2002

# Building complex language processors in VoiceXML.

Xuejun. Liu
*University of Windsor*

# Building Complex Language Processors in VoiceXML

By

Xuejun Liu

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

Windsor, Ontario, Canada
2002

0-612-75796-X

# Abstract

VoiceXML was accepted by the World Wide Web Consortium (W3C) as a standard XML-based markup language for distributed Web-based voice services. It was designed to provide a way for web developers to use a familiar markup style to deliver voice content to the Internet. Grammars are used to specify the words and patterns of words that a user can say at any particular point in a dialog.

However, in most current applications, VoiceXML is used as a simple language processor used to convert speech to text. In this thesis, we investigate the expressive power of "pure" VoiceXML and the expressive power of Java Speech Grammar Format (JSGF) tagging mechanism, ECMAScript and server-side processing. In order to build complex language processors in VoiceXML, we give a combination of VoiceXML JSGF tagging, ECMAScript and server-side processing. The thesis work is concerned with the ability to use VoiceXML to define semantics as well as syntax. A prototype has been implemented to demonstrate the efficiency of different approaches.

**Keywords:** VoiceXML, language processor, grammar, JSGF, tagging, ECMAScript, voice application

***To My Wife and Daughter***

# Acknowledgements

First and foremost, I would like to express my respect, appreciation and thanks to my advisor, Dr. Richard A. Frost, for all of his advice, enthusiasm, and patience in guiding me through completion of my Master thesis. Both his time and understanding have been invaluable to me.

I would also like to thank Dr. Peter Tsin and Dr. H. K. Kwan for their valuable advice and comments on this thesis work. I would like to extend my thanks to Dr. for his chairing my thesis defense presentation.

In addition, I would like to thank Mr. Walid Mnaymneh for his technical support. Many thanks go to the graduate students for their help, too. My special thanks go to helpful secretaries of the Department of Computer Science Ms. Mary Mardegan and Ms. Margaret Garabon.

Finally, I would like to thank my wife Xun Luo and my daughter Patricia Liu for their encouragement, understanding, and support. They are what make it all worthwhile.

# TABLE OF CONTENTS

## List of Figures

# Chapter 1

# Introduction

This chapter mainly gives the introduction to this thesis work. It analyzes the problems in building voice applications. It also gives the solutions to build complex language processors in VoiceXML. Finally, our thesis statement is given, which is "Complex language processors can be built in VoiceXML (possibly with the need to use it in combination with some other technologies)."

## 1.1 Problems

Speech technology has improved significantly over the past few years. The first reason for voice application was for visually impaired people [Boyce1996]. The need for speech also comes from several aspects:

- The needs for accessing web contents and services anywhere and anytime. In most cases, you don't have a PC to access Internet, but you do have access to a telephone or mobile phone. The voice applications are perfect fit in with this situation.
- A hands-free and eyes-free interaction through voice commands offers convenience to wireless phone users.
- The need for multi-modal interfaces and customer services. The rapid pace of business today requires employees and customers to have fast, constant access to

information. Voice applications can eliminate the constraints of the telephone keypad and provide easy-to-use services.

Although voice applications are becoming popular, they are still difficult to build. The technology expertise required to build speech user interfaces prevents many individuals from participating in the speech interface design process. There is need to have a new technology in building voice applications. VoiceXML is emerged to remedy these problems. However, VoiceXML does not provide enough expressive power. There is a need to extend the expressive power of VoiceXML.

## 1.2 Observations

Spoken language systems are starting to become commonly used in applications ranging from automated answering services to stock trading to weather reporting. VoiceXML [W3C Voice] is designed to extend the existing web environment by providing another way of accessing distributed information and services. There are many similarities between the HTML web world and the VoiceXML web world. Voice browsers present information to the user through VoiceXML, as web browsers present information to the user through HTML.

The primary goal of VoiceXML is to bring the power of web development and content delivery to voice application. It was design to provide a way for web developers to use a familiar markup style and existing web server-side logic to deliver voice content to the

Internet [W3C VXML]. It is easier to develop and deploy than proprietary speech web applications.

However, VoiceXML is only a simple language processor that provides recognizer functions and some translator functions. This limits the power of VoiceXML.

Our goal was to investigate the expressibility of VoiceXML. The investigation will provide techniques to help people build complex language processors in VoiceXML.

First of all, VoiceXML is becoming a standard for developing voice web application. More and more companies provide VoiceXML development tools. More complicated applications are built in VoiceXML. There is a need to have more powerful expression of VoiceXML.

Complex language processors can give more powerful expressions. This makes it easier to develop complicated voice applications. These complex processors are not only used for black box creation of systems by non-experts, but they can also scale to more complex systems.

The first is that language processors should be easy to use. Since our motivation was to enable complicated voice application be built using these language processors. This made it necessary to hide as much of the underlying complexity as it could.

Second, language processors need to be flexible. It can be portable to any VoiceXML browser that support VoiceXML. We achieve this by using Java speech Grammar Format (JSGF) tagging mechanism [SUN JSGF] [Lucas 2001] and ECMAScript [ECMA Script] [Lucas 2001]. We also allow the construction of an arbitrary server-side computing [ASP] [CGI] [SUN JSP] [SUN Servlet].

Finally, language processors need to be efficient. There is a trade-off between placing computing on server-side or on client-side.

## 1.3 Solutions

The observations above lead to following possible solutions:

- A combination of VoiceXML with JSGF tagging.
- A combination of VoiceXML with ECMAScript.
- A combination of VoiceXML with JSGF tagging and ECMAScript.
- A combination of VoiceXML with server-side processing.
- A combination of VoiceXML with JSGF tagging and server-side processing.
- A combination of VoiceXML with JSGF tagging, ECMAScript and server-side processing.

## 1.4 The Thesis Statement

The thesis statement is:

"Complex language processor can be built in VoiceXML (possibly with the need to use it in combination with some other technologies)."

The following work shows that how the thesis will be supported:

- Investigate expressive power of VoiceXML.
- Investigate expressive power of JGSF grammar.
- Analyze a combination of VoiceXML with JSGF tagging.
- Analyze a combination of VoiceXML with ECMAScript.
- Analyze a combination of VoiceXML with server-side processing.
- Analyze a combination of VoiceXML with JSGF tagging, ECMAScript and server-side processing.
- Implement a prototype system.
- Analyze the results and give suggestions on future works.
- Draw some conclusions as to what extent the thesis statement is true.

## 1.5 Organization of the Thesis Report

This thesis consists of eight chapters. Chapter two discusses the details of VoiceXML. Chapter three mainly discusses JSGF grammar and tagging mechanism. Chapter four briefly introduces ECMAScript and some server-side processing techniques. Chapter five focuses on the approaches to build complex language processors in VoiceXML with JSGF tagging, ECMAScript and server-side processing. Chapter six gives the implementation of the prototype. Chapter seven analyzes the combinations of VoiceXML with JSGF tagging, ECMAScript and server-side processing. Chapter eight gives related works that have been done on voice applications. Chapter nine gives some conclusions on this thesis and points out the future work.

# Chapter 2

# VoiceXML

Although traditional Interactive Voice Response (IVR) applications have been deployed in enterprises for decades, but they've faced poor usability and the inability to go beyond providing access to proprietary information. VoiceXML is an open standard for developing voice web applications.

## 2.1 An Overview of VoiceXML

VoiceXML is a language for creating voice-user interfaces including applications involving speech recognition and DTMF keypad for input, and pre-recorded audio and text-to-speech synthesis (TTS) for output. It is based on the Worldwide Web Consortium's (W3C's) Extensible Markup Language (XML), and leverages the web paradigm for application development and deployment. By having a common language, application developers, platform vendors, and tool providers all can benefit from code portability and reuse [W3C VXML].

With VoiceXML, speech-recognition application development is greatly simplified by using familiar web infrastructure, including tools and Web servers. VoiceXML has features to control audio output; audio input; presentation logic and control flow; event handling; and basic telephony connections [W3C VXML].

VoiceXML does not support unconstrained speech recognition: the ability to listen to any speech in any context and transcribe it accurately. To achieve reasonable recognition accuracy and response time, VoiceXML constrain what they listen for by using grammars.

The role of grammars in a spoken dialog application is to define for the VoiceXML browser the words and patterns of words that a user can say at any particular point in a dialog [W3C VXML]. Grammar authoring is a critical facet in the development of robust, usable and complex speech applications. When an application's grammars accurately model the speech input from callers, the usability of the application is enhanced and caller satisfaction is likely to be higher.

The VoiceXML 1.0 specification documents the use of the Java Speech Grammar Format (JSGF) [SUN JSGF] to describe grammars but does not mandate that browsers support JSGF. Current deployments of VoiceXML and other speech applications most often use proprietary grammar formats embodied in the browser.

There are several significant advantages to build and deploy VoiceXML-based applications, in place of proprietary voice applications.

- Deliver web content and services through telephone.
- Leverage existing Internet infrastructure and legacy systems.

- Ensure portability across implementation platforms.
- Decrease the level of expertise required to create voice applications.
- Enable rapid voice application development, similar to HTML for the web.
- Provide "Voice View" for web content.

## 2.2 Evolution of VoiceXML

VoiceXML combines proven technologies in speech recognition, telephony, and Internet services. Companies such as IBM, Motorola, and AT&T have been researching the simple declarative markup languages for voice applications for more than five years [W3C Voice]. There is an overwhelming trend in enterprise deployments to more deeply embrace technologies such as XML, XSL, and HTTP as a universal transport. This paradigm allows companies to preserve flexibility and work more efficiently by cleanly separating data from the user interface. Developers build shared business logic once, then use standardized markup languages such as HTML, VoiceXML, and WML to create the appropriate user interface for each device. VoiceXML is simply a committed by technology leaders to adopt a universal open standard for voice web applications [W3C VForum].

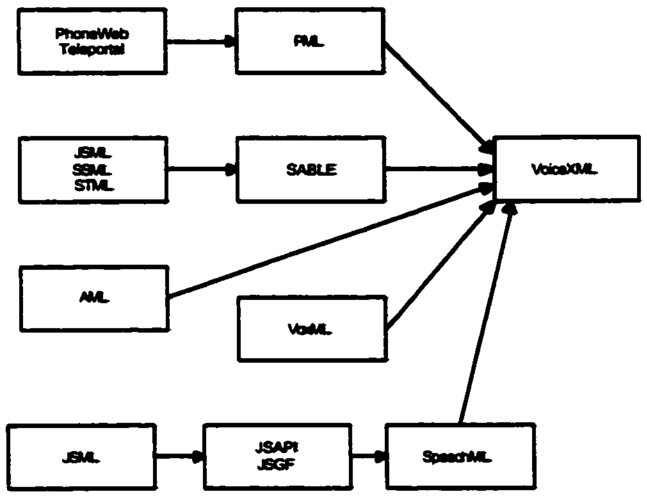

**Figure1: Evolution of VoiceXML**

## 2.3 Architecture of VoiceXML

The basic architecture of a VoiceXML system is shown in Figure 2.

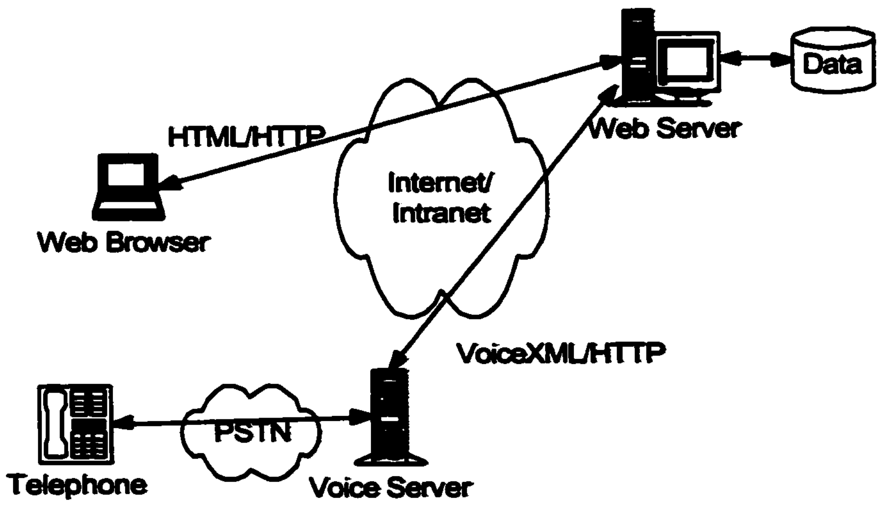

**Figure2. Architecture of VoiceXML system**

VoiceXML introduces a new way of presenting the web information. Instead of presenting the information visually, the voice server presents the information to the caller using VoiceXML. A customer calls a designated phone number. The call is answered by a Voice Server. The Voice Server sends an HTTP request to the Web Server, which may access the back-end infrastructure. The Web Server returns an initial VoiceXML document to Voice Server. The Interpreter of Voice Server parses and executes this document playing prompt, hearing responses and passing them on to a speech recognition.

When all the necessary responses have been collected from the user, the interpreter assembles them into a request to the Web Server. The Web Server responds with a dynamically generated VoiceXML page containing the information requested by the user. The processor can be repeated indefinitely to produce a conversation between the user and the Voice Server.

## 2.4 Components of a VoiceXML System

Figure 3 shows a generic block diagram of a VoiceXML system [W3C VXML].

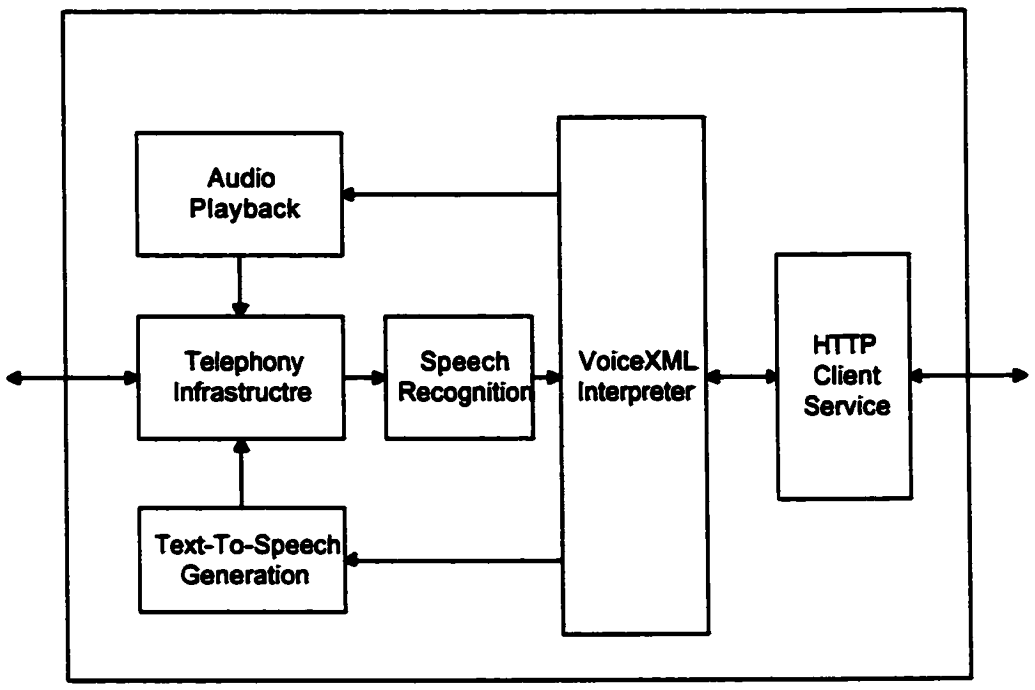

**Figure 3. Components of a VoiceXML System**

### 2.4.1 Telephony Infrastructure

Telephony infrastructure is used to answer incoming calls and initiate voice sessions. It delivers voice input from the user via telephone network to speech recognition engine. Similarly, this component delivers audio playback or speech synthesized by text-to-speech engine to the user.

## 2.4.2 Speech Recognition

Speech recognition is the process of converting spoken input to text. The speech recognition process is performed by speech recognition engine. The primary function of the speech recognition engine is to process speech input and translate it into text that an application understands.

Speech recognition can be divided into several phases. In signal processing the incoming audio frequencies are analyzed. In phoneme recognition the incoming audio pattern are compared to language phoneme. Then the frequencies of phonemes are compared to words and patterns of words defined by the recognition grammar. The final phase is to provide the interested applications information that has been gathered about the incoming audio.

Speech recognition engine processes an utterance and returns a result. The speech recognition engine tries very hard to match the utterance to a word or phrase in the active grammar. Sometimes the match may be poor because the caller said something that the application was not expecting, or the caller spoke indistinctly. In these cases, the speech recognition engine returns the closest match, which might be incorrect. A confidence score is returned along with the text to indicate the likelihood that the returned text is correct.

### 2.4.3 VoiceXML Interpreter

Once a call is received, the VoiceXML interpreter fetches VoiceXML document, parses through and executes the instructions in the VoiceXML documents. The VoiceXML interpreter manages the dialog between the application and the user by playing audio prompts, accepting user inputs, and acting on those inputs. The action might involve jumping to a new dialog, fetching a new document, or submit user input to the web server for processing.

### 2.4.4 Text-To-Speech (TTS) Generation

Speech synthesis, or text-to-speech is the process of converting text to speech. Speech synthesis can be divided into several steps. Structure analysis finds sentences, paragraphs and other structures that are common to all language. Text preprocessing is performed to find special constructs like abbreviations, dates, time and other constructs that are usually specific for each language and even different situations. After the first two steps the text has been structured into a spoken form. Then it can be converted to speech.

Text-to-phoneme conversion converts each word to phoneme. Prosody analysis processes the sentences into a more sophisticated form. Pitch, timing, pauses, emphasis and other features that affect the way sentences are interpreted in human brains are processed.

The last step is waveform production. All sentences are produced into audio waveforms according to the phoneme and prosody information acquired in previous steps. Waveform production can be performed using concatenation of pieces of recorded human speech or through format synthesis using signal-processing technique.

### 2.4.5 Audio Playback

Audio playback outputs prerecorded audio files to produces natural speech output.

### 2.4.6 HTTP Client Service

The voice server also serves as an HTTP client sending messages and receiving VoiceXML documents from the web server. Communications between the voice server and the web server follow standard HTTP protocols. Outgoing requests are in the form of an HTTP "get" or "post" command.

## 2.5 VoiceXML Format

VoiceXML is an XML language and can produces well-formed programs with consistent structure [W3C VXML].

VoiceXML code is contained in a document. A VoiceXML application is a collection of VoiceXML documents. The main document is the current (active) VoiceXML document. The documents in a VoiceXML application share the same application root document. The application root document is loaded whenever any document within the application is active [W3C VXML].

A VoiceXML application defines a series of dialogs between a user and a computer. Each VoiceXML document forms a conversational finite state machine. The user is always in one conversational state, or dialog, at a time [W3C VXML].

Each dialog determines the next dialog to which to transition. Transitions are specified using URIs, which define the next document and dialog to use. If a URI does not refer to a document, the current document is assumed. If it does not refer to a dialog, the first dialog in the document is assumed. Execution is terminated when a dialog does not specify a successor, or if it has an element that explicitly exits the conversation.

There are two types of dialogs: forms and menus. Forms define an interaction that collects values for a set of field item variables. A menu presents the user with a choice of options and then transitions to another dialog based on the choice.

A subdialog provides a mechanism for invoking a new interaction and returning to the original form.

A dialog is managed through the use of grammars, prompts, events, and control elements such as gotos, links, submits, and the like.

### 2.5.1 Prompts

The prompt element controls the output of synthesized speech and prerecorded audio [W3C VXML]. Any text within the body of a <prompt> element is spoken.

```
<prompt>
        <audio src="http://speech88.cs.uwindsor.ca/voice/welcome.wav" >
                Welcome to speech web.
        </audio>
        Please say Weather, Travel, or Bank.
</prompt>
```

This prompt example first plays the audio file. If the audio file cannot be played, "Welcome to speech web" is output instead. Then "Please say Weather, Travel, or Bank" is output.

### 2.5.2 Menus

A menu prompts the user to make a choice and transitions to different places based on that choice [W3C VXML]. The following menu offers the user three choices:

```
<menu>
  <prompt> Welcome to speech web. Say one of: <enumerate/></prompt>
  <choice next="http://speech88.cs.uwindsor.ca/weather.vxml">
        Weather
  </choice>
  <choice next="http://speech88.cs.uwindsor.ca/travel.vxml">
```

```
          Travel
      </choice>
      <choice next="http://speech88.cs.uwindsor.ca/bank.vxml">
          Bank
      </choice>
      <noinput>Please say one of <enumerate/></noinput>
    </menu>
```

This dialog might proceed as follows:

C: Welcome to speech web. Say one of : Weather; Travel; Bank.

U: Home.

C: I did not understand what you said.

C: Welcome to speech web. Say one of : Weather; Travel; Bank.

U: Weather.

C: (proceeds to http://speech88.cs.uwindsor.ca/weather.vxml)

### 2.5.3 Forms

Forms are the key component of VoiceXML documents. A form is a collection of one or more fields that the caller fills in by saying something [W3C VXML].

```
  <form id="travel">
    <block>Welcome to the travel information service.</block>
    <field name="state">
          <prompt>What state?</prompt>
          <grammar src="state.gram" type="application/x-jsgf"/>
          <catch event="help">
                Please speak the state you want to travel.
          </catch>
    </field>
    <field name="city">
          <prompt>What city?</prompt>
```

```
            <grammar src='city.gram" type="application/x-jsgf"/>
            <catch event="help">
                    Please speak the city you want to travel.
            </catch>
    </field>
    <block>
            <submit next="/servlet/travel"  namelist="city state"/>
    </block>
  </form>
```

This dialog may proceeds as follows:

C : Welcome to the Travel information service. What state?

U: Guangdong

C: I did not understand what you said. What state?

U: Help

C: Please speak the state you want to travel.

U: Ontario

C: What city?

U: Windsor

C: (proceeds to /servlet/travel)

### 2.5. 4 Subdialog

A subdialog is an entire form that is executed, the result of which is used to provide one input field to another form. Subdialogs can be used to provide a disambiguation or conformation dialog, as well as to create reusable dialog components for data collection and other common tasks [W3C VXML].

In the example below, the birthday of an individual is used to validate their driver's license. The src attribute of the subdialog refers to a form that is within the same document. The <param> element is used to pass the birthday value to the subdialog.

```
<!-- form dialog that calls a subdialog -->
<form>
  <subdialog name="result" src="#getdriverslicense">
   <param name="birthday" expr="'2001-08-20'"/>
   <filled>
      <submit next="http://speech88.uwindsor.ca/servlet/"/>
   </filled>
  </subdialog>
</form>

<!-- subdialog to get drivers license -->
<form id="getdriverslicense">
  <var name="birthday"/>
  <field name="drivelicense">
   <grammar src="http://grammarlib/drivegrammar.gram"
          type="application/x-jsgf"/>
  <prompt> Please say your driver's license. </prompt>
  <filled>
    <if cond="validdrivelicense(drivelicense,birthday)">
      <var name="status" expr="true"/>
    <else/>
      <var name="status" expr="false"/>
    </if>
    <return namelist="drivelicense status"/>
  </filled>
 </field>
</form>
```

The driver's license value is returned to calling dialog, along with a status variable in order to indicate whether the license is valid or not.

## 2.5.5 Grammars

In VoiceXML, A <grammar> element is used to define what the user can say at any given time [W3C VXML]. The VoiceXML 1.0 specification documents the use of the Java Speech Grammar Format (JSGF) to describe grammars. There are three different types of grammars supported in VoiceXML:

- Inline
- External
- Built-in

Inline grammars are those that are defined right in VoiceXML documents. For example:

```
<grammar>
Toronto |  Windsor |  London | Cambridge
</grammar>
```

External grammars are specified externally from the VoiceXML document in another file and are referenced from within the VoiceXML document. For example:

```
<grammar> src="city.gram" type="application/x-jsgf"></grammar>
```

This snippet is referencing the external grammar file city.gram.

```
#JSGF V1.0;

grammar city;

public <citys> = Toronto |  Windsor |  London | Cambridge;
```

Built-in grammars are provides for common field types so that they can be used directly. There are several built-in grammars:

- Boolean
- Currency
- Date
- Digits
- Number
- Phone
- Time

Built-in grammars are specified using the type attribute of the <field> element.

```
<field name="birthday" type="date">
        <prompt> How old are you?</prompt>
</field>
```

# Chapter 3

# Java Speech Grammar Format

VoiceXML uses grammars to constrain what they listen for to achieve reasonable recognition accuracy and response time. The VoiceXML 1.0 specification documents the use of the Java Speech Grammar Format (JSGF) [SUN JSGF] to describe grammars but does not mandate that browsers support JSGF.

JSGF defines a platform-independent, vendor-independent way of describing rule-based grammars. The logical structure of the grammar is captured by a combination of traditional BNF (Backus-Naur Form) and a regular expression language following some of the conventions of the syntax of java programming language [SUN JSGF].

## 3.1 JSGF Format

Grammars are based on a set of rules that define which groups of words are legal. A grammar is defined in a single file. It has a unique name and consists of a grammar header and a grammar body [SUN JSGF].

The grammar header contains a self-identifying header and declares the name of the grammar. The declare is also the import of public rules of other grammars. An imported rule can be referenced locally using its unqualified name if the name is unambiguous [SUN JSGF].

The grammar body defines the rules of the grammar. Each rule has a unique name within its grammar and only one definition. The ordering of the rules is not relevant.

Rules are defined by tokens, references to other rules, and their logical combinations. Tokens represent the actual words that may be spoken to the recognizer. Tokens are separated by white spaces and are usually single words. A token is a reference to an entry in the recognizer's vocabulary or lexicon which defines the pronunciation of the token [SUN JSGF].

There are two patterns for rule definitions.

```
<ruleName> = ruleExpansion;
public <ruleName> = ruleExpansion;
```

A simple rule consists of a rule name and a token or a reference to another rule. For example:

```
<city> = Windsor;
<state> = <com.windsor.grammar.state>;
```

More advanced rules can be formed by composition, grouping and unary operations.

Rules can compose of sequences of rule expansions. A sequence defines an accepted order of words. For example:

```
<greeting> = Please;
<command> = <greeting> open the door;
```

The rule can also define a set of alternatives. A portion of speech that matches any of the rule alternatives matches the rule. For example:

```
<city> = Windsor | Toronto | London;
```

In the previous example the user can say any of the three alternatives above to match the rule city.

Weights can be used to indicate the likelihood of different alternatives. Following example defines the likelihood of each city before the word:

```
<city> = /0.6/ Windsor |  /0.3/ Toronto | /0.1/ London
```

Parenthesis can be used to override precedence and brackets to indicate optional content. Unary operators ( * and + ) indicate if a word can be spoken zero or more times, or one or more times. For example:

```
<command> = Please* (open | close) [the] (door | window) ;
```

In the previous example, the word "please" can be spoken zero or more times before one of the words "open" or "close". The word "the" is optional. It may or may not be spoken before the word "door" or "window".

JSGF allows right recursion. In the right recursion, the rule refers to itself as the last part of its definition. For example:

```
<exp> = <digit> | <digit> <op> <exp>;
```

However, JSGF does not support other forms of recursion, such as left recursion and embedded recursion. JSGF is a regular grammar with some features of a context-free grammar.

## 3.2 JSGF Tags

Tags provides a mechanism for grammar writers to attach application-specific information to rule definitions. These tag attachments do not affect recognition of a grammar. Instead, the tags are attached to the result object returned by the recognizer to the application [SUN JSGF]. Tags are useful in more complex applications and in more sophisticated dialog applications, in which any utterance may simultaneously provide several pieces of information to the application.

Tags are used to associate attributes with grammar rules. An attribute can represent a string, a number, a type or whatever. Each grammar symbol can have an associated set of attributes, partitioned into two subsets calls the synthesized and inherited attributes of that grammar symbol [Lucas 2001].

The value of an attribute at a parse-tree node is defined by a semantic rule associated with the production used at that node. The value of a synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree. The value of an inherited attribute is computed from the values of attributes at the siblings and parent of that node [Lucas 2001].

A parse tree can have non-terminals, terminals, tags and sequences from the original grammar that correspond to the content of the utterance, and in which a separate copy of each non-terminal has been made for each use of the non-terminal [Lucas 2001]. The parse tree evaluation is to recursively compute a value for each non-terminal in the tree.

The value of each non-terminal is an object. In order to provide a simple value, such as a number or string, a special field this.$value is used in the non-terminal's value object.

The default value for this.$value is a string which is the concatenation of the string values of all the items in the non-terminal, separated by spaces.

The object of a child non-terminal may be used in the enclosing parent non-terminal to construct its attribute value by referring to child non-terminal using the variable $rulename. The attribute value can be referred by using dot operator followed by attribute name like $rulename.attributename.

The following example explains the usage of JSGF tags.

Consider a form for transfer amount between different accounts. A directed dialogue conversation might proceed as follows:

C: What kind of account you want to transfer from?

U: Saving Account

C: Account number?

U: 01368

C: What kind of account you want to transfer into?

U: Checking Account

C: Account number?

U: 01366

C: How much money do you want to transfer?

U: 500 dollars

In contrast, a somewhat more natural dialogue might proceed as following :

C: How can I help you?

U: I want to transfer from saving account 01368 to checking account 01366 500 dollars.

The grammar for this dialog using JSGF tags is as following:

```
< types> = ( saving  [ account ] ) { this.type= "saving" }  |
           ( checking  [ account ] ) { this.type = "checking"} |
           ( credit  [card ] ) { this.type = "credit" };
<account> = <digits>;
<amount> = <money>;
<transfer> = [ I want to | I would like to ] transfer from
               <types>  {this.fromType = $types.type}
               <account> {  this.from = $account} to
               <types>  {this.toType = $types.type}
                <account>  {this.to = $account}
                <amount>   {this.amount = $amount;};
```

This grammar illustrate:

- Define attribute and assign values to them.
- Reference to the value of a non-terminal by using $account, $amount.
- Reference to the attribute value of a non-terminal by using $types.type.

The utterance "I want to transfer from saving account 01368 to checking account 01366 500 dollars", when parsed against above grammar, produce the following parse tree:

```
<transfer>
    I
    want
    to
    transfer
```

```
from
<types>
    saving account
    {this.type = "saving"}
{this.fromType = types.type}
<account>
    <digits>
        01368
        {this.$value = "01368"}
    {this = $digits}
{this.from = $account}
to
<types>
    checking account
    {this.type = "checking"}
{this.toType = types.type}
<account>
    <digits>
        01366
        {this.$value = "01366"}
    {this = $digits}
{this.to = $account}
<amount>
    <money>
        500 dollars
        {this.$value = "500"}
    {this = $money}
{this.amount = $amount}
```

When this parse tree is evaluated, the ECMAScript value {fromType: "saving", from: "01368", toType: "checking", to: "01366", amount: "500"} is returned.

The following is VoiceXML document which uses attribute values returned:

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

  <var name = "fromType"/>
  <var name = "fromAccount"/>
  <var name = "toType"/>
  <var name = "toAccount"/>
  <var name = "amount"/>
```

```
<form id="form1">
  <field name="transfer">
    <prompt>  How can I help you? </prompt>
    <grammar src="http://www.uwindsor.ca/bank/transfer.gram"
        type="text/jsgf"/>
  </field>
  <filled>
     <assign name = "fromType" expr = "transfer.fromType"/>
     <assign name = "fromAccount" expr = "transfer.from"/>
     <assign name = "toType" expr = "transfer.toType"/>
     <assign name = "toAccount" expr = "transfer.to"/>
     <assign name = "amount" expr = "transfer.amount"/>
     <submit next= "/servlet/Transfer"
               namelist = "fromType fromAccount toType toAccount amount"/>
  </filled>
 </form>

</vxml>
```

## 3.3 Building an Attribute Grammar in VoiceXML with JSGF Tagging

W/AGE [Frost 1994 ] enables language recognizer, parser and evaluators to be built as executable specification of attribute grammars. Each grammar production is implemented as syntax-directed evaluator using a functional variation of classical top-down parsing with back-tracking. Left-recursive productions are supported. W/AGE programs are expressive and complete declarative.

Compare to W/AGE, JSGF does not support left recursion and top-down parsing with back-tracking. Though there is no same expressive power as W/AGE, Attribute grammars can be built in VoiceXML with JSGF tagging. Attributes can be manipulated in JSGF grammar with some operations and desired result is returned. It is simpler and less power to manipulate attributes in JSGF than W/AGE. But we can combine ECMAScript to have the same expressive power.

## 3.4 Embedding Computation in JSGF Grammar

ECMAScript basic computing functions, such as addition, subtraction, multiplication and division, can be embedded in JSGF grammar. The attributes of JSGF grammar can have the type of number besides string. But the power of JSGF is limited by not allowed to call any ECMAScript functions.

The following grammar adds computation in JSGF.

```
<digit> = <1to9>
     | <zero>;
<zero> = (zero | oh ) {this.$value = 0;};
<1to9> = one   {this.$value = 1;}
      | two    {this.$value = 2;}
      | three {this.$value = 3;}
      | four  {this.$value = 4;}
      | five  {this.$value = 5;}
      | six   {this.$value = 6;}
      | seven {this.$value = 7;}
      | eight {this.$value = 8;}
      | nine  {this.$value = 9;};

<exp> = <digit> {this.$value = $digit;}
     | <digit>  plus {this.$value = $digit ;}
               <exp>  {this.$value = this.$value + $exp;};
     | <digit>  minus {this.$value = $digit ;}
               <exp>  {this.$value = this.$value - $exp;};
     | <digit>  multiply {this.$value = $digit ;}
               <exp>  {this.$value = this.$value * $exp;};
     | <digit>  divide {this.$value = $digit ;}
               <exp>  {this.$value = this.$value / $exp;};
```

This grammar is used by the following VoiceXML document:

```
<?xml version="1.0"?>
<vxml version="1.0">
<form id="calc">
    <field name="exp">
      <prompt>Please say an expression.  </prompt>
                <grammar src= “calc.gram” type="text/jsgf"/>
    </field>
       <filled>
                <prompt> The result is <value expr="exp" />. </prompt>
        </filled>
</form>
</vxml>
```

When the utterance “two multiply three plus four” is parsed, the value 14 is returned instead of 10. It is because of using right recursion. To have a complete computing power, it has to use scripts in VoiceXML documents.

# Chapter 4

# Client-side and Server-side Processing

VoiceXML can take advantage of existing Internet infrastructure. It can use client-side and server-side processing to expend its power. In client-side, most vendors provide ECMAScript support. In server-side, there are many choices, such as CGI, Servlets, JSP or ASP. This chapter brief introduces these techniques and gives examples of each technique. The limitations of each approach are also given.

## 4.1 ECMAScript

ECMAScript is a standard version of JavaScript backed by the European Computer Manufacturer's Association [ECMA Script]. ECMAScript is a way to do complex things on the client side of voice browser. ECMAScript is a relatively powerful and flexible object-oriented programming language. It provides dynamic capabilities. For example, we can write a routine that is executed each time the user speech some words. ECMAScript provides functions to construct arrays, fill multiple optional slots, construct objects within objects, perform trivial and complex numerical operations, manipulate dates/strings and other standard object types and more.

JSGF tagging mechanism together with ECMAScript specify a transformation from an utterance to information that is meaningful to the application. The information is returned in the form of ECMAScript values, such as strings and sets of attribute-value pairs.

The <script> tag allows a block of client-side ECMAScript code embedded in VoiceXML document. For example, Following VoiceXML document has a script that converts binary number to decimal number.

```
<?xml version="1.0"?>
<vxml version="1.0">
   <form id="convert">
    <field name="bits">
      <prompt>  Please say a binary number </prompt>
      <grammar src="http://localhost/application/bits/bits.gram" type="text/jsgf"/>
    </field>
    <filled>
       <prompt>
         The result is <value expr="convert(bits)" />
       </prompt>
    </filled>
   </form>

  <script> <![CDATA[
    function convert(str) {
     var i, len, value, tmp;
     len = str.length;
     i = 0;
     value = 0;
     while ( i < len) {
       tmp = str.charAt(i);
       value = value * 2 + parseInt(tmp);
       i++;
     }
     return value.toString();
    };
  ]]>
  </script>
 </vxml>
```

The script codes are downloaded with VoiceXML document and execute on client machine. It is very efficient to make use of the power of client machine and execute some script codes again and again. But the script can't access data stored on server and it rely on client browser support.

## 4.2 CGI

CGI (Common Gateway Interface) is a protocol, which defines the standard way in which external programs should communicate with a web server or other information servers [CGI]. CGI allows the user to invoke another program, such as a Perl script, on the web server to create the dynamic web page. CGI can be used to generate dynamic VoiceXML pages.

CGI can now be used to access every known server environment. Its function is to allow the web server to go beyond its normal boundaries for retrieving and accessing information from external databases and files. VoiceXML pages can query databases through CGI.

However, CGI has some severe shortcomings. CGI is slow. First, most CGI programs are written in Perl script, which run slower than compiled code. Second, CGI launches a new process to service each incoming client request. Finally it is not convenient to embed VoiceXML code in CGI code.

For a very simple VoiceXML document:

```
<?xml version="1.0"?>
<vxml version="1.0">
   <form id="form1">
    <field name="num" type ="number">
      <prompt>  Please say a number </prompt>
    </field>
    <filled>
       <submit next="http://speech88.cs.uwindsor.ca/cgi-bin/num.cgi" />
    </filled>
   </form>
 </vxml>
```

The CGI script to process may be as follow:

```
#!/opt/perl5/bin/perl
print " Content type: text/vxml\n\n";
print "<?xml version="1.0"?>\n";
print "<vxml version="1.0">\n";
print "<form id="form1">\n";
print "<block>\n";

($key, $value)=each(%ENV);

print "The number you said is:";
print "$value\n";
print "</block>\n";
print "</form>\n";
print "</vxml>\n";
```

## 4.3 Servlet

Servlets are the Java platform technology for extending and enhancing web servers. Servlets provide a component-based, platform-independent method for building web-based applications, without the performance limitations of CGI programs [SUN Servlet]. Servlets are server- independt and platform-independent.

Servlets are effective for effective for developing Web-based solutions that help provide secure access to a web site, that interact with databases on behalf of a client, that dynamically generate custom HTML documents to be displayed by browsers and that maintain unique session information for each client.

Servlets have access to the entire family of Java APIs, including the JDBC to access enterprise databases. Servlets can also access a library of HTTP-specific calls and receive all the benefits of the mature Java language, including portability, performance, reusability, and crash protection.

But it is still not convenient to code using servlet, which is the same as CGI. For the same VoiceXML document in 4.2, change <submit> element to

<submit next="http://speech88.cs.uwindsor.ca/servlet/numservlet " />

The code for servlet may be as follow:

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public calss HTTPGetServlet extends HTTPServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
        Throws ServletException, IOException
  {
        PrintWriter output;
        response.setContentType("text/vxml");
        output = response.getWriter();
        String num = request.getParameter("num");

        output.println("<?xml version="1.0"?>");
        output.println("<vxml version="1.0">");
```

```
            output.println("<form id="form1">");
            output.println("<block>");
            output.print("The number you said is: " + num);
            output.println("</block>");
            output.println("</form>");
            output.println("</vxml>");
    }
  }
```

## 4.4 JSP

JavaServer Pages technology (JSP) is an extension of Servlets. JSP uses XML-like tags and scriptlets written in the Java programming language to encapsulate the logic that generates the content for the page [SUN JSP]. JSP technology makes it faster and easier than ever to build web-based applications by separating the page logic from its design and display and supporting a reusable component-based design.

JSP and servlets provide an attractive way that offers platform independence, enhanced performance, separation of logic from display, ease of administration, extensibility into the enterprise and most importantly, ease of use.

Now some JSP engine provides support for VoiceXML. VoiceXML tags can be directly used in JSP documents. It is very easy to write VoiceXML code in JSP document.

For the same VoiceXML document in 4.2, change <submit> element to

<submit next="http://speech88.cs.uwindsor.ca/jsp/num.jsp " />

The following is JSP document:

```
<?xml version="1.0"?>
<vxml version="1.0"?>
<form id="form1">
<block>
  The number you said is: <%= request.getParameter("num") %>.
</block>
</form>
</vxml>
```

## 4.5 ASP

Active Server Pages (ASP) is a Microsoft technology for creating dynamic web pages [ASP]. An ASP page is an HTML page that contains server-side scripts that are processed by the Web server before being sent to the user's browser. ASP can be combined with Extensible Markup Language (XML), Component Object Model (COM), and Hypertext Markup Language (HTML) to create powerful interactive Web sites.

ASP allows persistent connections between the client and server, the development of Client server sessions, and the access and management of Databases from the client side. ASP supports VBScript and JScript. It is possible to use other scripting languages with ASP as long as you have an ActiveX scripting engine for the language installed.

Writing VoiceXML code in ASP is also very simple. The limitation of ASP is that it is only runs on Windows platform.

For the same VoiceXML document in 4.2, change <submit> element to

<submit next="http://speech88.cs.uwindsor.ca/asp/num.asp " />

The following is ASP document:

```
<?xml version="1.0"?>
<vxml version="1.0"?>
<form id="form1">
<block>
   The number you said is: <%= Request.QueryString("num") %>.
</block>
</form>
</vxml>
```

# Chapter 5

# Approaches to build complex language processors in VoiceXML

Although VoiceXML itself only performs the function of recognition, there are many approaches to expand VoiceXML expressive power: a combination of VoiceXML with Java Speech Grammar Format (JSGF) tagging, a combination of VoiceXML with ECMAScript, and a combination of VoiceXML with server-side processing. These combinations make it possible to build more complex language processors and voice web applications. These approaches give a step toward building complex voice web applications. These language processors include recognizers, translators, evaluators, data input processors, query processors and dialogue processor.

## 5.1 Recognizers

A recognizer for a language is a program that takes as input a string x and answers "yes" if x is a sentence of the language and "no" otherwise [Aho 1986].

A VoiceXML system is a recognizer which can recognizes words specified by speech grammar. It also translates speech input into strings. JSGF is a rule grammar that specifies the types of utterances a user might say.

For the following JSGF example:

```
<city> = Windsor | Toronto
```

If the user says Windsor or Toronto, VoiceXML interpreter will accept it and return a string. Otherwise VoiceXML interpreter will prompt error.

JSGF only supports right recursion. It does not support left recursion. However, there is no problem with recognition. Any speech input can be recognized against JSGF grammars.

## 5.2 Parsers

A Parser obtains a string of token from the lexical analyzer, and verifies that the string can be generated by the grammar for the language. The output of the parser is a parse tree for the stream of tokens [Aho 1986].

A parser can be constructed for any grammar. There are two commonly used parsing methods, which are expressive enough to describe most syntactic constructs in program languages. Top-down parsers build parse trees from the top to the bottom, while bottom-up parsers start from the leaves and work up to the root. The purpose of parse tree evaluation is to recursively compute a value for each non-terminal in the tree [Aho 1986].

VoiceXML with JSGF tagging is a top-down parser which does not support left recursion and top-down parsing with back-tracking. VoiceXML can generate a parse tree corresponding to an utterance against a JSGF grammar with tagging.

Consider the following grammar:

```
<zero> = (zero| oh| null) {this.$value = "0";};

<1to9> = one     this.$value = "1";}
       | two    {this.$value = "2";}
       | three  {this.$value = "3";}
       | four   {this.$value = "4";}
       | five      {this.$value = "5";}
       | six       {this.$value = "6";}
       | seven  {this.$value = "7";}
       | eight   {this.$value = "8";}
       | nine    {this.$value = "9";};

<digit> = <zero> {this.$value = "(DIG," + $zero + ")"
        | <1to9>{this.$value = "(DIG," + $1to9 + ")";

<oper> = plus     {this.$value = "(OP, +) "; }
       | minus  {this.$value = "(OP, -) "; }
       | times  {this.$value = " (OP, *) "; }
       | divide {this.$value = "(OP, / )"; }

<expr> = <digit> { this.$value = $digit;}
       | <digit> ( <oper> {this.$value = "(EXP, " + $digit + "," + $oper; }
         <expr> {this.$value = this.$value + "," + $expr + ")";});
```

The utterance " one plus six divide three", when parsed against above grammar, produces the following parse tree:

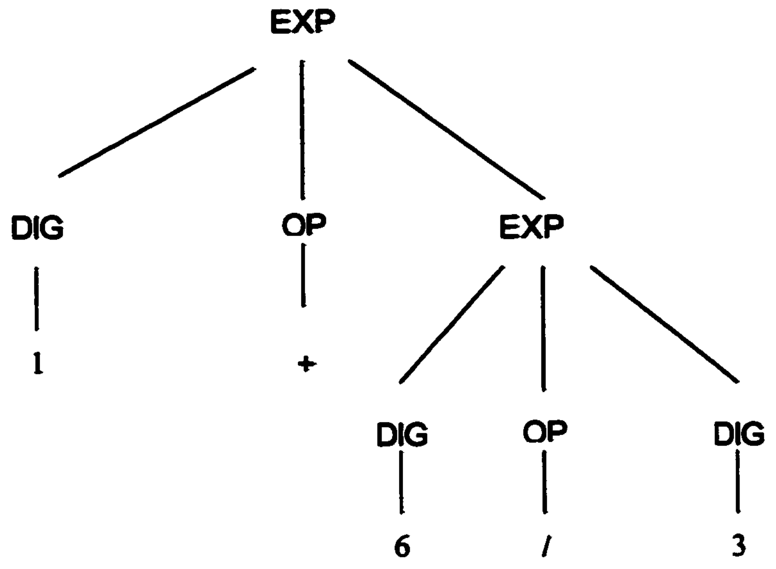

Figure 4. Parse Tree

The return value of the parser is "(EXP, (DIG, 1), (OP, +), (EXP, (DIG, 6), (OP, /), (DIG, 3))). Because JSGF grammar only supports right-most recursion, the parse tree produced is hard to translate expressions containing left-associative operators, such as plus and minus.

It is very difficult to build a parser just using VoiceXML with inline grammar or JSGF grammar without tagging. The parser built using VoiceXML with JSGF tagging is easy. A parser built using VoiceXML with ECMAScript or Server-side processing is more flexible or powerful. It can be built as top-down parser or bottom-up parser. With ECMAScript, the parser is downloaded and executes at client-side. It is more efficient if the parser is not complex and the time for downloading is short.

## 5.3 Translators

A translator is used to transform one language construct to the other. A translation is an input-output mapping. The output for each input x is specified in the following manner. First, construct a parse tree for x. Suppose a node n in the parse tree is labeled by the grammar symbol X. Write X.a to denote the value of attribute a of X at that node. The value of X.a at n is computed using the semantic rule for attribute an associated with the X-production used at node n [Aho 1986].

JSGF tags provide a mechanism to attach attribute-value pairs to parts of rule definitions for semantic interpretation. With JSGF tags, VoiceXML can translate syntactic structure of speech input.

The following grammar illustrates JSFG tags.

```
<city> = Toronto {this.$value = "TON"}
        | Windsor {this.$value = "WIN"};
<travel> = I want to go to <city>{this.$value = $city};
```

When parsed against this grammar, the utterance " I want to go to Windsor" produces the following parse tree:

```
<top>
   I
   want
   to
   go
   to
  <city>
```

```
        Windsor
        {this.$value = “WIN”}
      {this.$value =$city}
```

When evaluated this parse tree, “WIN” will be returned. VoiceXML parser translates the utterance “I want to go to Windsor” into “WIN”. It is easier for the application to use.

In VoiceXML, an expression is input as an utterance, such as “one plus three times six”. VoiceXML interpreter can't understand what it means. We can translate this into an expression using JSGF tagging.

```
One → 1
Plus → +
Three → 3
Times → *
Six → 6
```

“one plus three times six” is translated into “1 + 3 * 6”.

Following is JSGF grammar for translating utterance into an expression:

```
<zero> = (zero| oh| null) {this.$value = “0”;};
<1to9> = one     this.$value = “1”;}
         | two    {this.$value = “2”;}
         | three  {this.$value = “3”;}
         | four   {this.$value = “4”;}
         | five     {this.$value = “5”;}
         | six      {this.$value = “6”;}
         | seven  {this.$value = “7”;}
         | eight   {this.$value = “8”;}
         | nine    {this.$value = “9”;};
<digit> = <zero>
         | <1to9>;
<oper> = plus     {this.$value = “ + ”; }
         | minus   {this.$value = “ - ”; }
         | times   {this.$value = “ * ”; }
         | divide  {this.$value = “ / ”; }
```

```
<expr> = <digit> { this.$value = $digit;}
        | <digit> ( <oper> {this.$value = $digit + $oper; }
          <expr> {this.$value = this.$value + $expr;});
```

Using JSGF tagging, we can reorder our expression. For example, "one plus three times six" can be translated into + 1 * 3 6 or +(1) *(3)(6). Then it will be very easy to write the evaluator. The last rule of above grammar can be rewritten as following:

```
<expr> = <digit> { this.$value = $digit;}
        | <digit> ( <oper> {this.$value = $oper + $digit; }
          <expr> {this.$value = this.$value + $expr;});
```

If we want to add bracket, rewrite rule <digit>:

```
<digit> = <zero>  {this.$value = "(" + $zero + ")" ;}
        | <1to9> {this.$value = "(" + $1to9 + ")";} ;
```

A translator is effective and easy to be built by associating attribute values to each JSGF grammar rule. We can reorder input by concatenating strings. A language translator, such as English-to-French translator, using VoiceXML with JSGF tagging is simple and ambiguous. A very simple translator, which translates words to words, can be built using inline grammar. It is difficult to build a translator using ECMAScript. If we need powerful translators, the best choice is to using VoiceXML with server-side processing. Although it is difficult to build a translator with server-side processing, there are many existing translators which can be reused.

## 5.4 Evaluators

An evaluator for a language is a program that takes as input an expression and evaluates the expression [Aho 1986].

We can build an evaluator with the combinations of VoiceXML with JSGF tagging, ECMAScript and server-side processing.

- Use "pure" VoiceXML. It can only be used to build very simple evaluators and it is difficult to build. Because VoiceXML is only a markup language with limited control statements. It is not power enough.

- Use VoiceXML with embedded computation in JSGF tagging. It is suitable for building some simple evaluators. It is easy to be used in VoiceXML documents. However, when building complex evaluator, it will make it too complex to building JSGF grammar and the performance of the grammar will be affected.

- Use VoiceXML with JSGF tagging and ECMAScript. It is suitable for building more complex evaluators. The evaluators may make use of the client-side computing while making it easy to build JSGF grammars.

- Use VoiceXML with JSGF tagging and Server-side processing. This approach is suitable for building complex evaluators with the power of server-side computing. It can also reuse some existing powerful evaluators. ECMAScript is not suitable for very complex evaluators. Even if it can, it will take long time to download the script codes to the client-side. This will affect the performance.

We give steps to build a simple calculator in VoiceXML with JSGF tagging and ECMAScript in the following. The example of using VoiceXML with embedded computation in JSGF tagging can be found in section 3.4.

The following VoiceXML document with an evaluator built in ECMAScript references the grammar described in section 5.3.

```
<?xml version="1.0"?>
<vxml version="1.0">
<form id="calc">
   <field name="exp">
     <prompt>Please say an expression. </prompt>
               <grammar src= "calc.gram" type="text/jsgf"/>
   </field>
   <filled>
       <prompt> The result is <value expr="calculate(exp)" />. </prompt>
    </filled>
 </form>

   <script> <![CDATA[
     function calculate(expr) {
       var value;
       value = eval(expr);
       return value.toString();
     };
    ]]>
   </script>
 </vxml>
```

## 5.5 Data Input Processor

A data input language is used to validate input data according to some input field constraints [Aho 1986].

VoiceXML provides limited input data validation using JSGF grammar. It does not provide data type checking and range checking. We can build subdialogs for type checking and range checking using VoiceXML file, ECMAScript and JSGF grammar.

Figure 5 shows the relationship between VoiceXML application, subdialog, ECMAScript and JSGF grammar files.

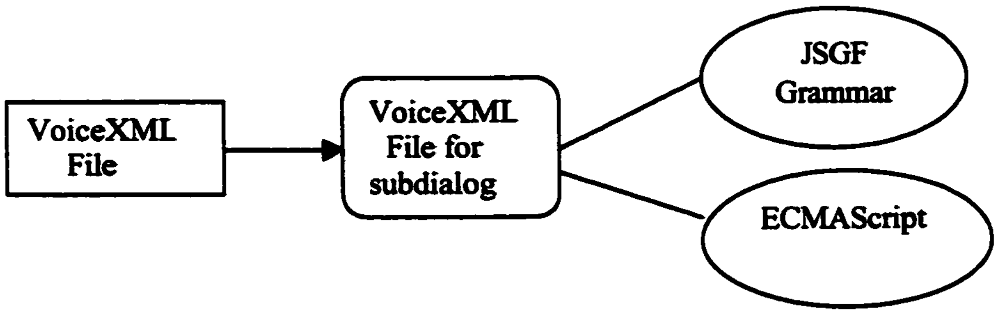

Figure 5. Subdialog Architecture

Each subdialog has a JSGF grammar to specify the list of valid utterances that a user could say at a given point in a voice application. JSGF action tags attach attribute-value pairs for a particular utterance. ECMAScript function is used to check these value according to some field input constraints.

We can define input constraints using JSGF action tags.

```
<number> = <digits>;
<type>  = credit
          | checking
         | saving;
<account> = my account number is <number>
                { this.$value = $number;
                  this.type = "credit";
                 this.min  = "1";
                 this.max = "999999999"; }
            | ( my <type> account number is <number> )
                { this.$value = $number;
                  this.type = $type";
                  this.min  = "1";
                  this.max = "999999999"; }
```

This grammar allows the user say " my saving account number is 988" and return value {value: "988", type: "saving", min: "1", max: "999999999"}. An ECMAScript function is used to check if the 988 is between 1 and 999999999. If true, then returns input data, otherwise, gives error information.

The benefit of using JFGS tagging is that we don't need to hard code ECMAScript functions for every input. The ECMAScript functions can be reused. The structure of application is clear. It is easier to maintain the voice application. When there is any requirement changed for input validation, we only need to modify JSGF grammar.

It is not effective to build an input processor using server-side processing, because there is round-trip for validating every input.

## 5.6 Query Processors

A query processor relies on the schema to devise efficient plans for computing query results [Agarwal 2000].

When a voice application needs to query information stored in database, Server-side processing should be used. There are several server-side techniques can be used, such as CGI, JSP, Servlet or ASP. Query processors can be built in client-side or server-side.

Structure of a query processor is shown in figure 6.

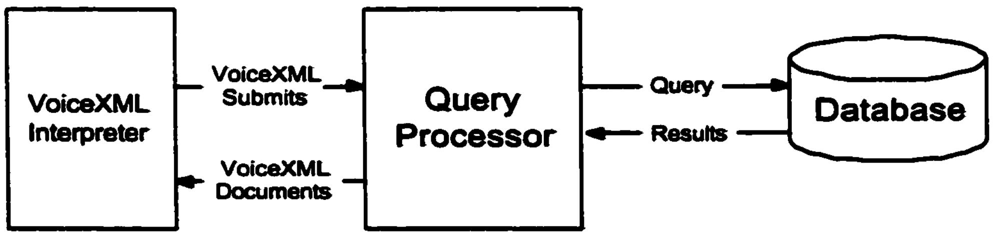

Figure 6. Structure of Query Processor

The query processor generates VoiceXML documents which are interpreted by the VoiceXML interpreter and the filled form is returned back to the query processor. Then the query processor accesses the database to obtain information. Base on results from database, the query processor generates next VoiceXML document, which should satisfy user's demand.

A client-side query processor can be built using ECMAScript. It downloads all the information from the server and access the information in the client-side. It is efficient when download information is not large and the query processor will access the information many times.

A server-side query processor executes on server-side and uses server sources. When there are many concurrent users using the query processor, the performance of the query processor will degrade. However, A server-side query processor can integrate existing software components, such as CORBA objects, EJB objects or COM/DCOM objects.

We have built a server-side query processor –course information query system in section 6.3.4. The query processor is built using VoiceXML with JSP and Java. The system dynamically generates JSGF grammar for VoiceXML documents and query course information store in a database.

## 5.7 Dialogue Processors

Dialogue process provides facilities for generating voice interactions aimed at cooperative human-machine problem solving [Allen 1995] [Boyce 1996].

VoiceXML supports simple directed dialogues. The computer directs the conversation at each step by prompting the user for the next piece of information.

A directed dialogue conversation for completing a transfer transaction in a banking system may be as follow:

C: What kind of account will you transfer from?

U: VISA card.

C: From what account number?

U: 92183214.

C: What kind of account will you transfer to?

U: Saving account.

C: To what account number?

U: 31892388.

C: How much?

U: 200 dollars.

A natural dialogue, which is referred to as a "mixed-initiative" dialogue, can be built in VoiceXML incorporating JSGF tagging mechanism and server-side processing.

The above dialogue may be more naturally as follows (Related JSGF grammar and VoiceXML codes are in section 3.2):

C: What can I help you?

U: I'd like to transfer 200 dollars from VISA card account 92183214 to my saving account 31892388.

If the user misses some information, VoiceXML browser will prompt the user. For example, if user omits account information from which to be transfer, the dialogue may be as follows:

C: What can I help you?

U: I'd like to transfer 200 dollars to my saving account 31892388.

C: What kind of account will you transfer from?

U: from VISA card 92183214.

A VoiceXML application defines a series of dialogs between a user and a computer. The application is a dialog processor. It provides directed dialogs and branching dialogs according to the user speech. The limitation of directed dialog is that the user has to follow the computer prompts step by step and gives answer at each prompts.

With JSGF tagging and server-side processing, it is flexible and scalable to build natural language dialog processors. The processors can use knowledge base to produce natural dialogs.

# Chapter 6

# Some Examples Developed to Help Identify Capability of VoiceXML with Combinations

We built a voice binary-decimal converter, a simple voice calculator and a voice course information system with combinations of VoiceXML with JSGF tagging, ECMAScript, and Server-side processing to help identify capability of VoiceXML with combinations. We use IBM WebSphere Voice Server SDK1.5 and WebSphere Studio 3.5. IBM WebSphere Voice Server SDK supports VoiceXML 1.0, JSGF grammar and ECMAScript. We can create and test voice applications on PC workstation.

## 6.1 IBM WebSphere Voice Server SDK

### 6.1.1 Installation Requirements

Hardware requirements [IBM VSDK] :

- Intel Pentium 366 MHz processor
- 128 MB RAM
- 130MB disk space for SDK
- A Microsoft Windows compatible 16-bit full duplex sound card
- A good quality microphone

Software requirements [IBM VSDK]:

- Microsoft Windows NT 4.0 with Service Pack 6 or later
- Java Runtime Environment 1.3

### 6.1.2 Components of IBM WebSphere Voice Server SDK

IBM WebSphere Voice Server SDK consists of the following components [IBM VSDK]:

- Speech recognition engine

  Speech recognition engine converts spoken input to text. It is done by IBM ViaVoice Speech Recognition engine.

- Text-to-Speech Engine

  Text-to-Speech Engine generates spoken output from text input. It is done by IBM ViaVoice Text-to-Speech engine.

- VoiceXML browser

  VoiceXML browser fetches VoiceXML documents to process. The VoiceXML browser manages the dialogue between the application and the user by playing audio prompts, accepting user inputs, and acting on those inputs. The action might involve jumping to a new dialogue, fetching a new document, or submitting user input to the Web server for processing.

## 6.2 Voice Application Design Consideration

There are four iterative phases in developing voice application: design phase, prototype phase, test phase and refinement phase [IBM VSDK].

In design phase, the first step is analyzing user requirements to identify user characteristics. The next step is making high-level application decision to define proposed functionality. Then defining information flow that maps the interaction between the application and the user. In prototype phase, a prototype of the voice application is created. In test phase, you will test the working prototype to find problems and fix them. In last phase, you will update the user interface based on the results of testing the prototype.

There are some design considerations in developing voice applications:

- Using grammars
- Constructing always-active commands
- Designing prompts
- Choosing client-side or server-side processing

### 6.2.1 Using Grammars

Since grammars constrain what the user can say, designing good grammars has a major impact on speech recognition accuracy. You should decide what you choose to put in a grammar and when you choose to make each grammar active. You also need to decide whether to use static grammar or dynamically generate grammars from a back-end database.

There are many tradeoffs in using grammars:

- Word and phrase length. Longer words and phrases have greater recognition accuracy than shorter words and phrases. When a grammar permits many short words, it is important to minimize acoustic confusability by making them as dissimilar sounding as possible. Longer words and phrases may make dialogs slower.

- Grammar complexity. Complex grammars can enhance the ability of the system to recognizer what users can say and increase dialog efficiency. But grammar files are larger and load slower. It will increase chance of recognition errors.

- Number of active grammars. More active grammars may improve usability but increase chance of recognition conflicts. Performance may degrade.

### 6.2.2 Constructing Always-active commands

Always-active commands give users a easy way to speech out the same command in every dialog. These commands, like backup, exit, help, quiet, repeat and start over, are always active. However, there is a tradeoff to having a large set of always-active commands: the potential for misrecognition increases.

### 6.2.3 Designing Prompts

One of challenge of designing a good voice application is providing just enough information at just the right time. So there are many considerations in designing prompts:

- Limiting menu length.
- Separating text for each menu item
- Ordering menu items
- Minimizing prompt length
- Choosing the right words

### 6.2.4 Choosing Client-side or Server-side Processing

Using ECMAScripts embedded in VoiceXML documents make voice applications execute some processes in client-side. It is fast to execute some data input validation using ECMAScript functions. It is also efficient to write some functions which need to be executed many times in ECMAScript, download and execute it. But if a complex function which written in ECMAScript is very large, It will take a long time to download and there is an impact to application performance.

Though Server-side processing sometimes has a lower performance than client-side processing, it is flexible and powerful. It can use existing server-side technologies and integrate legacy voice applications.

## 6.3 Voice Applications

Our Voice application consists of a voice binary-decimal converter, a simple calculator and a course information query system. We build the application using a combination of VoiceXML with JSGF tagging, ECMAScript and Server-side processing. In server-side, we choose Servlet directly access database. We can use distributed components, such as EJB or CORBA to access database.

### 6.3.1 Application Architecture

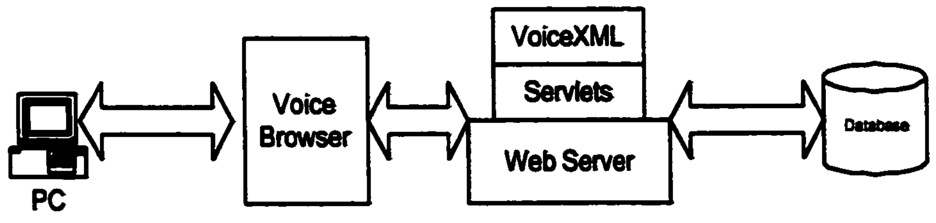

Figure 7. Application Architecture

The voice application consists of three parts: a binary-decimal converter, a calculator, and a course information query system.

When the user enters the system, the system prompts welcome information. Then prompts main menu, binary converter and simple calculator to let the user say one of these words. If the user says one of these words which is recognized by voice browser, the system will go to corresponding subsystem to continue next dialogue. Otherwise, the system will give help information. At any time, the user can say "main menu" go back to main menu and say "goodbye" to exit the system.

### 6.3.2 Voice Binary-decimal Converter

This converter is used to convert binary string said by the user to a decimal number, and speak it out. The application uses JSGF action tags to perform following translation:

One → 1

Zero →0

Then an ECMAScript function, which convert binary number to decimal number, takes translated binary string and output corresponding decimal number.

Following is JSGF grammar:

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

<bit> = one {this.$value = "1"; }
    |<zero>;
```

```
<zero> = (zero | oh ) {this.$value = "0";};

<bitval> = <bit> {this.$value = $bit; }
       | <bit> {this.$value = $bit; }
         (<bitval> {this.$value = this.$value + $bitval;});

public <retval> = <bitval>;
```

The following is VoiceXML code and ECMAScript function that performs converting:

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

  <link next = "http://localhost/application/main.vxml#main">
    <grammar>Home | Main Menu</grammar>
  </link>
  <link next = "http://localhost/application/main.vxml#end">
     <grammar>Goodbye | Bye bye | Quit | Exit</grammar>
  </link>

  <form id="convert">
    <field name="bits">
      <prompt>Welcome to use voice converter to convert binary into decimal
              number.
             Please say a binary number.
      </prompt>
      <grammar src="http://localhost/application/bits/bits.gram" type="text/jsgf"/>

      <catch event="nomatch noinput help" count="1">
            Please say a binary number. or say goodbye to exit.
      </catch>
      <catch event="nomatch noinput help" count="2">
            Please say a binary number with one or zero.
           Or say goodbye to exit.
      </catch>
    </field>

    <filled>
       <prompt>
        The binary number you said is <value expr="bits$.utterance" />
      </prompt>
      <prompt>
        The Decimal number  is <value expr="convert(bits)" />
      </prompt>
    </filled>

    <block>
      <clear namelist="bits" />
      <goto  next="#convert" />
    </block>
  </form>

  <script> <![CDATA[
    function convert(str) {
      var i, len, value, tmp;
      len = str.length;
      i = 0;
      value = 0;
      while ( i < len) {
        tmp = str.charAt(i);
        value = value * 2 + parseInt(tmp);
        i++;
      }
      return value.toString();
    };
  ]]>
  </script>

</vxml>
```

The possible dialog will be:

C: Welcome to use voice converter to convert binary number into decimal number.

Please say a binary number.

U: One zero one one.

C: The Binary number you said is 101.

The Decimal number is 5.

## 6.3.3 Voice Calculator

This is a very simple calculator with addition, subtraction, multiplication and division to one digit number. The calculator can recognize the spoken expression, calculate it and speak out the result. The application uses JSGF tags to translate utterance into an expression. Then uses an ECMAScript function to evaluate the expression.

The JSGF grammar can convert utterance like "three plus six multiply five" into expression "3 + 6 * 5".

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

<digit> = <1to9>
      | <zero>;

<zero> = (zero | oh ) {this.$value = "0";};

<1to9> = one   {this.$value = "1";}
      | two    {this.$value = "2";}
      | three {this.$value = "3";}
      | four  {this.$value = "4";}
      | five  {this.$value = "5";}
      | six   {this.$value = "6";}
      | seven {this.$value = "7";}
      | eight {this.$value = "8";}
      | nine  {this.$value = "9";};


<op> = plus     {this.$value = " + ";}
    | minus    {this.$value = " - ";}
    | (times | multiply) {this.$value = " * ";}
    | divide   {this.$value = " / ";};

<exp> = <digit> {this.$value = $digit;}
    | <digit> ( <op> {this.$value = $digit + $op;}
      <exp> {this.$value = this.$value + $exp;});
```

We can use the function provided by ECMAScript in VoiceXML to make evaluator very simple:

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

  <link next = "http://localhost/application/main.vxml#main">
    <grammar>Home | Main Menu</grammar>
  </link>
  <link next = "http://localhost/application/main.vxml#end">
    <grammar>Goodbye | Bye bye | Quit | Exit</grammar>
  </link>

  <form id="calc">
    <field name="exp">
      <prompt>Welcome to use voice calculator.
            Please say an expression.
      </prompt>
      <grammar src= “http://localhost/application/calculator/calc.gram”
                    type="text/jsgf"/>

      <catch event="nomatch noinput help" count="1">
                Please say an integer expression, or say goodbye to exit.
      </catch>

      <catch event="nomatch noinput help" count="2">
                Please say an expression with addition, subtraction, multiplication and
               division.
               Or say goodbye to exit.
      </catch>
    </field>
    <filled>

    <prompt>
        The expression you said is <value expr="exp$.utterance" />
    </prompt>
    <prompt>
        The result is <value expr="calculate(exp)" />
    </prompt>
    </filled>

    <block>
      <clear namelist="exp" />
      <goto  next="#calc" />
    </block>
    </form>

    <script> <![CDATA[

      function calculate(expr) {
        var value;
        value = eval(expr);
        return value.toString();
      };
    ]]>
    </script>

</vxml>
```

The possible conversation will be:

C: Welcome to use voice calculator.

Please say an expression.

U: three plus six multiply five.

C: The expression you said is 3 + 6 * 5.

The result is 33.

### 6.3.4 Voice Course Information Query System

This system can query course information stored in a database according to spoken input, and speak out the information.

The application uses Java Servlet to process user requests, access database, dynamically generate JSGF grammars, and give responds to the user. The server-side technology makes the application flexible and powerful.

The possible conversation will be:

C: Welcome to voice course information query system.

Please say one of these:

Name

Number

Lecturer

All

U: Name

C: Please say a course name

U: Concepts on Computer Science

C: The course Concepts on Computer Science

Course number 0360100

Section 1

Course type lecture

Course day Monday and Friday

Start time 10:30

End time 11:50

Course room ER 2123

Lecturer Dr. SABA

The following is the main VoiceXML document:

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

<vxml version="1.0">
<link next = "http://localhost/application/main.vxml#main">
   <grammar>Home | Main Menu</grammar>
</link>

<link next = "http://localhost/application/main.vxml#end">
      <grammar>Goodbye | Bye bye | Quit | Exit</grammar>
</link>

<form>
 <block>
  Welcome to the Voice Course Information Query System. You can inquire
    course information by course name,  by course number, by lecturer, or by
    all.
  <goto next="#course" />
  </block>
</form>

 <menu id="course">
  <prompt>To inquire course information, Please say one of these words:
        <enumerate/></prompt>
  <choice
        next="http://localhost:8080/servlet/CourseNameServlet">Name</choice
        >
    <choice
        next="http://localhost:8080/servlet/CourseNumberServlet">Number</ch
        oice>
   <choice
        next="http://localhost:8080/servlet/CourseLecturerServlet">Lecturer</c
        hoice>
   <choice next="http://localhost:8080/servlet/CourseAllServlet">All</choice>


   <catch event="nomatch noinput help" count="1">
     Please say name, Number, Lecturer or All.
   </catch>
  </menu>

 </vxml>
```

The following Servlet codes dynamically generates JSGF grammar and VoiceXML document which allows the user input course name by speech it. Then submit it to another Servlet to get course information by course name.

```
public class CourseNameServlet extends HttpServlet {
  PrintWriter out;
  Connection con = JdbcManage.getConnection();
  Statement statement;
  ResultSet resultSet;
  GrammarBuilder gb = new GrammarBuilder();

   public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    buildgrammar();
    try {
     con.close();
    }
    catch(SQLException ex) {
     ex.printStackTrace();
    }
    out = response.getWriter ();
    out.println("<?xml version=\"1.0\"?> \r\n" +
            "<vxml version=\"1.0\"> \r\n");
     out.println("<link next =
                  \"http://localhost/application/course/course.vxml#course\"> \r\n" +
            " <grammar>Return | Back</grammar> \r\n" +
           "</link> \r\n");
    out.println("<link next = \"http://localhost/application/main.vxml#end\"> \r\n" +
            " <grammar>Goodbye | Bye bye | Quit | Exit</grammar> \r\n" +
            "</link> \r\n");
    out.println(" <form id=\"course\"> \r\n" +
            " <field name=\"courseName\"> \r\n" +
            " <prompt>You are quering course information by course name. " +
             " Please say a course name. \r\n" +
             " </prompt> \r\n" +
            "<grammar   src=\"http://localhost/application/course/coursename.gram\"
           " +   " type=\"text/jsgf\"/> \r\n" );
   out.println("<catch event=\"nomatch noinput help\" count=\"1\"> \r\n " +
             "Please say a course name. \r\n" +
             "</catch> \r\n" +
             "</field> \r\n");
  out.println("<filled> \r\n " +
             "<prompt>The course name you said is \r\n" +
             "<value expr=\"courseName\" />. \r\n" +
                "We are going to query course information. \r\n" +
                "</prompt> \r\n" +
                "<submit next=\"http://localhost:8080/servlet/CourseServlet\"  " +
                "method = \"post\" /> \r\n" +
                "</filled> \r\n" );
    out.println("</form> \r\n ");
    out.println("</vxml> \r\n" );
  }

     private void buildgrammar(){
      //get course name from table course
       getTable();

      //build grammar file
       gb.createGramFile("d:\\ibm http server\\htdocs\\application\\course",
                  "coursename", "title");

       try {
         while (resultSet.next()) {
           gb.addWord(resultSet.getString(1));
           }
       }
       catch(SQLException ex) {
         ex.printStackTrace();
       }

       gb.closeGramFile();

      }

      private void getTable() {
        try{
          String query = "SELECT DISTINCT title FROM course";
          statement = con.createStatement();
          resultSet = statement.executeQuery(query);
        }
        catch (SQLException ex) {
          ex.printStackTrace();
        }
      }
  }
```

The following Servlet queries database by course name, the generates VoiceXML to speak out information to the user.

```
public class CourseServlet extends HttpServlet {
 PrintWriter out;
 Connection con = JdbcManage.getConnection();
 Statement statement;
 ResultSet resultSet;

  public void doPost (HttpServletRequest request, HttpServletResponse response)
   throws ServletException, IOException {
   out = response.getWriter ();
   //get POST parameter
   String tmp = request.getParameter("courseName");
   //get course information
   getTable(tmp);

  //generate VoiceXML output
   out.println("<?xml version=\"1.0\"?> \r\n" +
          "<vxml version=\"1.0\"> \r\n");
   out.println("<link next =
            \"http://localhost/application/course/course.vxml#course\"> \r\n" +
           " <grammar>Return | Back</grammar> \r\n" +
           "</link> \r\n");

   out.println("<link next = \"http://localhost/application/main.vxml#end\"> \r\n" +
           " <grammar>Goodbye | Bye bye | Quit | Exit</grammar> \r\n" +
           "</link> \r\n");

   try {
    boolean moreRecords = resultSet.next();
    if (! moreRecords ) {
      out.println(" <form id=\"course\" scope=\"document\"> \r\n" +
           " <block>There is no information for course" +
            tmp + ".\r\n" +
           " Please try another course name. \r\n" +
           " </block> \r\n"  );
     out.println("<block> \r\n" +
          " <goto next = \"http://localhost:8080/servlet/CourseNameServlet\" > /
          \r\n" + "</block> \r\n");
     out.println("</form> \r\n ");
     out.println("</vxml> \r\n" );
     con.close();
    }
    else {
     out.println(" <form id=\"course\"> \r\n" +
            " <block>Following is information about course " +
            tmp + ".\r\n" +
        " </block> \r\n"  );

   while ( moreRecords ) {
    out.println(" <block>Course name " +
            resultSet.getString(1) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Course number " +
            resultSet.getString(2) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Section " +
            resultSet.getString(3) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Course Type " +
            resultSet.getString(4) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Lecture day " +
            resultSet.getString(5) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Start time " +
            resultSet.getString(6) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Stop time " +
            resultSet.getString(7) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Location " + "Building "+
            resultSet.getString(8) + ". Room " +
            resultSet.getString(9) + ".\r\n" +
            " </block> \r\n"  );
    out.println(" <block>Lecturer " +
            resultSet.getString(10) + ".\r\n" +
            " </block> \r\n"  );
    moreRecords = resultSet.next();
   }

   con.close();
   out.println(" <block>That's end of information for course " +
          tmp + ".\r\n" +
          " </block> \r\n"  );
   out.println("<block> \r\n" +
         " <goto next =
         \"http://localhost/application/course/course.vxml#course\" /> \r\n" +
         "</block> \r\n");
   out.println("</form> \r\n ");
   out.println("</vxml> \r\n" );
  }
 }
```

```
  catch (SQLException ex) {
    ex.printStackTrace();
  }
 }

 private void getTable(String str) {
  try{
     String query = "SELECT * FROM course " + "where title like '" +
                 str + "'";
    statement = con.createStatement();
    resultSet = statement.executeQuery(query);
  }
  catch (SQLException ex) {
    ex.printStackTrace();
  }
 }
}
```

# Chapter 7

# Analysis of VoiceXML Combinations

Some example language processors are successfully built using VoiceXML with combinations of JSGF tagging, ECMAScript and Server-side processing. This chapter continues analyzing the combinations of VoiceXML with other technologies to prove that complex language processors can be constructed in VoiceXML.

## 7.1 Analysis of VoiceXML with JSGF Grammar

"Pure" VoiceXML can translate voice input into strings and recognize these strings using grammars. "Pure" VoiceXML can also build simple directed dialogue conversation. It is impossible to build a parser and a query processor in "pure" VoiceXML. It is difficult to build a simple evaluator.

## 7.2 Analysis of VoiceXML with JSGF Tagging

We can build a simple token translator in VoiceXML with JSGF tagging. JSGF tagging specifies a transformation from an utterance to information that is meaningful to the application, such as "two plus three minus four" can be translated to "2 + 3 – 4" that is easy to understand by an evaluator.

With embedded computation in JSGF tagging, a simple evaluator can be built in rightmost way, because JSGF only supports right recursion. Natural dialogs can be built using JSGF tagging. A parser built using JSGF tagging works on top-down and rightmost way. Query processors and data input processors are impossible to built.

## 7.3 Analysis of VoiceXML with ECMAScript

More language processors, such as parser, translator, evaluator and data input processor, can be built in client-side with the combination of VoiceXML with ECMAScript. It is difficult to build query processors and natural dialog processors in this combination.

## 7.4 Analysis of VoiceXML with Server-side Processing

With the combination of VoiceXML with server-side processing, any language processor can be built in server-side. The processors built on server-side can be more complicated and powerful. These processors can integrate existing components.

## 7.5 Analysis of VoiceXML with JSGF Tagging, ECMAScript and Server-side Processing

This combination makes it flexible and effective to construct language processors. There are some tradeoffs between client-side and server-side processors. Some processors such as parser, translator, evaluator and data input processor may perform efficiently in client-side. Query processors works well on server-side. Natural dialogs are easier design using all the technologies. Server-side processing can give more powerful computing and leverage legacy systems.

# Chapter 8

# Overview of Related Work

Some of the related work and recent projects in the voice application are discussed. However, their purpose of design is quite different from the approaches described this thesis. Also no work has been done on the building complex language processors in VoiceXML.

[Dou 2001] builds a generalized mapping model for the transformation from XML to VoiceXML. This model is used to simplify the development of speech applications, especially for voice access and input of information in the XML documents. The architecture and algorithm cover comprehensive considerations on speech user interface design and mapping strategies in the translation process.

[Agarwal 2000] classifies the variety of web application that can be built using voice browsers into three categories: web brosing, limited information access and spoken dialog system.

[Klautau 2000] proposes an architecture for deploying speech recognition over the Internet. The client performs the recognition, which is assisted by the server who computes the speech parameters. A Java-based Web-navigation prototype system shows good recognition can be achieved in acceptable download and calculation time even on clients with modest connection speeds and computational powers.

[Frost 1999] develops a SpeechNet, which integrates distributed speech-accessible hyperlink objects and executable specifications of attribute grammars. It is independent of HTML. The SpeechNet need to be built from the very beginning for a specific domain.

[Tu 1999] presents a speech recognition system based on an Internet client/server model. A Java applet records the voice at the client computer, sends the recorded speech file over the Internet, and the server computer recognizes the speech and displays the recognized text back to the user. Using this structure, an isolated digit recognition application was realized.

[Frost 1998] develops a speech web, which accepts user-independent continuous speech commands and responds back to the user using a text-to speech synthesizer. The paper indicates that one way to improve the speech recognition accuracy is to reengineer the spoken input language so that it is better suited to recognition process.

# Chapter 9

# Conclusions and Future Work

This thesis has investigated the expressive power of VoiceXML and the combinations with other technologies. These involve the combinations of VoiceXML with JSGF, VoiceXML with JSGF tagging, VoiceXML with ECMAScript, VoiceXML with server-side computing, and VoiceXML with JSGF tagging, ECMAScript and server-side computing. We also built a prototype system, which includes a voice binary-decimal converter, a voice translator and a course information query system. What we have done proves the thesis statement "Complex language processor can be built in VoiceXML".

"Pure" VoiceXML works as a speech recognizer. With JSGF tagging, a translator, which translates utterance into other tokens, can be built. It is more efficient to build evaluators or data input processors using JSGF tagging and ECMAScript in VoiceXML. To build a query processor or a dialog processor, it is more flexible and powerful in using server-side processing which integrates other technologies. All processors can be built as reusable subdialog components, which can be reused in different voice applications. Our investigation will provide techniques to help people build complex voice applications in VoiceXML.

However, there are some fields that are not suitable for voice applications, such as search engines, which work very well in graph web browser. VoiceXML currently is for developing voice applications for telephone, not for voice web browsing in PC. It is still difficult to build dialog language in VoiceXML. This needs to be studied in the future.

There is one critical problem in VoiceXML applications – speech recognition accuracy. The voice applications must recognize every user's speaking because there is no learning stage for every user. Current recognize engines work fine for a native language speaker. But for a person with accent, it will be frustrated. He has to use DTMF keypad input again.

Future work also include a detailed investigation of the combination of VoiceXML with JSGF tagging and ECMAScript. This combination may give the same expressive power as executable specification in SpeechNet [Frost 1999] [Frost 1995] [Frost 1994].

# References

[Agarwal 2000] Agarwal, R., Muthusamy, Y., and Viswanathan. 2000. *Voice Browsing the Web for Information Access.* 9th International WWW Conference.

[Aho 1986] Aho, A. V., Sethi, R., Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

[Allen 1995] Allen, J. F., Ferguson, B. W., Miller B. W., and Ringger E. 1995. Spoken Dialogue and Interactive Planning. *Proceeding of the ARPA Spoken Language Technology Workshop.* Austin, Texas, pp.7-48.

[ASP] Active Server Pages. http://msdn.microsoft.com/library/

[Boyce 1996] Boyce, S., and Gorin, A. L. 1996. User Interface Issues for Natural Spoken dialogue Systems. *Proceeding of International Symposium on spoken dialogue.* ISSD, pp65-68.

[CGI] Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/

[Dou 2001] Dou, H. 2001. Translation of XML Applications to VoiceXML Applications. Department of Computer Science, University of Windsor, Ontario, Canada.

[ECMA Script] ECMAScript. http://www.ecma.ch/stand/ECMA-262.htm

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

[Frost 1998] Frost, R. A. and Haddad, T. 1998. Engineering and Reengineering a Speech Interface to the Web. Proceedings of the 9th International Conference on Computing and Information, ICCI'98. University of Manitoba, pp. 237-244, June 1998.

[Frost 1995] Frost, R. A. 1995. Use of Executable Specifications in the Construction of Speech Interface. *IJCAI Workshop on Developing AI Applications for the Disabled.*

[Frost 1994] FROST, R. A. 1994. W/AGE the Windsor Attribute Grammar Programming Environment. *Schloss Dagstuhl International Workshop on Functional Programming in the Real World.*

[Hunt 2000] Hunt, A. and Walker, W. 2000. *A Fine Grained Component Architecture for Speech Application Development.* Sun Microsystems Laboratories.

[IBM VSDK] IBM WebSphere Voice Server Software Developers Kit (SDK) Programmer's Guide. Version 1.5. 2001. http://www-4.ibm.com/software/speech/

[IBM Studio] IBM WebSphere Studio, Version 3.5.
http://www-4.ibm.com/software/webservers/studio/

[Klautau 2000] Klautau, A., Jevtic, N. and Orlitsky, A. 2000. Server-assisted Speech Recognition over the Internet. ECE Department, UCSD. http://speech.ucsd.edu.

[Lucas 2001] Lucas, B., Walker, W. and Hunt, A. 2001. ECMAScript Action Tags for JSGF. Documentation of IBM Voice Server SDK.

[Lucas 2000] Lucas, B. 2000. VoiceXML for Web-based Distributed Conversational Applications. Communications of ACM. Vol 43, No. 9, pp. 53-57, September 2000.

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

[SUN JSGF] Java Speech Grammar Format Specification, Version 1.0.
http://java.sun.com/products/java-media/speech

[SUN JSML] Java Speech Markup Language Specification.
http://java.sun.com/products/java-media/speech/fordevelopers/JSML/index.html

[SUN JSP] Java Server Pages. http://java.sun.com/products/jsp/index.html

[SUN Servlet] Java Servlets. http://java.sun.com/products/servlet/index.html

[Tu 1999] Tu, Z. and Loizou, P. 1999. Speech recognition over the Internet using Java. *IEEE International Conference on Acoustics, Speech, and Signal Processing.* Phoenix, AZ:2367--70, Mar. 1999.

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

[W3C VXML] Voice Extensible Markup Language VoiceXML 1.0 Specification. http://www.voicexml.org/

[W3C VForum] VoiceXML Forum. http://www.voicexml.org/

[W3C SRG] W3C Speech Recognition Grammar Specification. 2001. http://www.w3.org/TR/2001/WD-speech-grammar-20010103

## Program Listing

### 1. Voice Converter Document

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

<vxml version="1.0">

 <meta name="Content-Type" content="text/x-vxml"/>
 <meta name="author" content="Xuejun Liu"/>

 <property name="caching" value="fast" />

  <link next = "http://localhost/application/main.vxml#main">
    <grammar>Home | Main Menu</grammar>
  </link>

  <link next = "http://localhost/application/main.vxml#end">
     <grammar>Goodbye | Bye bye | Quit | Exit</grammar>
  </link>


  <form id="convert">
    <field name="bits">
      <prompt>Welcome to use voice converter to convert binary into decimal
      number.
             Please say a binary number.
      </prompt>
      <grammar src="http://localhost/application/bits/bits.gram"

 type="text/jsgf"/>

      <catch event="nomatch noinput help" count="1">
       Please say a binary number. or say goodbye to exit.
      </catch>

      <catch event="nomatch noinput help" count="2">
       Please say a binary number with one or zero.
       Or say goodbye to exit.
      </catch>
    </field>

    <filled>
    <prompt>
       The Bianary string you said is <value expr='bits$.utterance" />
     </prompt>

     <prompt>
       The result is <value expr="convert(bits)" />
     </prompt>

  </filled>

  <block>
    <clear namelist="bits" />
    <goto  next="#convert" />
  </block>
</form>

<script> <![CDATA[
  function convert(str) {
    var i, len, value, tmp;
    len = str.length;
    i = 0;
    value = 0;
    while ( i < len) {
      tmp = str.charAt(i);
      value = value * 2 + parseInt(tmp);
      i++;
    }
    return value.toString();
  };
 ]]>
 </script>

</vxml>
```

**2. Voice Converter JSGF Grammar File**

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

<bit> = one {this.$value = "1"; }
    | <zero>;

<zero> = (zero | oh ) {this.$value = "0";};

<bitval> = <bit> {this.$value = $bit; }
```

```
        | <bit> {this.$value = $bit; }
         (<bitval> {this.$value = this.$value + $bitval;});

    public <retval> = <bitval>;
```

## 3. Voice Calculator Document

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

    <vxml version="1.0">

      <meta name="Content-Type" content="text/x-vxml"/>
      <meta name="author" content="Xuejun Liu"/>

      <property name="caching" value="fast" />


       <link next = "http://localhost/application/main.vxml#main">
         <grammar>Home | Main Menu</grammar>
       </link>

       <link next = "http://localhost/application/main.vxml#end">
          <grammar>Goodbye | Bye bye | Quit | Exit</grammar>
       </link>


       <form id="calc">
         <field name="exp">
           <prompt>Welcome to use voice calculator.
                  Please say an expression.
           </prompt>
           <grammar src="http://localhost/application/calculator/calc.gram"
                type="text/jsgf"/>

           <catch event="nomatch noinput help" count="1">
            Please say an integer expression. or say goodbye to exit.
           </catch>

           <catch event="nomatch noinput help" count="2">
            Please say an expression with addition, subtraction, multiplication and
           division.
            Or say goodbye to exit.
           </catch>
         </field>
    <filled>

      <prompt>
        The expression you said is <value expr="exp$.utterance" />
       </prompt>

       <prompt>
        The result is <value expr="calculate(exp)" />
       </prompt>

    </filled>

    <block>
      <clear namelist="exp" />
      <goto  next="#calc" />
    </block>
  </form>

  <script> <![CDATA[

    function calculate(expr) {
      var value;
      value = eval(expr);
      return value.toString();
    };
  ]]>
  </script>

</vxml>
```

**4. Voice Calculator JSGF Grammar File**

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

<digit> = <1to9>
      | <zero>;
<zero> = (zero | oh ) {this.$value = "0";};
<1to9> =  one   {this.$value = "1";}
      | two    {this.$value = "2";}
      | three {this.$value = "3";}
      | four  {this.$value = "4";}
      | five  {this.$value = "5";}
      | six    {this.$value = "6";}
      | seven {this.$value = "7";}
```

```
        | eight {this.$value = "8";}
        | nine  {this.$value = "9";};

    <op> = plus     {this.$value = " + ";}
        | minus     {this.$value = " - ";}
        | multiply {this.$value = " * ";}
        | divide    {this.$value = " / ";};

    <exp> = <digit> {this.$value = $digit;}
         | <digit> ( <op> {this.$value = $digit + $op;}
           <exp> {this.$value = this.$value + $exp;});

    public <cal> = <exp>;
```

## 5. Voice Course Information Query System Document

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

<vxml version="1.0">
  <meta name="Content-Type" content="text/x-vxml"/>
  <meta name="author" content="Xuejun Liu"/>

  <property name="caching" value="fast" />

   <link next = "http://localhost/application/main.vxml#main">
     <grammar>Home | Main Menu</grammar>
   </link>

   <link next = "http://localhost/application/main.vxml#end">
      <grammar>Goodbye | Bye bye | Quit | Exit</grammar>
   </link>

   <form>
     <block>
      Welcome to the Voice Course Information Query System.
      You can inquire course information by course name,
      by course number, by lecturer, or by all.
      <goto next="#course" />
     </block>
   </form>

   <menu id="course" scope="document">
     <prompt>To inquire course information, Please say one of these words:
     <enumerate/></prompt>
  <choice
     next="http://localhost:8080http://localhost:8080http://localhost:8080http://lo
     calhost:8080http://localhost:8080http://localhost:8080/servlet/CourseNameS
     ervlet">Name</choice>
  <choice
     next="http://localhost:8080http://localhost:8080http://localhost:8080http://lo
     calhost:8080http://localhost:8080http://localhost:8080/servlet/CourseNumbe
     rServlet">Number</choice>
  <choice
     next="http://localhost:8080http://localhost:8080http://localhost:8080http://lo
     calhost:8080http://localhost:8080http://localhost:8080/servlet/CourseLectur
     erServlet">Lecturer</choice>
  <choice
     next="http://localhost:8080http://localhost:8080http://localhost:8080http://lo
     calhost:8080http://localhost:8080http://localhost:8080/servlet/CourseServlet
     ">All</choice>

  <catch event="nomatch noinput help" count="1">
   Please say name, Number, Lecturer or All.
  </catch>

  <catch event="nomatch noinput help" count="2">
   You can query course information by course name, by course number,
   by lecturer, or by all. Please say one of these words: Name, Number,
   Lecturer, or all.
  </catch>
 </menu>

</vxml>
```

# VITA AUCTORIS

| | |
|---|---|
| **NAME:** | Xuejun Liu |
| **PLACE OF BIRTH:** | Shaoguan, China |
| **YEAR OF BIRTH:** | 1967 |
| **EDUCATION:** | South China University of Technolofy Guanzhou, China<br>1985-1989 B. Sc. |
| | University of Windsor, Windsor, Ontario<br>1999-2001 M. Sc. |