rule section, please say a query
              rule.</block>
    <field>
       <filled></filled>
       <help> query rule, please</help>
    </field>
    <link next="#query1">
       <grammar> how many courses and sections in this XML document
       </grammar>
    </link>

    <link next="#query2">
       <grammar> How many sections (does| in) course 254 [have]
       </grammar>
    </link>
    <link next="#query3">
       <grammar> [Answer the] title and credit of the course 797
       </grammar>
    </link>
    (active-commands process omitted)
</form>

<form id="query1">
    <block> There are 3 courses, and three sections in XML document.
       <assign name="formID" expr="'query'" />
       <submit next="#door" namelist="formID" />
       (active-commands process omitted)
    </block>
</form>

<form id="query2">
    <block> <prompt>There are <emp> 2 sections </emp> of course 254.
          </prompt>
       <assign name="formID" expr="'query'" />
       <submit next="#door" namelist="formID" />
       (active-commands process omitted)
    </block>
</form>
```

```
<form id="query3">
   <block>
      <prompt> <div>The title of course
         <sayas class="digit">797 </sayas> is <emp>master computer
                science Thesis.</emp>
               </div>  <break />
         <div>The credit of course
         <sayas class="digit">797 </sayas> is <emp> 12 </emp>
               </div>  <break />

      </prompt>
      <assign name="formID" expr="'query'" />
      <submit next="#door" namelist="formID" />
      (active-commands process omitted)
   </block>
</form>

<menu id="courses">
   <prompt> please choose <enumerate/></prompt>
   <choice next="#courses.course1" > [course] 100 </choice>
   <choice next="#courses.course2" > [course] 254 </choice>

   (active-commands process omitted)
</menu>

<menu id="courses.course1">
   <prompt> please choose 100's all information, title, credit,
             identification</prompt>
   <choice next="#courses.course1.all" > all information </choice>
   <choice next="#courses.course1.title" >  title </choice>
   <choice next="#courses.course1.credit" >  credit </choice>
   <choice next="#courses.course1.id"> identification </choice>

   (active-commands process omitted)
</menu>

<form id="courses.course1.all">
   <block>
      <prompt> <div>The title of course
         <sayas class="digit">100 </sayas> is
         <emp>key concepts in computer science.</emp>
               </div>  <break />
         <div>The credit of course
         <sayas class="digit">100 </sayas> is <emp> 3 </emp>
               </div>  <break />
         <div>The id of course
         <sayas class="digit">100 </sayas> is
         <emp> <sayas class="digit"> 100 </sayas> </emp>
               </div>  <break />

      </prompt>
      <assign name="formID" expr="'courses.course1'" />
      <submit next="#door" namelist="formID" />

   </block>
```

```
</form>

<form id="courses.course1.title"> .. </form>
<form id="courses.course1.credit"> .. </form>
<form id="courses.course1.id"> .. </form>

<menu id="courses.course2">
   <prompt> please choose 254's all information, title, credit,
             section, prerequisite</prompt>
   <choice next="#courses.course2.all" > all information </choice>
   <choice next="#courses.course2.title" >  title </choice>
   <choice next="#courses.course2.credit" >  credit </choice>
   <choice next="#courses.course2.section" >  section </choice>
   <choice next="#courses.course2.id"> course's identification
   </choice>
   <choice next="#courses.course2.prerequisite">
            course's prerequisite </choice>

   (active-commands process omitted)
</menu>

<form id="courses.course2.all"> .. </form>
<form id="courses.course2.title"> .. </form>
<form id="courses.course2.credit"> .. </form>

<menu id="courses.course2.section">
   <prompt> please choose course 254  <enumerate/></prompt>
   <choice next="#courses.course2.section1" > section 1 </choice>
   <choice next="#courses.course2.section2" >  section 51 </choice>
   (active-commands process omitted)
</menu>


<form id="courses.course2.id"> .. </form>
<form id="courses.course2.prerequisite"> .. </form>

<menu id="courses.course2.section1">
   <prompt> please choose the following of course 254 section 1's
             all information, class day, start time, stop time,
        room, instructor, exam slot, section's identification,
        section's activity</prompt>
   <choice next="#courses.course2.section1.all" > all information
   </choice>
   <choice next="#courses.course2.section1.classDay" > class day
   </choice>
   <choice next="#courses.course2.section1.startTime" > start time
   </choice>
   <choice next="#courses.course2.section1.stopTime" > stop time
   </choice>
   <choice next="#courses.course2.section1.room" > location
   </choice>
   <choice next="#courses.course2.section1.instructor" > instructor
   </choice>
   <choice next="#courses.course2.section1.examslot" >  exam slot
   </choice>
   <choice next="#courses.course2.section1.num" > section's id
   </choice>
   <choice next="#courses.course2.section1.activity" > section's
            activity      </choice>

   (active-commands process omitted)
</menu>

<menu id="courses.course2.section2">
      .. <choices> .. </choice>
      (active-commands process omitted)
</menu>

<form id="courses.course2.section1.all"> .. </form>
<form id="courses.course2.section1.classDay"> .. </form>
<form id="courses.course2.section1.startTime"> .. </form>
<form id="courses.course2.section1.stopTime"> .. </form>
<form id="courses.course2.section1.room"> .. </form>
<form id="courses.course2.section1.instructor"> .. </form>
<form id="courses.course2.section1.examSlot"> .. </form>
<form id="courses.course2.section1.num"> .. </form>
<form id="courses.course2.section1.activity"> .. </form>

<form id="courses.course2.section2.all"> .. </form>
<form id="courses.course2.section2.classDay"> .. </form>
<form id="courses.course2.section2.startTime"> .. </form>
<form id="courses.course2.section2.stopTime"> .. </form>
<form id="courses.course2.section2.room"> .. </form>
<form id="courses.course2.section2.instructor"> .. </form>
<form id="courses.course2.section2.examSlot"> .. </form>
<form id="courses.course2.section2.num"> .. </form>
<form id="courses.course2.section2.activity"> .. </form>

<form id="defaultQueryPattern">
   <block>
      <prompt> please search by course identification,
         please say an identification number </prompt>
      <assign name="formID" expr="'defaultQueryPattern'" />
   </block>
   <help> For example, course 100 say 100, course 254 say 254
   </help>
   <field name="search" type="number" >
      <grammar> 100 | 254 </grammar>
      <filled>
      <if cond="search=='254'">
      <goto next="#courses.course2.all"/>
      <elseif cond="search=='100'" />
            <goto next="#courses.course1.all"/>
      <else /> <throw event="noSuchCourse" />
      </if>
      <submit next="#door" namelist="formID " />
      </filled>
      (active-commands process omitted)
   </field>
   <catch event="noSuchCourse"> no such course included
```

```
        in the document
        <submit next="#door" namelist="formID " />
     </catch>
  </form>

  <form id="door">
     <field name="confirm" type="boolean">
        <prompt> Do you want to continue? </prompt>
        <filled>
        <if cond="confirm">
           <goto expr="'#'+ formID" />
        <else />  Thanks for your attention! <exit />
        </if>
        </filled>
     </field>
  </form>

</vxml>
```

Appendix -6 VXML data access application

BIBLIOGRAPHY

[Agarwal, 2000] R. Agarwal, Y. Muthusamy, V. Viswanathan, "Voice browsing the web for information access", *9th International WWW Conference*, Amsterdam, 2000.

[Allen, 1995] J. F. Allen, G. Ferguson, B. W. Miller, E. Ringger, "Spoken dialogue and interactive planning", *Proceedings of the ARPA Spoken Language Technology Workshop*, Austin, Texas, pp7-48.

[Aust, 1995] H. Aust, M. Oerder, F. Seide, V. Steinbiss, "The Philips automatic train timetable information system", *Speech Communication* 17, 249-262.

[Badros, 2000] G. J. Badros, "JavaML: a markup language for Java source code", *9th International WWW Conference*, Amsterdam, 2000.

[Bonifati, 2001] A. Bonifati, D. Lee, "Technical Survey of XML Schema and Query Languages", 2001.

[Bosak, 1997] J. Bosak, "XML, Java, and the future of the web", *Sun Microsystems*, March 1997, http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.htm.

[Boyce, 1996] S. Boyce, A.L. Gorin, "User interface issues for natural spoken dialogue systems", *Proceedings of International Symposium on spoken dialogue, ISSD*, 1996, pp65-68.

[Chamberlin, 2000] D. Chamberlin, J.Robie, D. Florescu, "Quilt: an XML query language for heterogeneous data sources", *In Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, May 2000.

[Deutsch, 1999] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for XML", *International World Wide Web Conference*, 1999.

[Frost1, 1999] R. A. Frost, "SpeechNet: A network of hyperlinked speech-accessible objects", *Proceedings of the IEEE WECWIS International Workshop on Advanced Issues of E-Commerce and Web-based Information System.* San Jose, April 1999, pp. 71-76.

[Frost2, 1999] R. A. Frost, "A natural language speech interface constructed entirely as a set of executable specifications", *Proceedings of the AAAI '99 Intelligent Systems Demonstrations,* Orlando July 1999.

[Frost, 1998] R. A. Frost, T. HADDAD, "Engineering and re-engineering a speech interface to the Web", *Proceedings of the 9th International Conference on Computing and Information, ICCI'98*, University of Manitoba, June 1998.

[Frost, 1995] R. A. Frost, "Use of executable specifications in the construction of speech interfaces", *Proceedings of the IJCAI Workshop on Developing AI Applications for the Disabled,* Montreal 1995.

[Frost, 1994] R. A. Frost, "W/AGE the Windsor attribute grammar programming environment", *Schloss Dagstuhl International Workshop on Functional Programming in the Real World.* 1994.

[Frost, 1992] R. A. Frost, "Constructing programs as executable attribute grammars", *the Computer Journal* 35(4): 376-389.

[Goose, 2000] S. Goose, M. Newman, C. Schmidt, L. Hue, "Enhancing web accessibility via the Vox Portal and a web hosted dynamic HTML<-> VoxML converter", *9th International WWW Conference*, Amsterdam, 2000.

[Hakkinen, 1997] M. Hakkinen, "Issues in Non-Visual Web browser design: pwWebSpeak", *Proceedings of the 6th International World Wide Web Conference,* April 1997.

[Hemphill, 1997] C. Hemphill, Y. K. Muthusamy, "Developing Web-based Speech Applications", *Proceedings of Eurospeech '97,* September 1997, Vol. 2, pp.895-898.

[Holman, 1999] K. G. Holman, "The XML family of standards", *XML Finland'99: SGML Users Group Finland,* September 23, 1999.

[Hunt, 2000] A. Hunt, W. Walker, " A fine grained component architecture for speech application development", *Sun Microsystems Laboratories*, Burlington MA, USA.

[IBM, 2000] "IBM WebSphere Voice Server Software Developers Kit (SDK) Programmer's Guide", 2000, http://www-4.ibm.com/software/speech/.

[Ide, 2000] N. Ide, P. Bonhomme, L. Romary, " XCES: an XML-based encoding standard for linguistic corpora", *LREC 2000 2nd International Conference on Language Resources & Evaluation*, 2000.

[Klemmer, 2000] S. R. Klemmer, A. K. Sinha, J. Chen, J. A. Landay, N. Aboobaker, A. Wang, "SUEDE: A Wizard of Oz Prototyping Tool for Speech User Interfaces", *the 13th ACM Symposium on User Interface Software and Technology: UIST 2000.*

[Krell, 1996] M. Krell, D. Cubranic, "V-Lynx: bring the World Wide Web to sight impaired users", *International ACM Conference on Assistive Technologies*, April 1996, pp. 23-26.

[Lau, 1997] R. Lau, G. Flammia, C. Pao, and V. Zue, "WebGALAXY: beyond point and click: a conversational interface to a browser", *Proceeding of the 6th International World Wide Web Conference*, April 1997.

[Lucas, 1999] B. Lucas, W. Walker, A. Hunt, "ECMAScript Action Tags for JSGF", *documentation of IBM Voice Server SDK*, September 1999.

[Martin, 2000] D. Martin, "Adapting Content for VoiceXML", http://www.xml.com/pub/a/2000/08/23/didier/.

[Motorola Mix] Motorola Mix system, http://mix.motorola.com.

[Motorola Vox] Motorola, VoxML, http://www.motorola.com, Mobile Application Development Kit.

[RefsnesData] "XML Attributes", http://www.w3schools.com/xml/xml_attributes.asp.

[Robie, 1999] J. Robie, editor, "XQL (XML Query Language)", August 1999, Available online at http://metalab.unc.edu/xql/xql-proposal.html.

[Rollins, 2000] S. Rollins, N. Sundaresan, "AVoN Calling: AXL for Voice-enabled web Navigation", 9th *International WWW Conference*, Amsterdam, 2000.

[SAX, 1999] Megginson Technologies, SAX 1.0: The simple API for XML, 1999, http://www.megginson.com/SAX.

[SUN JSAPI] "Java Speech API Programmer's Guide", http://java.sun.com/products/java-media/speech.

[SUN JSGF] "Java Speech Grammar Format Specification v1.0", October 1998, http://java.sun.com/products/java-media/speech.

[SUN JSML] "Java Speech Markup Language Specification v0.5", August 1997, http://java.sun.com/products/java-dedia/speech/forDevelopers/JSML/index.html.

[VoiceServerSDK] IBM WebSphere Voice Server SDK development toolkit, http://www.alphaworks.ibm.com.

[VRS] Voice Recognition Software: Comparison and Recommendations, http://www.io.com.

[VxmlOrg, 2000] "Voice extensible Markup Language VoiceXML 1.0 Specification", *VoiceXML Forum*, March 2000. http://www.voicexml.org.

[W3CGDTD] Guide to the W3C XML specification ("XMLspec") DTD, Version2.1, http://www.w3.org/XML/1998/06/xmlspec-report.

[W3C MathML] D. Carlisle, P. Ion, R. Miner, N. Poppelier, "Mathematical Markup Language (MathML) 2.0 Specification", February 2001, http://www.w3.org/TR/MathML2.

[W3C Voice] Voice Browser Activity, http://www.w3.org/Voice/.

[W3C XML] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, "Extensible Markup Language 1.0, Second Edition", Oct. 2000, http://www.w3.org/TR/REC-xml.

[W3C Schema] D. C.Fallside, H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, P.V. Biron, A. Malhotra, "XML Schema Part0: Primer, Part1: Structures, Part2: Datatypes", http://www.w3.org/XML/Schema, May 2001.

[W3C XPath] J. Clark, S. DeRose, XML Path Language (XPath) Version 1.0, November 1999, http://www.w3.org/TR/xpath.

[W3C Dom] A.L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Chapmion, S. Byrne, Document Object Model (DOM) Level 2 Specification – Core Specification, http://www.w3.org/DOM/, Nov. 2000.

[W3C XSLT] J. Clark, XSL Transformation (XSLT) 1.0, http://www.w3.org/TR/xslt, 1999.

[W3C XQuery] "XQuery: a query language for XML", W3C Working Draft, work in progress, *World Wide Web Consortium*, February 2001, http://www.w3.org/TR/2001/WD-xquery-20010215.

[WroxJ2EE, 2000] "Professional Java Server Programming", Wrox Press Ltd., 2000.

[WroxJava, 2001] "Professional XML and Java", Wrox Press Ltd. 2001.

[WroxXML, 2001] "Professional XML", Wrox Press Ltd. 2001.

[W3C XHTML] "XHTML 1.0: The Extensible HyperText Markup Language – A Reformulation of HTML 4 in XML 1.0", January 2000, http://www.w3.org/TR/xhtml1.

[Zajicek, 1998] M. Zajicek, C. Powell, C. Reeves, "A Web navigation tool for the blind", *3rd ACM/SIGAPH on Assistive Technologies*, California, 1998.

[Zue, 2000] V. Zue, S. Seneff, J. Glass, J. Polifroni, C. Pau, T. J. Hazen, L. Hetherington, "Jupiter: A telephone-based conversational interface for weather information", *IEEE Transactions on Speech and Audio Processing*, Vol. 8, No. 1, 2000.

VITA AUCTORIS

Hong Ying (Hydi) Dou originally came from Beijing, China. She graduated from Yuying High School in 1987. From there, she went to the Computer Institute of Beijing Polytechnic University where she obtained a B.Sc. in Computer Science in 1991. After that, Hydi went to work, as a software engineer in People's Bank of China. In 1999, She went to the University of Windsor to pursue her Master's degree in Computer Science. Hydi is currently a candidate for the Master's degree in Computer Science at the University of Windsor.