XHTML+Voice Programmer's Guide

Version 1.0

Printed in the USA

Note:
Before using this information and the product it supports, read the general information in “Notices” on page 133.

First Edition (February 2004)
This edition applies to release 1, modification 0 of the Multimodal Programmer’s Guide and to all subsequent releases and modifications until otherwise indicated in new editions. IBM may publish one or more new editions of this publication in a downloadable format after the program is generally available. To obtain the most recent edition of this publication, go to the Web site at
http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi

Contents

About this Book **1**

Who should read this book? 1

Related programs and publications 1

Multimodal user-interface design 2

Specifications and standards 2

How this book is organized 2

Document conventions 3

Chapter 1 **Overview of XHTML+Voice** **5**

XHTML+Voice as a markup language 5

What can a multimodal interaction offer? 6

How XHTML+Voice works 7

Starting with a visual interface 7

Adding voice markup 8

Combining voice and visual markup 9

Correlating voice and visual input/output 9

The architecture of X+V 10

Advantages of separating visual and voice 11

Coding a multimodal interaction 12

Conclusion 16

Individual elements of XHTML+Voice 17

What is VoiceXML? 17

What is XHTML? 17

What is an event handler? 18

What is a conformance document? 18

Chapter 2 **Elements and attributes of the XHTML+Voice Language** **21**

VoiceXML elements supported in X+V 21

Form and Form Items 22

<form> 22
<initial> 24
<field> 25
<block> 28
<record> 29
Catching/Throwing Events 30
<catch> 30
<throw> 31
<error> 32
<help> 33
<noinput> 34
<nomatch> 35
Speech Input 36
<grammar> 36
<option> 37
<lexicon> 38
Executable Content 40
<assign> 40
<clear> 41
<else> 41
<elseif> 42
<filled> 43
<if> 44
<log> 44
<var> 45
Speech and Audio Output 46
<audio> 46
<enumerate> 49
<prompt> 50
<reprompt> 53
<value> 54
<lexicon> 56
Subdialog Support 57
<param> 57
<return> 59
<subdialog> 61
Property 65
<property> 65
XHTML+Voice tags 68
<sync> 69

<cancel> . . . 72

XML Events supported in X+V . . . 74

<listener> . . . 74

Compatibility with the XHTML+Voice Specification . . . 77

XHTML+Voice . . . 77

XHTML . . . 77

VoiceXML . . . 77

JSGF . . . 79

SISR . . . 80

Setting MIME types . . . 80

Chapter 3 **Adding Grammars** **81**

What is a grammar? . . . 81

Grammar considerations . . . 82

Using fast match grammar . . . 83

Grammar features available in the Multimodal Toolkit . . . 84

Creating JSGF grammars . . . 84

Adding an external JSGF grammar . . . 85

Adding an inline JSGF grammar . . . 86

Exceptions to the JSGF specification . . . 86

Importing a JSGF grammar into another JSGF grammar . . . 87

Adding semantic interpretation . . . 87

Exceptions to the SISR specification. . . . 88

Creating a pronunciation pool file . . . 88

Adding a pool file for an external grammar . . . 89

Adding a pool file for an inline grammar . . . 89

Pronunciation features available in the Multimodal Toolkit . . . 89

Importing Reusable Dialog Components . . . 90

Adding mixed initiative applications and form level grammars . . . 90

Chapter 4 **Example Applications** **93**

Introduction . . . 93

Three basic examples to get started . . . 94

Example 1 . . . 96

Example 2 . . . 99

Example 3 . . . 103

Yes/no JSGF grammar . . . 109

Beverage JSGF grammar 110
Example 4 112
Yes/no JSGF grammar 121

Chapter 5 Multimodal Browser 123

What is a Multimodal Browser? 123
Browser features available in the Multimodal Toolkit 123
Running the Multimodal Browser 124
Using the Opera browser 125
Setting Voice preferences 125
Using the NetFront browser 127
Setting Voice preferences 127
Troubleshooting tips 129

Chapter 6 References 131

Appendix A Notices 133

Copyright License 135
Trademarks. 135

About This Book

This book provides information about the XHTML + Voice 1.2 language to create multimodal applications written in XHTML and VoiceXML 2.0. The resulting applications can then be deployed in a browser that has been modified to accept speech input, referred to as a multimodal browser.

This chapter contains the following sections:

- "Who should read this book?" on page 1.
- "Related programs and publications" on page 1.
- "How this book is organized" on page 2.
- "Document conventions" on page 3.

Who should read this book?

The following users can benefit from this book:

- An application developer interested in creating XHTML + Voice applications.
- A content creator responsible for the creative aspects of multimodal applications.
- A multimodal user-interface designer interested in promoting and maintaining uniformity in the visual and voice interfaces.

Related programs and publications

Reference, design, and programming information for creating multimodal applications is available from a variety of sources, as represented by the documents listed in this section.

Note:

Guidelines and publications cited in this book are for your information only and do not in any manner serve as an endorsement of those materials. You alone are responsible for determining the suitability and applicability of this information to your needs.

Multimodal user-interface design

The user-interface guidelines presented in this book are an evolving set of recommendations based on industry research and lessons learned in the process of developing our own speech and multimodal applications. For more information, refer to speech industry literature and publications such as the following sources:

- *Audio System for Technical Readings (ASTeR)* by T. V. Raman, a Ph.D. Thesis published by Cornell University, May 1994.
- *Auditory User Interfaces—Towards The Speaking Computer* by T. V. Raman, published by Kluwer Academic Publishers, August 1997.
- *"Directing the Dialog: The Art of IVR"* by Myra Hambleton, published in Speech Technology, Feb/Mar 2000.
- *Handbook of Human-Computer Interaction* by Thomas K Landauer, Martin Helander, and Prasad V. Prabhu, published by Elsevier Science, Amsterdam, North-Holland, June 1997.
- *How to Build a Speech Recognition Application: A Style Guide for Telephony Dialogues* (Second Edition) by Bruce Balentine, David P. Morgan, and William S. Meisel, published by Enterprise Integration Group, San Ramon, CA, 2001.

Specifications and standards

For the specifications related to this version of XHTML+ Voice, see Chapter 6, "References" on page 131.

How this book is organized

This book is organized into the following sections:

- Chapter 1, "Overview of XHTML+Voice" on page 5 provides an overview of creating multimodal applications using the XHTML+Voice language.
- Chapter 2, "Elements and attributes of the XHTML+Voice Language" on page 21 introduces the basic concepts and constructs of XHTML+Voice.

- Chapter 3, "Adding Grammars" on page 81 contains basic information about valid grammars for XHTML+Voice.
- Chapter 4, "Example Applications" on page 93 contains sample code for example applications using XHTML+Voice.
- Chapter 5, "Multimodal Browser" on page 123 contains information about the multimodal browser.
- Chapter 6, "References" on page 131 contains useful Web links and locations of related specifications and documents.
- Appendix A: "Notices" on page 133 contains notices and trademark information.

Document conventions

This document uses the following conventions:

Italic	Used for emphasis, to indicate variable text, and for references to other documents.
Bold	Used for names of elements, attributes, and events. Also used for properties, file names, URLs, and user interface controls such as commands and menus.
`Courier Regular`	Used for sample code.

Chapter 1 Overview of XHTML+Voice

The XHTML+Voice (X+V) language lets you develop multimodal applications. This chapter introduces the underlying concepts for developing multimodal applications.

This introduction discusses the following topics:

- "XHTML+Voice as a markup language" on page 5.
- "What can a multimodal interaction offer?" on page 6.
- "How XHTML+Voice works" on page 7.
- "Individual elements of XHTML+Voice" on page 17.

This chapter is based on "X+V is a markup language, not a Roman math expression," by Les Wilson, IBM developerWorks(R) (http://www.ibm.com/developerworks/), August 19, 2003. Reprinted with permission.

XHTML+Voice as a markup language

X+V is a Web markup language for developing multimodal applications. Like VoiceXML, X+V meets the increasing user demand for voice-based interaction in small and mobile devices. Unlike VoiceXML, X+V uses both voice and visual elements, bringing a world of new potential to the field of wireless user interface development. This section provides an introduction to X+V, including a conceptual overview of multimodal interface development, an architectural view of the three components that comprise X+V's core functionality, and a code example that demonstrates the utility of this promising new markup language.

The emerging world without wires has fostered a growing number of small and mobile devices (everything from PDAs to smart phones) capable of accessing data and running applications. The trouble is, while devices are getting smaller, human hands and fingers are not. To assist users in managing their devices, user-interface designers have begun to combine the traditional keyboard-input model with such interactive technologies as voice-directed input. This type of interaction, in which the user has more than one means of accessing data in his or her device, is sometimes called multimodal interaction. It is fast becoming the norm in the world of wireless mobile computing.

X+V is a Web markup language for developing multimodal interfaces for the Web. With X+V, a Web page developer can code both the visual and voice elements of a user interaction. Because it is based on existing, tested standards, X+V is an exceptionally powerful markup language, bringing a great deal of versatility to the field of multimodal interface development.

In the rest of this chapter, you'll learn the basics of X+V, including the concepts of multimodal interface design, such as how multimodal interactions function for the user, and the essential elements of X+V. You'll learn about the three main standards that comprise X+V (XHTML, VoiceXML, and XML Events), the language's architectural model, and the coding of a simple multimodal interaction.

For more information, locate the specification in Chapter 6, “References” on page 131.

What can a multimodal interaction offer?

If asked, most developers will cite speed and efficiency as the main reasons for developing multimodal interfaces. Parallel input, such as the ability to both key in commands and voice them, allows users to more quickly access and respond to information delivered by their devices. In fact, multimodal systems don't just enable faster interactions, they also add value to the overall experience of interaction. Multimodal interfaces allow more room for user preference (giving users a choice of how they interact with the system) and reduce the overexertion that can result from single-modality interaction. Being able to switch between modes of interaction (using a combination of keyboard, touch screen, stylus, telephone keys, and voice) can lead to a lower incidence of error (because users can choose the mode most suited to different activities), as well as easier error recovery. And, finally, multimodal interfaces have the capacity to accommodate a wider range of tasks and environments than single-modality interfaces.

While speech adds value to small mobile devices, mobility and wireless connectivity are also moving computing itself into new physical environments. In the past, checking your e-mail meant sitting down at a desktop or laptop with a modem and dialing up an e-mail service. Now, you can do it from a bench in the park or walking from your desk to your car. Bringing devices into new environments and circumstances requires new ways to access them. The ability to switch between interaction modes -- eyes-free, hands-free, audio-only -- is essential to facilitating true device mobility. And, thinking it through, the need for multimodal interaction doesn't end with the device interface. Wireless networks now provide connectivity anywhere and anytime. Connecting mobile devices to the network links mobile computing to back-end data anywhere and anytime.

If the need for multimodal interaction extends to the network, then the Internet needs new technologies and standards to enable that functionality. Increasingly, Web developers are seeking ways to turn existing visually oriented Web pages into multimodal ones. And that's where X+V comes in.

How XHTML+Voice works

XHTML+Voice (X+V) is a proposed markup language for developing multimodal Web pages. X+V combines XHTML and a subset of VoiceXML. XHTML is essentially HTML 4.0 adjusted to comply with the rules of XML. It is the current standard for building Web pages. VoiceXML was one of the first XML-based languages developed in the W3C(R). It provides an easy, standardized format for building speech-based applications. Together, XHTML and VoiceXML enable Web developers to add voice input and output to traditional, graphically-based Web pages.

X+V is still in the proposal phase, but it promises to deliver the feature set, flexibility, and ease of use that developers need to write one application that supports visual-only, voice-only, and multimodal interaction. The versatility of the Web and XML is reflected in the fact that X+V nicely integrates VoiceXML into the Web by marrying it with XHTML.

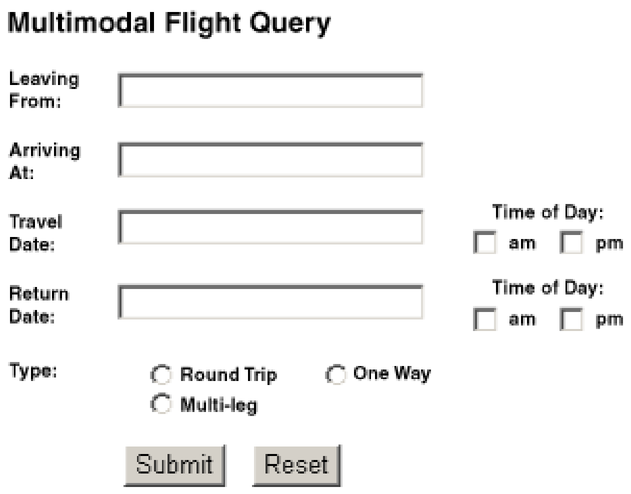
For more information, locate the XHTML+Voice specification in Chapter 6, “References” on page 131.

Starting with a visual interface

Today, most Web application developers use some type of markup language to code an application's user interface. The markup language for the user interface is called the presentation layer of the application. The presentation layer defines how the user can interact with the application. It is in the presentation layer that an application is enabled for voice.

HTML was once the ruling standard for coding the presentation layer, but in recent years it has been supplanted by XHTML. Building an XHTML user interface typically involves laying out graphics, input fields, text prompts, check boxes, and so on. More sophisticated user interfaces might also include some type of scripting, such as JavaScript, to enable input checking and other minor computation or user-interface tasks. Figure 1 shows a portion of a flight information application UI, where you can see a variety of input fields, check boxes, and so on, combined.

Figure 1. Multimodal Flight Query example



Adding voice markup

X+V incorporates a subset of VoiceXML, a fully standardized and complete markup language for creating voice applications. VoiceXML has been developed and revised over several years by industry experts in voice programming and tested in complex real-world programming scenarios such as call centers. VoiceXML is a rich language for developing a wide range of applications, and with X+V, it is not limited to just voice applications. X+V uses the most essential elements of VoiceXML, applying them to the specific task of speech-enabling application interfaces.

One advantage of basing X+V on VoiceXML is the existing, highly trained developer community, as well as the educational materials, infrastructure, tooling, and test facilities that come with a standardized language. The other advantage is the powerful framework that VoiceXML provides to developers working with X+V.

Taking the simple interface in Figure 1 as an example, you see several input fields: a few check boxes, a bank of radio buttons, and some push buttons. A basic X+V implementation of this application would speech-enable each input field so that, as you move between the fields (check boxes, and so on), you get a voice prompt as well as a visual one. This fairly simple type of speech interaction is called a “directed dialog” interaction.

A richer implementation would allow more conversational voice input from the user, such as "I'm going from Miami to Atlanta on May 21 and returning on June 1." This type of interaction, called a “mixed initiative” interaction, is enabled by VoiceXML and is available in X+V.

Combining voice and visual markup

Visual markup tells a Web browser what you want the user interface to look like and how you want it to behave when the user types, points, or clicks. Similarly, voice markup tells the Web browser what you want it to do when the user speaks to it. For visual markup, the browser uses a graphics engine; for voice markup, the browser uses a speech engine.

Just as visual markup specifies the visual interface items, voice markup specifies the voice interface items. Speech-enabling an application interface is a matter of first breaking the visual interface into its basic components (for example, an input field for a time of day and a check box for "a.m." or "p.m."), creating snippets of voice markup for each component, and then associating the snippets to the existing visual markup for each component. Consider the following examples:

- What words should the speech engine speak or synthesize?
- What words and phrases should the speech engine listen for?
- What should the browser do if the speech engine doesn't recognize a word or phrase?
- What will be the result of the speech engine recognizing a word or phrase that has been spoken?

Correlating voice and visual input/output

Given an application's visual markup plus a collection of voice markup snippets, you have almost everything you need to create the presentation layer of a multimodal Web application. In fact, the only thing you still need is a way to tell the browser which snippets of voice markup go with which visual elements, and (because a speech engine can only have one snippet active at a time) when to activate each snippet of voice markup.

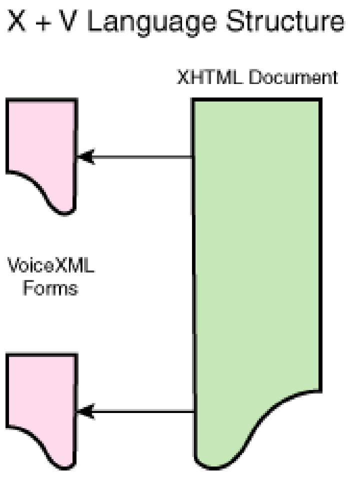
Given that the Web application environment is event-driven, X+V incorporates the Document Object Model (DOM) eventing framework used in the XML Events standard. Using this framework, X+V defines the familiar event types from HTML such as "on mouse-over" or "on input focus" to create the correlation between visual and voice markup. Using XML Events provides X+V with a uniform and standards-based eventing model that enables event integration between XML languages.

The architecture of X+V

So far, you know that a multimodal Web application written in X+V consists of visual markup, a collection of snippets of voice markup for each element in the user interface, and event markup that tells the application which snippets to activate when. For visual markup, X+V uses the familiar XHTML standard. For voice markup, it uses a subset of VoiceXML defined by the VoiceXML Form construct. For associating VoiceXML with visual interface elements, X+V uses the XML Events standard. All of these are official standards for the Web as defined by the Internet Engineering Task Force (IETF) that governs Web standards.

Thinking of this visually (or architecturally), you can imagine the XHTML document as a container of markup for visual elements (forms, fields, check boxes, text); a container of markup that speech-enables those elements (VoiceXML fields, forms); and a container for XML Event markup that correlates voice and visual elements so that they behave as you want them to. Figure 2 is a visual representation of the X+V language structure.

Figure 2. X+V’s language architecture



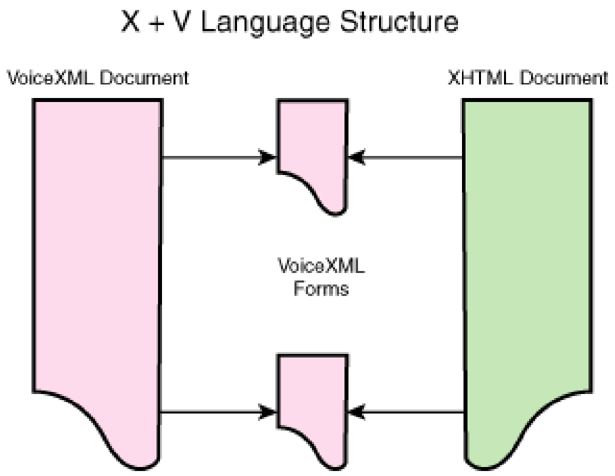
Advantages of separating visual and voice

Because all the parts of X+V are XML-compliant, the voice markup can be packaged in two ways: in the same file as the XHTML or in separate files. Separating voice markup from visual markup gives you more flexibility in developing your applications. For example, you can develop the voice markup separately from the visual markup and combine the two later.

Another advantage of keeping the files separate is reuse, such as the ability to reuse snippets of VoiceXML in numerous XHTML pages. In the example of our flight-reservation application, when a user makes a reservation he will be asked if he wants a one-way, round-trip, or multi-leg reservation. For each answer, the system will call up a different form. While the three forms differ with regard to the type of trip desired, each one has the same departure city. If you have separated the voice snippet for the departure city you can reuse it in each of the three different XHTML forms, or containers.

The final advantage of keeping the VoiceXML separate from the XHTML is that it allows the snippets of VoiceXML to be reused in containers other than XHTML. For example, we might use a VoiceXML document as a container, as shown in Figure 3.

Figure 3. X+V language structure with multiple containers



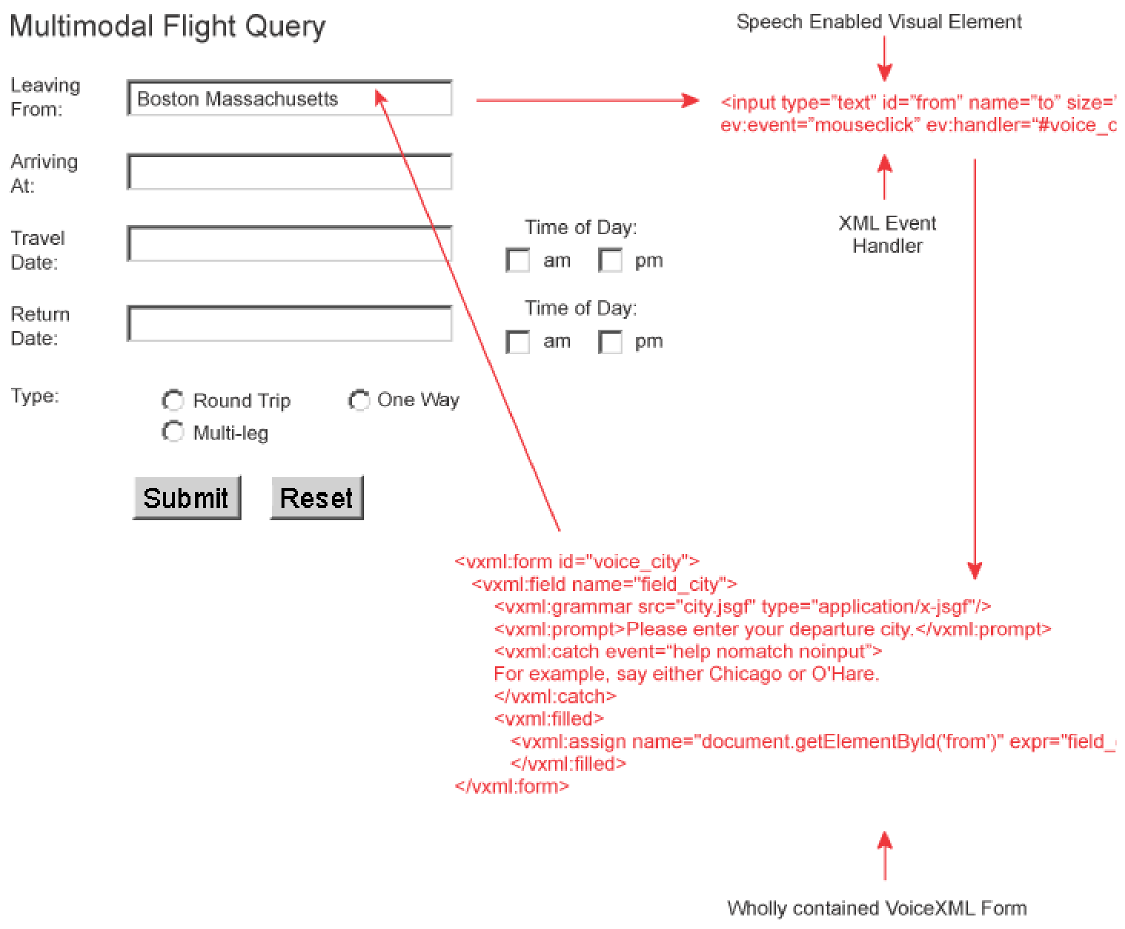
In this case, X+V is utilizing the VoiceXML notion of documents and forms, wherein a VoiceXML document contains one or more forms. You already know that VoiceXML forms can be linked to XHTML to create multimodal applications. But such forms can also be stitched together in a

VoiceXML document (or container) to create voice-only applications. The end result is that you can (by reuse) create a single application that simultaneously supports multimodal browsers, GUI-only browsers, and voice-only systems such as IVRs.

Coding a multimodal interaction

You know that X+V uses XHTML for visual interaction, a subset of VoiceXML (basically the <form> tag and everything it contains) for voice interaction, and XML Events to correlate the two. The next step is to see how the different code elements come together to create a multimodal interaction. We'll take the original example shown in Figure 1 and advance it to implement the scenario diagrammed in Figure 4.

Figure 4. Multimodal scenario



```
<input type="text" id="from" name="to" size='
ev:event="mouseclick" ev:handler="#voice_c
```

```
<vxml:form id="voice_city">
  <vxml:field name="field_city">
    <vxml:grammar src="city.jsgf" type="application/x-jsgf"/>
    <vxml:prompt>Please enter your departure city.</vxml:prompt>
    <vxml:catch event="help nomatch noinput">
    For example, say either Chicago or O'Hare.
    </vxml:catch>
    <vxml:filled>
      <vxml:assign name="document.getElementById('from')" expr="field_
      </vxml:filled>
</vxml:form>
```

In this scenario, the user is prompted both visually and by a synthesized voice. The user responds to the first directive, "Enter the departure city," with voice input: "Boston, Massachusetts." The speech engine recognizes the phrase and returns a text string. The text is displayed and the application moves the input focus to the next field, where the next interaction takes place.

The XHTML markup for the Departure City field is essentially a one-line Field tag:

```
<input type="text" id="from" name="to" size="20">
```

The VoiceXML markup for the Departure City field is a bit more complex, having the following elements:

- A voice prompt for Departure City
- A grammar that lists all the Airport Cities
- A directive telling the speech engine where to put the results
- Directives for what to do in case of failure (for example, if the user says "Help," the speech engine can't match the user's word or phrase to a grammar element, or the user says nothing).

Grammars are the way that application developers tell the recognition engine what words and phrases are allowable in the application. In this example, the application developer provides a grammar for all the phrases that might be spoken to fill out all the fields in the page. Other grammars are provided for the individual fields. The VoiceXML snippet that speech enables a field will use the grammar for that field but the grammar with the phrases for all the fields would be used to speech enable the whole page. This is where XML Events ties the voice and visual together. XML Events is how the application developer indicates what conditions the system activates the grammar for the page (e.g. when the page is loaded) or the grammar for the field (e.g. when the user clicks on a specific field).

The sample code below shows the snippet of VoiceXML for the Departure City field.

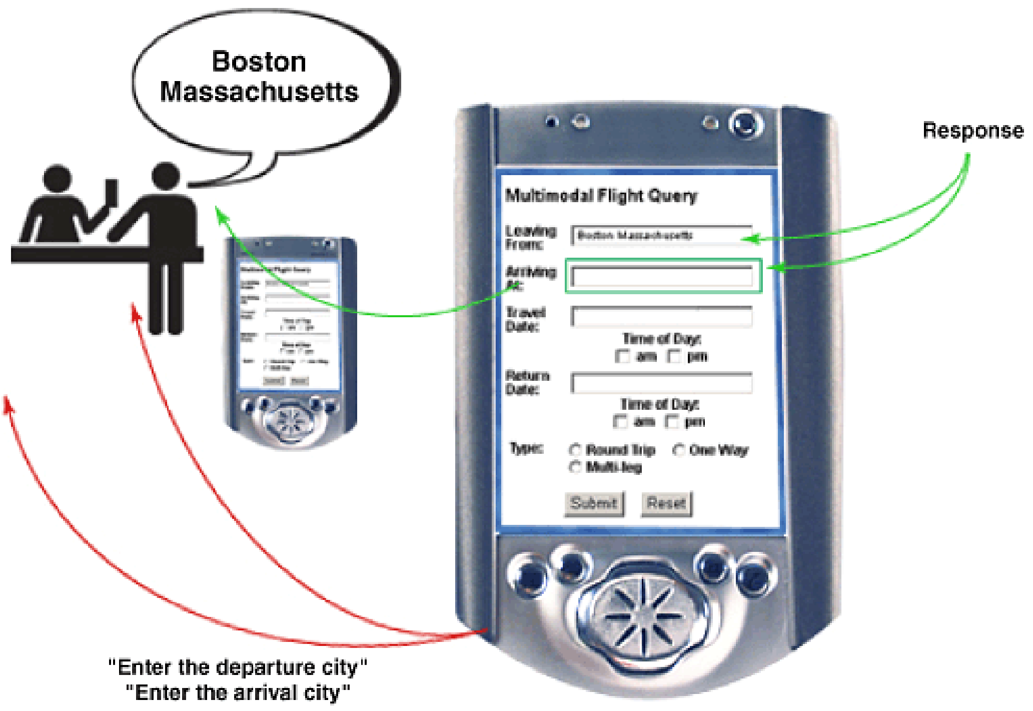
```
<vxml:form id="voice_city">
         <vxml:field name="field_city">
           <vxml:grammar src="city.grxml" type="application/srgs+xml"/>
           <vxml:prompt>Please enter your departure city.</vxml:prompt>
           <vxml:catch event="help nomatch noinput">
             For example, say either Chicago or O'Hare.
           </vxml:catch>
           <vxml:filled>
             <vxml:assign name="document.getElementById('from')"
   expr="field_city"/>
           </vxml:filled>
       </vxml:form>
```

The final step is to add the XML Events markup to the XHTML tag. The event markup does two things: It identifies the snippet of VoiceXML that speech-enables the XHTML tag and it identifies the conditions or event that will activate the VoiceXML snippet. The resulting <field> tag activates the VoiceXML form named voice_city when an input focus event occurs, as shown below.

```
<input type="text" id="from" name="to" size="20" ev:event="inputfocus"
ev:handler="#voice_city"/>
```

In Figure 5 we see how all of this comes together. The visual markup for the departure city field is denoted in green, the voice markup is in red, and the event that ties them together is in purple.

Figure 5. Implementing a multimodal scenario in X+V



Conclusion

X+V is the latest addition to the XML family of technologies for user interface development. Whereas XHTML is for developing visual interfaces, and VoiceXML focuses entirely on voice-based development, X+V is a hybrid, dedicated to developing multimodal application interfaces. X+V is particularly well suited to wireless development, where developers are faced with small visual interfaces and increasing user demand for voice input and output.

As you can see from this section, X+V's foundation in existing XML standards lends it tremendous strength and versatility. Interfaces developed using X+V are portable to a wide range of applications and development environments, can be easily developed in teams, and are highly scalable over time.

Developers working with X+V can access the numerous resources that come with a well-developed standard such as XML. X+V also takes developers out of the loop of learning a new development language such as SALT, or adapting to the constraints of a more visually oriented development environment. Perhaps best of all, X+V does not require a degree in linguistics to operate; a basic knowledge of XML and related standards is sufficient to get started.

Individual elements of XHTML+Voice

The following topics provide background information of the individual elements of the X+V markup language.

What is VoiceXML?

The Voice eXtensible Markup Language (VoiceXML) is an XML-based markup language for creating distributed voice applications, just as HTML is a language for distributed visual applications. VoiceXML was defined and promoted by an industry forum, the VoiceXML Forum$^{(TM)}$, founded by AT&T$^{(R)}$, Lucent$^{(R)}$, Motorola$^{(R)}$, and IBM, and supported by approximately 500 member companies. Updates to VoiceXML are a product of the W3C voice working group. The language is designed to create audio dialogs that feature text-to-speech, pre-recorded audio, recognition of both spoken and DTMF key input, recording of spoken input, telephony, and mixed-initiative conversations. Its goal is to provide voice access and interactive voice response (such as by telephone, PDA, or desktop) to Web-based content and applications.

Users interact with these Web-based voice applications by speaking or by pressing telephone keys rather than through a graphical user interface.

For more information, locate the **VoiceXML specification** in Chapter 6, “References” on page 131.

What is XHTML?

The eXtensible HyperText Markup Language (XHTML) is an XML-based markup language for creating visual applications that users can access from their desktops or wireless devices. XHTML is the next generation of HTML 4.01 in XML, meaning the XHTML markup language can create pages that can be read by all XML-enabled devices.

If you have an existing application with HTML pages, you will have to make some simple structural changes to comply with XHTML conventions. When creating an XHTML+Voice application, your XHTML pages will remain the visual portion of the application, and at points in the interaction where voice input would help your users, you can add VoiceXML.

XHTML has replaced HTML as the supported language by the World Wide Web Consortium$^{(R)}$ (W3C), so future-proofing your Web pages by using XHTML will not only help you with multimodal applications, but will ensure that users with all types of devices will be able to access your pages correctly. For more information, locate the **XHTML specification** in Chapter 6, “References” on page 131.

What is an event handler?

An event handler specifies an action to be performed when a particular event (such as a mouse click) takes place. In XHTML+Voice, event handlers enable interaction between XHTML and VoiceXML markup. The XML Events specification specifies the XML language with the ability to uniformly integrate event listeners and associated event handlers with Document Object Model (DOM) Level 2 event interfaces.

For more information, locate the **XML Events and Document Object Model (DOM) specification** in Chapter 6, “References” on page 131.

What is a conformance document?

A conforming XHTML+Voice document must meet all of the following criteria:

- It must validate against the XML Schema found in schema listed in this document.
- The root element of the document must be html.
- The name of the default namespace on the root element must be the XHTML namespace name: “http://www.w3.org/1999/xhtml”

- If a DOCTYPE declaration is present and includes a public identifier, the DOCTYPE declaration must reference the DTD provided in this document using its Formal Public Identifier. The system identifier may be modified appropriately.

For more information, locate the **XHTML+Voice specification** in Chapter 6, “References” on page 131.

Chapter 2

Elements and attributes of the XHTML+Voice Language

This chapter provides a brief introduction to basic XHTML+Voice (X+V) concepts and constructs, and describes IBM's implementation of X+V. For a complete description of the functionality of the language, refer to the XHTML+Voice 1.2 specification, which is based on the VoiceXML 2.0 specification.

The elements and attributes included in this chapter are supported in the XHTML+Voice markup language, except when noted "not supported."

Note:

The supported XHTML elements are not included in this guide. Please refer to the specification (as well as other specifications), listed in Chapter 6, "References" on page 131.

The information in this chapter is NOT a substitute for thoroughly reading the XHTML+Voice 1.2 specification.

This chapter includes the following sections:

- "VoiceXML elements supported in X+V" on page 21.
- "XHTML+Voice tags" on page 68.
- "XML Events supported in X+V" on page 74.
- "Compatibility with the XHTML+Voice Specification" on page 77.
- "Setting MIME types" on page 80.

VoiceXML elements supported in X+V

The following elements and attributes of VoiceXML are supported and in certain cases extended by the X+V language. Refer to the **VoiceXML specification** for further information on these and other VoiceXML elements and attributes.

VoiceXML elements supported in X+V:

- "Form and Form Items" on page 22

- “Catching/Throwing Events” on page 30.
- “Speech Input” on page 36.
- “Executable Content” on page 40.
- “Speech and Audio Output” on page 46.
- “Subdialog Support” on page 57.
- “Property” on page 65.

Form and Form Items

The <form> element and its children defines a speech dialog. The form items are immediate children of the <form> element that can be visited in the main loop of the VoiceXML form interpretation algorithm (FIA). The subset of form items supported in XHTML+Voice include <field>, <record>, <subdialog>, <block>, and <initial>. The latter two elements are for procedural statements and mixed-initiative processing, respectively. The other elements are for collecting user input. The <subdialog> element has its own section, below.

<form>

Description

The <form> element is the top level element of an XHTML+Voice speech dialog. It collects user input and presents information to the user using speech. A <form> element also represents a voice handler that is activated in response to either an HTML or VoiceXML event.

Syntax

```
<form
id = "string"
xmlns = "URI">
   child elements
</form>
```

Attributes

Attribute	Description
id	The form identifier, unique to the document in which it is contained.
scope	Not supported.
xmlns	The VoiceXML 2.0 namespace URI: http://www.w3.org/2001/vxml

Parents

`<head>`

Children

`<initial> <field> <record> <block> <filled> <subdialog> <catch> <error> <noinput> <nomatch> <help> <grammar> <var> <property>`

Remarks

XHTML+Voice requires the id attribute. A voice handler, specified by the XML Events handler attribute, is activated in response to a specified HTML or VoiceXML event.

Example

This example simply says "Hello, world!" when the user clicks on the paragraph.

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:vxml="http://www.w3.org/2001/vxml"
      xmlns:ev="http://www.w3.org/2001/xml-events"
      xmlns:xv="http://www.voicexml.org/2002/xhtml+voice">
   <head>
      <title>XHTML+Voice Example</title>
      <!-- voice handler -->
      <vxml:form id="sayHello">
         <vxml:block><vxml:prompt xv:src="#hello"/>
         </vxml:block>
      </vxml:form>
   </head>
   <body>
      <h1>XHTML+Voice Example</h1>
      <p id="hello" ev:event="click" ev:handler="#sayHello">
```

```
            Hello, world!
         </p>
      </body>
   </html>
```

<initial>

Description

The user may use one or more <initial> element to prompt for form-wide information, before the user is prompted on a field-by-field basis. Like field items, initial item has prompts, catches, and event counters. Unlike field items, it has no grammars and no <filled> action. To use <initial> elements, the user needs form level grammar that can match the result to one of field items' slot name.

Syntax

```
   <initial
       name="string"
       expr="ECMAScript Expression"
   cond="ECMAScript Expression"
   />
```

Attributes

Attribute	Description
name	The name of a form item variable used to track whether the <initial> is eligible to execute; defaults to an inaccessible internal variable.
expr	An ECMAScript expression that supplies the initial value for the form item associated with this element. If the expression evaluates to something other than null or ECMAScript undefined, the element will not be run until the form item variable is explicitly cleared.
cond	An ECMAScript expression that evaluates to true or false. If false, the element is not run. If true, the element is run.

Parents

`<form>`

Children

`<audio> <catch> <enumerate> <error> <help> <noinput> <nomatch> <prompt> <property> <value>`

Remarks

None.

<field>

Description

Defines an input field in a form and formulates a speech dialog between the user and the browser.

Syntax

```
<field
    name="string"
    expr="ECMAScript Expression"
cond="ECMAScript Expression"
type=string
slot=string
modal=boolean
/>
```

Attributes

Attribute	Description
name	The form item variable in the dialog scope that will hold the result. The name must be unique among form items in the form. If the name is not unique, then a badfetch error is thrown when the document is fetched.
expr	An ECMAScript expression that supplies the initial value for the form item associated with this element. If the expression evaluates to something other than null or ECMAScript undefined, the element will not be run until the form item variable is explicitly cleared.
cond	An ECMAScript expression that evaluates to true or false. If false, the element is not run. If true, the element is run.

type	The type of field, i.e., the name of a built-in grammar type. If the specified built-in type is not supported by the platform, an error.unsupported.builtin event is thrown.
slot	The name of the grammar slot used to populate the variable (if it is absent, it defaults to the variable name). This attribute is useful in the case where the grammar format being used has a mechanism for returning sets of slot/ value pairs and the slot names differ from the form item variable names.
modal	If this is false (the default) all active grammars are turned on while collecting this field. If this is true, then only the field's grammars are enabled: all others are temporarily disabled.
xv:id	Unique document identifier for <field>.

Parents

`<form>`

Children

`<audio> <catch> <enumerate> <error> <filled> <grammar> <help> <noinput> <nomatch> <option> <prompt> <property> <value>`

Shadow Variables

The field element exposes the following shadow variables:

`name$.utterance`	`The raw string of words that were recognized.`
`name$.inputmode`	`The mode in which user input was provided (always voice).`
`name$.interpretation`	`The ECMAScript variable containing the interpretation of recognition result.`
`name$.confidence`	`The confidence level (0.0-1.0) of the matched recognition result.`

Built-in Grammar

The supported built-in types are:

boolean	The user can say positive responses such as *yes*, *true,* and *okay* or negative responses such as *no*, *false*, or *wrong*. The return value sent is a boolean true or false.
date	The user can say a day using months, days, and years. The return value sent is a string in the format yyyymmdd, and ????mmdd when the year is omitted in the spoken input.
digits	The user can say numeric integer values as individual digits (0 through 9). The return value sent is a string of one or more digits.
currency	The user can say US currency values in dollars and cents from 0 to $999,999. The return value sent is a string in the format USDdddddd.cc.
number	The user can say positive number from 0 to 999,999. The return value sent is a string of one or more digits.
phone	The user can say a telephone number, including the optional word extension. The return value sent is a string of digits without hyphens, and including and x if an extension was specified.
time	The user can say a time of day using hours and minutes in either 12- or 24-hour format as well as the word now. The return value sent is a string in the format *hhmmx*, where *x* is *a* for AM, *p* for PM or *?* if unspecified.

Remarks

On IBM Websphere Multimodal Browser release 4.1, the shadow variable name$.confidence is always 0.5.

XHTML+Voice adds an optional id attribute to the VoiceXML <field> element. The id attribute is used by the XHTML+Voice <sync> element's field attribute to uniquely specify a VoiceXML <field> element. The id attribute is prefixed with the identifier specified in the document for the XHTML+Voice namespace.

<block>

Description

A block is a form item that is used to contain executable content. The content is executed if the block's form item variable is undefined and the block's cond attribute, if present, evaluates to true.

Blocks are typically executed just once per voice form.

Syntax

```
<block>
     Welcome to my multimodal application.
</block>
```

Attributes

Attribute	Description
name	Optional name of the form item variable. The default is an internal value.
expr	Optional initial value of the form item variable. The default is ECMAScript undefined.
cond	An optional expression that must evaluate to true in order for this block to be visited. The default is true.

Parents

`<form>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt> <return> <throw> <value> <var>`

Remarks

None.

<record>

Description

Records spoken user input.

Syntax

```
<record
    name="string"
    expr="ECMAScript Expression"
cond="ECMAScript Expression"
/>
```

Attributes

Attribute	Description
name	The input item variable that will hold the recording data.
expr	An ECMAScript expression that supplies the initial value for the form item associated with this element. If the expression evaluates to something other than null or ECMAScript undefined, the element will not be run until the form item variable is explicitly cleared.
cond	An ECMAScript expression that evaluates to true or false. If false, the element is not run. If true, the element is run.
modal	Not supported.
beep	Not supported.
maxtime	Not supported.
finalsilence	Not supported.
dtmfterm	Not supported.
type	Not supported.

Parents

`<form>`

Children

`<audio> <catch> <enumerate> <error> <filled> <noinput> <prompt> <property> <value>`

Remarks

Speech recognition grammar is not supported in recording.

Catching/Throwing Events

<catch>

Description

Catches an event thrown from a VoiceXML element or interpreter.

Syntax

```
   <catch event="nomatch help">
      Please say the name of a city.
   </catch>
```

Attributes

Attribute	Description
event	A space separated list of events to catch. If empty, all events will be caught.
count	The occurrence of the event. This allows you to handle different occurrences of the same event differently. The default is 1. See the VoiceXML 2.0 specification section 5.2.2 for a complete description.
cond	A condition evaluated to determine if this catch handler will be used for the event being thrown. The default is true.

Parents

`<field> <form> <initial> <record> <subdialog>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt> <return> <throw> <value> <var>`

Remarks

None.

<throw>

Description

Throws an event in the VoiceXML form which is propagated to the HTML element which invoked the VoiceXML form.

Syntax

```
<throw event="some.event"/>
```

Attributes

Attribute	Description
event	The name of the event to throw.
eventexpr	An expression evaluating to the name of the event to throw.
message	An optional message string to provide additional information about the event being thrown.
messageexpr	An expression evaluating to the message string.

Parents

`<block> <catch> <error> <filled> <help> <if> <noinput> <nomatch>`

Children

None

Remarks

When throwing an event that is intended to be used in the HTML and not in the VoiceXML, you must still provide a <catch> handler in the VoiceXML form for that event. Otherwise, an error will be generated from the voice form. The event will be caught by the default catch handler and the text output for the default catch handler will be played. After the event is caught by the default catch handler, the voice handler will exit.

<error>

Description

This catches all error events. This is equivalent to `<catch event="error">`.

Syntax

```
<error>
   An error has occurred.
</error>
```

Attributes

Attribute	Description
count	The occurrence of the event. This allows you to handle different occurrences of the same event differently. The default is 1. See the VoiceXML 2.0 specification section 5.2.2 for a complete description.
cond	A condition evaluated to determine if this catch handler will be used for the event being thrown. The default is true.

Parents

`<field> <form> <initial> <record> <subdialog>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt>`
`<return> <throw> <value> <var>`

Remarks

None.

<help>

Description

This catches the help event which is thrown when the user says “Help.” This is equivalent to `<catch event="help">`.

Syntax

```
<help>
   Please say the name of a city.
</help>
```

Attributes

Attribute	Description
count	The occurrence of the event. This allows you to handle different occurrences of the same event differently. The default is 1. See the VoiceXML 2.0 specification section 5.2.2 for a complete description.
cond	A condition evaluated to determine if this catch handler will be used for the event being thrown. The default is true.

Parents

`<field> <form> <initial> <record> <subdialog>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt> <return> <throw> <value> <var>`

Remarks

None.

<noinput>

Description

This catches the noinput event which is thrown if a timeout occurs while waiting for user input. This is equivalent to `<catch event="noinput">`.

Syntax

```
<noinput>
   Sorry, I did not hear you.
</noinput>
```

Attributes

Attribute	Description
count	The occurrence of the event. This allows you to handle different occurrences of the same event differently. The default is 1. See the VoiceXML 2.0 specification section 5.2.2 for a complete description.
cond	A condition evaluated to determine if this catch handler will be used for the event being thrown. The default is true.

Parents

`<field> <form> <initial> <record> <subdialog>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt>`
`<return> <throw> <value> <var>`

Remarks

None.

<nomatch>

Description

This catches the nomatch event which is thrown if the user input does not match the active grammars. This is equivalent to `<catch event="nomatch">`.

Syntax

```
<nomatch>
   Sorry, I did not understand you.
</nomatch>
```

Attributes

Attribute	**Description**
count	The occurrence of the event. This allows you to handle different occurrences of the same event differently. The default is 1. See the VoiceXML 2.0 specification section 5.2.2 for a complete description.
cond	A condition evaluated to determine if this catch handler will be used for the event being thrown. The default is true.

Parents

`<field> <form> <initial> <record> <subdialog>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt> <return> <throw> <value> <var>`

Remarks

None.

Speech Input

<grammar>

Description

Defines a speech recognition grammar.

```
<grammar
   root="string"
   src="URI"
   type="media type"
   fetchhint="safe|prefetch"
   fetchtimeout="time interval"
   maxage="time interval"
   maxstale="time interval">
/>
```

Attributes

Attribute	**Description**
version	Not supported.
xml:lang	Not supported.
mode	Not supported.
root	Defines the rule which acts as the root rule of the grammar.
tag-format	Not supported.
xml:base	Not supported.
src	The URI specifying the location of the external or built-in grammar.
scope	Not supported.
type	The media type of the grammar. “application/x-jsgf” for the Java Speech Grammar Format (JSGF).
weight	Not supported.

fetchhint	Defines when the browser should retrieve content from the server. *prefetch* indicates a file may be downloaded when the page is loaded, whereas *safe* indicates a file that should only be downloaded when actually needed. If not specified, a value derived from the innermost relevant fetchhint property is used.
fetchtimeout	The time in seconds (s) or milliseconds (ms) for the browser to wait for content to be returned by the HTTP server before throwing an error.badfetch event. If not specified, a value derived from the innermost fetchtimeout property is used.
maxage	Indicates that the document is willing to use content whose age is no greater than the specified time in seconds. The document is not willing to use stale content, unless maxstale is also provided. If not specified, a value derived from the innermost relevant maxage property, if present, is used.
maxstale	Indicates that the document is willing to use content that has exceeded its expiration time. If maxstale is assigned a value, then the document is willing to accept content that has exceeded its expiration time by no more than the specified number of seconds. If not specified, a value derived from the innermost relevant maxstale property, if present, is used.

Parents

`<field> <form>`

Children

`<lexicon>`

Remarks

None.

<option>

Description

Specifies a field option.

<option> element is used as a convenient way to list a simple set of alternatives for the user within the field element.

Syntax

```
   <option
      accept="exact|approximate"
      value="string">
   text
   </option>
```

Attributes

Attribute	Description
dtmf	Not supported.
accept	When set to "exact" (the default), the text of the option element defines the exact phrase to be recognized. When set to "approximate", the text of the option element defines an approximate recognition phrase.
value	The string to assign to the field's form item variable when a user selects this option. The default assignment is the CDATA content of the <option> element with leading and trailing white space.

Parents

`<field>`

Children

#PCDATA.

Remarks

None.

<lexicon>

Description

The <lexicon> element is used to reference an external pronunciation lexicon document.

Syntax

```
   <lexicon uri="URI"
```

```
              type="media-type"/>
```

Attributes

Attribute	Description
uri	URI location of the pronunciation lexicon document.
type	The media type of the pronunciation lexicon document.

Parents

`<grammar>`

Children

None.

Remarks

None.

Example

```
<?xml version="1.0"?>
<html
xmlns="http://www.w3.org/1999/xhtml"
xmlns:vxml="http://www.w3.org/2001/vxml"
xmlns:ev="http://www.w3.org/2001/xml-events"
xmlns:xv="http://www.voicexml.org/2002/xhtml+voice"

  <head>
    <title>Lexicon Example</title>
    <!-- voice handler -->
    <vxml:form id="sayHello">
      <vxml:field name="fld1">
         <vxml:prompt xv:src="#hello">
         <vxml:grammar src="hello.gram">
              <vxml:lexicon uri="babushka.pbs"/>
          </vxml:grammar>
      </vxml:field>
    </vxml:form>
  </head>
  <body>
    <h1>Lexicon Example</h1>
    <p id="hello" ev:event="click" ev:handler="#sayHello">
      Say 'Hello babushka'.
    </p>
  </body>
</html>
```

Executable Content

<assign>

The <assign> element assigns a value of an expression to a variable. The variable can be either in the VoiceXML form or in the HTML document.

Syntax

```
<assign name="aVoiceXMLVar" expr="10"/>
<assign name="document.getElementById(‘input’).value" expr="10"/>
```

Attributes

Attribute	Description
name	Optional name of the form item variable. The default is an internal value.
expr	Optional initial value of the form item variable. The default is ECMAScript undefined.

Parents

`<block> <catch> <error> <filled> <help> <if> <noinput> <nomatch>`

Children

None

Remarks

XHTML+Voice allows the <assign> element to be used to update both XHTML control values (such as <input>, <button>, <select>) and JavaScript variables defined within an XHTML <script> element.

<clear>

Description

Resets one or more variables, including form items.

Syntax

```
<clear namelist="city state zip"/>
```

Attributes

Attribute	Description
namelist	List of variables to be reset. If this attribute is not specified, all form items are cleared.

Parents

`<block> <catch> <error> <filled> <help> <if> <noinput> <nomatch>`

Children

None.

Remarks

None.

<else>

Description

Used for conditional logic inside of an <if> element.

Syntax

```
<if cond="numberGuessed > actualNumber">
    That number is too high, try another number.
<else/>
    That number is too low, try another number.
</if>
```

Attributes

None.

Parents

`<if>`

Children

None.

Remarks

None.

<elseif>

Description

Used for conditional logic inside of an <if> element.

Syntax

```
<if cond="numberGuessed > actualNumber">
   That number is too high, try another number.
<elseif cond="numberGuessed &lt; actualNumber"/>
   That number is too low, try another number.
<else/>
   Congratulations! You guessed the number.
</if>
```

Attributes

Attribute	Description
cond	The condition that must evaluate to true or false.

Parents

`<if>`

Children

None.

Remarks

None.

<filled>

Description

Specifies an action to be performed after some combination of input items are filled.

Syntax

```
<filled>
   Your <value expr="drink"/> is coming right up.
</filled>
```

Attributes

Attribute	**Description**
mode	This attribute is used for form level filled elements only. The value can be either any or all. The default is all. If any, this action will be executed when any of the input items specified in the namelist attribute are filled. If all, this action will be executed when all of the input items have been filled.
namelist	This attribute is used for form level filled elements only. The value is a space separated list of input items to trigger on.

Parents

`<field> <form> <record> <subdialog>`

Children

`<assign> <audio> <clear> <enumerate> <if> <log> <prompt> <reprompt> <return> <throw> <value> <var>`

Remarks

None.

<if>

Description

Used for conditional logic. It can have optional <else> and <elseif> elements.

Syntax

```
<if cond="numberGuessed == actualNumber">
   You guessed the right number.
</if>
```

Attributes

Attribute	Description
cond	The condition that must evaluate to true or false.

Parents

`<block> <catch> <error> <filled> <help> <if> <noinput> <nomatch>`

Children

`<assign> <audio> <clear> <else> <elseif> <enumerate> <if> <log> <prompt>`
`<reprompt> <return> <throw> <value> <var>`

Remarks

None.

<log>

Description

Allows an application to generate a logging or debug message for debugging or performance monitoring purposes.

Syntax

```
<log>
   The error <value expr="_error"/> was thrown.
</log>
```

Attributes

Attribute	Description
label	A string that may be used, for example, to indicate the purpose of the log.
expr	An expression evaluating to a string.

Parents

`<block> <catch> <error> <filled> <help> <if> <noinput> <nomatch>`

Children

`<value>`

Remarks

The manner in which the message is displayed or logged is platform dependent. The IBM WebSphere Everyplace Multimodal Browser displays this logging information in the Voice Log window.

<var>

Description

Used to declare a variable in the VoiceXML form.

Syntax

```
<var name="player" expr="document.getElementById(‘name’).value"/>
<var name="guessCount" expr="0"/>
<var name="actualNumber" expr="Math.round(Math.random()*9)+1"/>
```

Attributes

Attribute	Description
name	The name of the variable to declare.
expr	An optional expression evaluating to the initial value of the variable. If not provided, the variable will retain its current value, if any. Variables start out with the ECMAScript value undefined if they are not given initial values.

Parents

`<block> <catch> <error> <filled> <form> <help> <if> <noinput> <nomatch>`

Children

None.

Remarks

None.

Speech and Audio Output

<audio>

Description

The <audio> element plays an audio file specified via a URL or an audio variable previously recorded. The <audio> element can have alternate content in case the audio sample is not available. The alternate content may include text, speech markup, or another audio element.

Syntax

```
   <audio
       src="URL"|expr="ECMAScript_Expression"/>
```

Attributes

Attribute	Description
src	The URI of the recorded audio file.
fetchtimeout	The time in seconds (s) or milliseconds (ms) for the voice browser to wait for content to be returned by the HTTP server before throwing an error.badfetch event. If not specified, a value derived from the innermost fetchtimeout property is used.
fetchhint	Defines when the voice browser should retrieve content from the server. *prefetch* indicates a file may be downloaded when the page is loaded, whereas *safe* indicates a file that should only be downloaded when actually needed. If not specified, a value derived from the innermost relevant fetchhint property is used.
maxage	Indicates that the document is willing to use content whose age is no greater than the specified time in seconds. The document is not willing to use stale content, unless maxstale is also provided. If not specified, a value derived from the innermost relevant maxage property, if present, is used.
maxstale	Indicates that the document is willing to use content that has exceeded its expiration time. If maxstale is assigned a value, then the document is willing to accept content that has exceeded its expiration time by no more than the specified number of seconds. If not specified, a value derived from the innermost relevant maxstale property, if present, is used.
expr	An ECMAScript expression that evaluates to a URL to be used in place of the src attribute or a variable associated with the *name* attribute of the record element.

Parents

`<audio>, <block>, <catch>, <enumerate>, <error>, <field>, <filled>, <help>, <if>, <initial>, <noinput>, <nomatch>, <record>, <subdialog>`

Children

`<audio>, <enumerate>, <value>`

Example

The following example includes both recorded audio and TTS. The location of the audio is relative to the location of the VoiceXML document that contains the audio element. If the recorded audio cannot be fetched, the VoiceXML interpreter plays back the TTS string instead.

```
<?xml version="1.0"?>
<vxml version="2.0">
<form>
   <block>
      <audio src="welcome.wav">Welcome to Online University</audio>
   </block>
</form>
</vxml>
```

The following example uses a variable and a constant string to reference an audio file. When referencing a variable, use the expr attribute instead of the src attribute.

```
<?xml version="1.0"?>
<vxml version="2.0">
<form>
   <var name="path_earcons" expr="'http://audio.en-US.onine.com/
              common-audio/'"/>
   <block>
      <audio expr="path_earcons + 'intellipause.wav'"/>
   </block>
</form>
</vxml>
```

The following example plays back TTS stored in a variable. To reference a variable containing TTS, use the value element.

```
<?xml version="1.0"?>
<vxml version="2.0">
<form>
   <var name="motd" expr="'I am sorry, Dave, but I cannot do that.'"/>
   <block>
      <audio src="sorry_dave.wav"><value expr="motd"/></audio>
   </block>
</form>
</vxml>
```

The following example attempts to retrieve a recorded audio file from **audio01.acme.net**. If the fetch fails, the interpreter attempts to retrieve an alternate recording from **audio02.acme.net**. If that fetch fails, the interpreter renders the TTS "123".

```
<vxml version="2.0">
   <form>
     <block>
       <audio src="http://audio01.acme.net/numbers/123.wav">
        <audio src="http://audio02.acme.net/numbers/123.wav">123</audio>
       </audio>
     </block>
   </form>
</vxml>
```

<enumerate>

Description

The <enumerate> element specifies a template that is applied to each choice in the order they appear in the field options. The <enumerate> element may be used within the prompt and catch elements associated with <field> elements that contain <option> elements.

Syntax

`<enumerate/>`

Attributes

None.

Parents

`<audio>, <block>, <catch>, <enumerate>, <error>, <field>, <filled>, <help>, <if>, <initial>, <noinput>, <nomatch>, <prompt>, <record>, <subdialog>`

Children

`<audio>, <enumerate>, <value>`

Example

The following example shows proper use of <enumerate> in a catch element of a form with several fields containing <option> elements.

```
<?xml version="1.0"?>
<vxml version="2.0">

  <form>
    <block>
      We need a few more details to complete your order.
    </block>
    <field name="color">
      <prompt>Which color?</prompt>
      <option>red</option>
      <option>blue</option>
      <option>green</option>
    </field>
    <field name="size">
      <prompt>Which size?</prompt>
      <option>small</option>
      <option>medium</option>
      <option>large</option>
    </field>
    <block>
      Thank you. Your order is being processed.
      <submit next="details.cgi" namelist="color size"/>
    </block>
    <catch event="help nomatch">
      Your options are <enumerate/>.
    </catch>
  </form>

</vxml>
```

<prompt>

Description

The <prompt> element queues recorded audio and synthesized text to speech in an interactive dialog.

Syntax

```
<prompt
    cond = "ECMAScript_Expression"
    count = "integer"
```

```
       timeout = "seconds_milliseconds"
       xv:src="URI+#+ID"|xv:expr="ECMAScript_Expression"/>
```

Attributes

Attribute	Description
bargein	Control whether a user can interrupt a prompt. This defaults to the value of the bargein property.
bargeintype	Not supported.
cond	A condition that determines whether or not the prompt is eligible to be played.
count	Each field maintains a prompt counter which tracks the number of times a prompt has been executed since the form was entered. The counter is reset when the VoiceXML interpreter enters the form. The *count* attribute indicates the number of times a prompt must be executed in the active field before the prompt with the specified count is selected and executed. If multiple **prompt** elements exist with the same count, the VoiceXML interpreter only executes the first one encountered in document source order. The default value is **1**.
timeout	The number of seconds (s) or milliseconds (ms) the platform waits for user input before throwing a noinput event. If multiple **prompt** tags specify a timeout, the last one is used.
xml:lang	Not supported.
xml:base	Not supported.
xv:src	Specifies a text source for speech output anywhere in the document or in an external document.
xv:expr	An ECMAScript expression that evaluates to a text source as a URI for speech output anywhere in the document or in an external document.

Parents

`<block>, <catch>, <error>, <field>, <filled>, <help>, <if>, <initial>, <noinput>, <nomatch>, <record>, <subdialog>`

Children

`<audio>, <enumerate>, <value>, <lexicon>`

Remarks

XHTML+Voice adds the optional src and expr attributes to the VoiceXML <prompt> element. These attributes are prefixed with the identifier specified in the document for the XHTML+Voice namespace.

Example

The following example shows a basic prompt that consists of both audio and text.

```
<?xml version="1.0"?>
<vxml version="2.0">

  <form>
    <prompt>
      Welcome to the Bird Seed Emporium.
      <audio src="birdsound.wav"/>
    </prompt>
  </form>

</vxml>
```

The following example shows how the count attribute of a <prompt> element is used. In the example, the first prompt element is spoken first to prompt the user to say the name of a fruit. If the user doesn't say anything or says something other than apple, orange, or pear, the combined nomatch/noinput handler is executed, and the second prompt element is executed.

```
<?xml version="1.0"?>
<vxml version="2.0">
  <form id="pick_fruit">
    <block>
       Welcome to the fruit picker.
    </block>

    <field name="fruit">
      <grammar type="application/x-gsl" mode="voice">
      <![CDATA[[
        #JSGF V1.0 iso-8859-1;
        grammar fruits;
        public <fruits> = apple {$="apple"}
```

```
                 | orange{$="orange"}
                 | pear{$="pear"}
        ]]>
        </grammar>

        <prompt count="1">
          Pick a fruit. Say apple, orange or pear.
        </prompt>

        <prompt count="2">
          Say the name of a fruit. For example, say apple.
        </prompt>

        <catch event="noinput nomatch">
          Sorry. I didn't get that.
          <reprompt/>
        </catch>

        <filled>
          You picked <value expr="fruit"/>.
        </filled>
      </field>
    </form>

  </vxml>
```

<reprompt>

Description

The <reprompt> element indicates that the appropriate <prompt> element will be selected and queued before entering a listen state in an interactive dialog.

Syntax

```
<reprompt/>
```

Attributes

None.

Parents

`<block>, <catch>, <error>, <filled>, <help>, <if>, <noinput>, <nomatch>`

Children

None.

Example

In the following example, the noinput catch expects the next form item prompt to be selected and played.

```
<?xml version="1.0"?>
<vxml version="2.0">

  <form>
    <field name="want_ice_cream">
      <grammar src="yesno.jsgf"/>
      <prompt>Do you want ice cream for dessert?</prompt>
      <prompt count="2">
        If you want ice cream, say yes.
        If you do not want ice cream, say no.
      </prompt>
      <noinput>
        I could not hear you.
        <!—- Cause the next prompt to be selected and played. -->
        <reprompt/>
      </noinput>
    </field>
  </form>

</vxml>
```

<value>

Description

The <value> element evaluates and returns an ECMAScript expression that is inserted into a prompt.

Syntax

```
<value
    expr = "ECMAScript_Expression"/>
```

Attributes

Attribute	Description
expr	Required. An ECMAScript expression evaluated and returned as text to the containing element.

Parents

`<audio>, <block>, <catch>, <enumerate>, <error>, <field>, <filled>, <help>, <if>, <initial>, <log>, <noinput>, <nomatch>, <prompt>, <record>, <subdialog>`

Children

The <value> element can be used evaluate a JavaScript expression contained in an XHTML <script> element.

Example

The following example shows how the variable assignment in a CDATA section is referenced in a prompt element.

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:vxml="http://www.w3.org/2001/vxml"
   xmlns:ev="http://www.w3.org/2001/xml-events"
   xmlns:xv="http://www.voicexml.org/2002/xhtml+voice">

   <head>

   <title>Value Example</title>
      <script type="text/javascript">
         var saythis = "Hello, world!";
      </script>
      <!-- voice handler -->
      <vxml:form id="sayHello">
         <vxml:block>
            <vxml:value expr="saythis"/>
         </vxml:block>
      </vxml:form>
   </head>
```

```
      <body>
         <h1>Value Example</h1>
      </body>
   </vxml>
```

<lexicon>

Description

The <lexicon> element is used to reference an external pronunciation lexicon document.

Syntax

```
   <lexicon uri="URI"
            type="media-type"/>
```

Attributes

Attribute	Description
uri	URI location of the pronunciation lexicon document.
type	The media type of the pronunciation lexicon document.

Parents

`<prompt>`

Children

None.

Remarks

None.

Example

```
   <?xml version="1.0"?>
   <html xmlns="http://www.w3.org/1999/xhtml"
         xmlns:vxml="http://www.w3.org/2001/vxml"
         xmlns:ev="http://www.w3.org/2001/xml-events"
         xmlns:xv="http://www.voicexml.org/2002/xhtml+voice">
      <head>
```

```
        <title>XHTML+Voice Example</title>
        <!-- voice handler -->
        <vxml:form id="sayHello">
           <vxml:block>
              <vxml:prompt xv:src="#hello">
                  <vxml:lexicon
                      uri="http://www.example.com/lex/words.file"
                      type="media-type"/>
              </vxml:prompt>
           </vxml:block>
        </vxml:form>
      </head>
      <body>
         <h1>XHTML+Voice Example</h1>
         <p id="hello" ev:event="click" ev:handler="#sayHello">
            Hello, world!
         </p>
      </body>
   </html>
```

Subdialog Support

<param>

Description

The <param> element specifies a value to pass to a subdialog element. The value specified is used to initialize a <var> declaration in the subdialog that is invoked. The initialization takes precedence over the expr attribute in <var>.

Syntax

```
   <param
       name="string"
       value="string"|expr="ECMAScript_Expression"/>
```

Attributes

Attribute	Description
name	Required. The name of the variable to initialize with the subdialog element.
expr	A ECMAScript expression that evaluates to the parameter value. *Exactly one of value and expr must be specified.*
value	The string value of the parameter. *Exactly one of value and expr must be specified.*
type	Not supported.
valuetype	Not supported.

Parents

`<subdialog>`

Children

None.

Example

Voice handler "topform" calls the "getdriverslicense" subdialog:

```
<?xml version="1.0"?>
<html xmlns=http://www.w3.org/1999/xhtml
      xmlns:vxml=http://www.w3.org/2001/vxml
      xmlns:ev=http://www.w3.org/2001/xml-events
      xmlns:xv=http://www.w3.org/2002/xhtml+voice
>
  <head>
    <vxml:form id="topform">
      <vxml:subdialog name="result"
              src="subdialog.vxml#getdriverslicense">
        <vxml:param name="birthday" expr="'2000-02-10'"/>
        <vxml:param name="age" value="100"/>
      </vxml:subdialog>
    </vxml:form>
   </head>
```

```
      <body ev:event="load" ev:handler="#topform">
        <h1>Param example</h1>
      </body>
   </html>
```

The "getdriverslicense" subdialog:

```
   <?xml version="1.0"?>
   <vxml version="2.0">
     <form id="getdriverslicense">
       <var name="birthday"/>
       <var name="age"/>
       <block>
         Hello, your birthday is <value expr="birthday"/>
         and you are <value expr="age"/> years old.
         <return/>
       </block>
     </form>
   </vxml>
```

<return>

Description

The <return> element completes execution of <subdialog> and returns control and data to the dialog that calling dialog.

Syntax

```
   <return
      event="string"|namelist="variable1 variable2 …"/>
```

Attributes

Attribute	**Description**
event	The event to be returned to the calling dialog and thrown. Exactly one of event, eventexpr, and namelist may be specified

namelist	A space-separated list of variables to be returned to the calling dialog. Exactly one of event, eventexpr, and namelist may be specified (Defaults to no variables)
eventexpr	Not supported.

Parents

`<block>, <catch>, <error>, <filled>, <help>, <if>, <noinput>, <nomatch>`

Children

None.

Remarks

XHTML+Voice allows the <return> element to run within executable content of a top level voice handler (i.e., one that is not called as a subdialog). The <return> element within executable content of a top level voice handler is used to end the execution of the voice handler.

When the <return> element is specified within a top-level voice form, its namelist attribute has no meaning and is ignored. However, either the event or eventexpr attribute can be used to return a VoiceXML event to the XHTML container.

Example

Voice handler topform calls the account subdialog:

```
   <?xml version="1.0"?>
   <?xml version="1.0"?>
   <html xmlns=http://www.w3.org/1999/xhtml
         xmlns:vxml=http://www.w3.org/2001/vxml
         xmlns:ev=http://www.w3.org/2001/xml-events
         xmlns:xv=http://www.w3.org/2002/xhtml+voice
   >
     <head>
     <vxml:form id="topform">
       <vxml:subdialog name="result"
                 src="subdialog.vxml#account">
         <vxml:filled>
           Your account number is
           <vxml:value expr="result.acctnum"/>.  Your phone
```

```
          is <vxml:value expr="result.acctphone"/>.
        </vxml:filled>
      </vxml:subdialog>
    </vxml:form>
    </head>
    <body ev:event="load" ev:handler="#topform">
      <h1>Return example</h1>
    </body>
  </html>
```

The account subdialog:

```
  <?xml version="1.0"?>
  <vxml version="2.0">
    <form id="account">
      <field name="acctnum" type="digits">
        <prompt> What is your account number? </prompt>
      </field>
      <field name="acctphone" type="phone">
        <prompt> What is your home telephone number? </prompt>
        <filled>
          <return namelist="acctnum acctphone"/>
        </filled>
      </field>
    </form>
  </vxml>
```

<subdialog>

Description

The <subdialog> element invokes another VoiceXML form as a subdialog of the current one. The subdialog form is a reusable dialog that allows values to be returned. The subdialog runs in a new application scope with all variables initialized. Values can be passed into the subdialog using <param> child elements, and the subdialog must contain <var> variable declaration for each parameter defined by <param>. The original dialog continues execution only when the subdialog executes the <return> element. The values returned by <return> are available as properties of the <subdialog> form item variable.

XHTML+Voice requires the <subdialog> element's src or srcexpr attribute to reference the subdialog form explicitly with the value of the form's id attribute appended to the URI as a fragment identifier. If the subdialog form is in the same document as the form that calls the subdialog, then the src or evaluated srcexpr attribute will contain only the fragment identifier referencing the value of the subdialog form's id attribute.

The *namelist* attribute is relevant only if the source of the <subdialog> element is a server-side script (e.g. CGI).

Only one of either the *src* or *srcexpr* attribute can be used to reference a subdialog form.

Syntax

```
<subdialog
    name="string"
    expr="ECMAScript_Expression"
    cond="ECMAScript_Expression"
    namelist="variable1 variable2 ..."
    src="URI"|srcexpr="ECMAScript_Expression"
    fetchhint="safe"
    fetchtimeout="time_interval"
    maxage="integer"
    maxstale="integer">
  child elements
</subdialog>
```

Attributes

Attribute	Description
name	The name of this subdialog, representing a variable that can be referenced anywhere within the subdialog's form. The results returned from the subdialog can be retrieved as properties of the subdialog variable: name.returnVariable.
expr	An ECMAScript expression that supplies the initial value for the form item associated with this element. If the expression evaluates to something other than null or ECMAScript undefined, the element will not be run until the form item variable is explicitly cleared.
cond	An ECMAScript expression that evaluates to true or false. If false, the element is not run. If true, the element is run.
namelist	A space-separated list of variables to be submitted to the referenced subdialog (VoiceXML form).
src	The URI of the containing document appended with the fragment identifier of the subdialog (VoiceXML form).
srcexpr	An ECMAScript expression that evaluates to the URI of the containing document appended with the fragment identifier of the subdialog.
method	Not supported.
enctype	Not supported.
fetchaudio	Not supported.
fetchtimeout	The time in seconds (s) or milliseconds (ms) for the voice browser to wait for content to be returned by the HTTP server before throwing an error.badfetch event. If not specified, a value derived from the innermost fetchtimeout property is used.
fetchhint	Defines when the voice browser should retrieve content from the server. *prefetch* indicates a file may be downloaded when the page is loaded, whereas *safe* indicates a file that should only be downloaded when actually needed. If not specified, a value derived from the innermost relevant fetchhint property is used.

maxage	Indicates that the document is willing to use content whose age is no greater than the specified time in seconds. The document is not willing to use stale content, unless maxstale is also provided. If not specified, a value derived from the innermost relevant maxage property, if present, is used.
maxstale	Indicates that the document is willing to use content that has exceeded its expiration time. If maxstale is assigned a value, then the document is willing to accept content that has exceeded its expiration time by no more than the specified number of seconds. If not specified, a value derived from the innermost relevant maxstale property, if present, is used.

Parents

`<form>`

Children

<audio>, <catch>, <enumerate>, <error>, <filled>, <help>, <noinput>, <nomatch>, <param>, <prompt>, <property>, <value>

Example

Voice handler topform calls the account subdialog:

```
   <?xml version="1.0"?>
   <?xml version="1.0"?>
   <html xmlns=http://www.w3.org/1999/xhtml
         xmlns:vxml=http://www.w3.org/2001/vxml
         xmlns:ev=http://www.w3.org/2001/xml-events
         xmlns:xv=http://www.w3.org/2002/xhtml+voice>
     <head>
     <vxml:form id="topform">
       <vxml:subdialog name="result"
                 src="subdialog.vxml#account">
         <vxml:filled>
           Your account number is
           <vxml:value expr="result.acctnum"/>.  Your phone
          is <vxml:value expr="result.acctphone"/>.
         </vxml:filled>
       </vxml:subdialog>
     </vxml:form>
     </head>
     <body ev:event="load" ev:handler="#topform">
```

```
      <h1>Subdialog example</h1>
    </body>
  </html>
```

The account subdialog:

```
  <?xml version="1.0"?>
  <vxml version="2.0">
    <form id="account">
      <field name="acctnum" type="digits">
        <prompt> What is your account number? </prompt>
      </field>
      <field name="acctphone" type="phone">
        <prompt> What is your home telephone number? </prompt>
        <filled>
          <return namelist="acctnum acctphone"/>
        </filled>
      </field>
    </form>
  </vxml>
```

Property

<property>

Description

The <property> element is used to set a speech parameter for the VoiceXML form or form input item. The parameter is a value that affects platform behavior, such as the recognition process, timeouts, caching policy, etc. Please refer to the list properties supported by XHTML+Voice below.

Syntax

```
  <property
      name="string"
      value="string"/>
```

Attribute

Attribute	Description
name	The property name. Required.
value	The property value. Required.

Parents

`<field> <form> <initial> <record> <subdialog>`

Children

None.

Example

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:vxml="http://www.w3.org/2001/vxml"
   xmlns:ev="http://www.w3.org/2001/xml-events"
   xmlns:xv="http://www.w3.org/2002/xhtml+voice">
   <head>
      <vxml:form id="topform">
         <vxml:property name="fetchtimeout" value="60s"/>
         <vxml:subdialog name="result"
            src="subdialog.vxml#getdriverslicense">
         <vxml:param name="birthday" expr="'2000-02-10'"/>
         <vxml:param name="age" value="100"/>
         </vxml:subdialog>
      </vxml:form>
   </head>
   <body ev:event="load" ev:handler="#topform">
      <h1>Param example</h1>
   </body>
</html>
```

Tables of properties and default values

Table 1 lists the VoiceXML properties that apply to XHTML+Voice. Properties with a strike-through are not supported in Multimodal Tools.

Table 1. List of properties

audiofetchhint	grammarfetchhint
audiomaxage	grammarmaxage
audiomaxstale	grammarmaxstale
bargein	~~incompletetimeout~~
~~bargeintype~~	inputmodes
~~completetimeout~~	~~interdigittimeout~~
confidencelevel	maxnbest
documentfetchhint	~~maxspeechtimeout~~
documentmaxage	~~sensitivity~~
documentmaxstale	~~speedvsaccuracy~~
fetchaudio	~~termchar~~
fetchaudiodelay	~~termtimeout~~
fetchaudiominimum	timeout
fetchtimeout	universals

Table 2 lists the default property values for all platforms.

Table 2. Table of default property values for all platforms:

bargein	true
timeout	infinite
audiofetchhint	prefetch
audiomaxage	infinite
audiomaxstale	0s
documentfetchhint	safe
documentmaxage	infinite
documentmaxstale	0s
grammarfetchhint	prefetch
grammarmaxage	infinite
grammarmaxstale	0s
fetchtimeout	30s
com.ibm.speech.asr.vocabtype	detailedmatch
maxnbest	0.2
confidencelevel	0.2
confidence shadow variable of the <field> element	0.5

XHTML+Voice tags

The X+V markup language offers the following elements and attributes. Refer to the **XHTML+Voice specification** for further information on these and other X+V elements and attributes.

<sync>

Description

The <sync> element adds support for synchronization of data entered via either speech or visual input. It binds the value property of an XHTML form input to the VoiceXML field with the given id attribute value. This means several things:

1) Speech dialog results are returned to both the VoiceXML field and the XHTML <input> element.
2) Keyboard data entered into the <input> element updates both the VoiceXML field and the XHTML <input> element.
3) Keyboard data entered into the <input> element satisfies the guard condition on the VoiceXML field.
4) For an active VoiceXML form with multiple fields, if the user gives focus to the input field, the FIA is instructed to visit the referenced VoiceXML field as the next item.

Syntax

```
<xv:sync xv:input="string" xv:field="URI+#+ID" xv:html-form-id="#+ID"/>
```

Attributes

Attribute	**Description**
input	The name of an XHTML form input field.
field	A URI reference to a field ID within a VoiceXML form.
html-form-id	A reference to the ID of the XHTML form enclosing the input field.

Parents

`<head>`

Children

None.

Remarks

The <sync> element does not activate a voice handler and the referenced XHTML input field is not cleared if data is already there.

Only changes made while a VoiceXML form is active are synchronized. An existing XHTML input value does not update the synchronized VoiceXML <field> when the VoiceXML form is activated.

Standard Grammars for XHTML Controls

The <sync> element synchronizes the results between a VoiceXML <field> and an XHTML input control, or group of controls. A VoiceXML field is filled when the user's utterance matches a word or phrase in the field's grammar. The grammar, along with [Semantic Interpretation], determines how the VoiceXML field is filled and can be used to determine how a field's contents updates an arbitrary XHTML control, or group of controls. Standardizing the grammars enables a straight-forward algorithm for updating an HTML input control based on the contents of a VoiceXML <field>.

The following standard grammars are used with the <sync> element for synchronizing HTML controls with the following property types: radio button and radio group, check box and check-box group, hidden, password, file, text, text area, select-one, select-multiple, submit, reset, and button.

Here is an example of a grammar for a single selection list (i.e., <select>) and a radio group (i.e., multiple HTML inputs of type "radio" with the same name).

```
<![CDATA[
  #JSGF V1.0;
  grammar crust;
  public <crust> = thin | medium | thick | chicago [style] | cheese;
]]>
```

Here is an example of a grammar for a multiple selection list (i.e., <select multiple="multiple">) and a checkbox group (i.e., multiple HTML inputs of type "checkbox" with the same name). Each selected item is pushed onto an array. The filled VoiceXML field is an array containing the selected items.

```
<![CDATA[
  #JSGF V1.0;
  grammar meat_toppings;
  <meats> = bacon | chicken | ham | meatball | sausage | pepperoni;
  public <toppings> = <NULL> { $= new Array; }
                  ( <meats> [and] { $.push($meats) } )+;
]]>
```

Here is an example of a grammar for a single radio button, check box, or button (button includes the submit and reset buttons). For the radio button or check box, the "checked" attribute is toggled according to the semantic interpretation tag contained in the filled VoiceXML field. For the button input type, a semantic interpretation value of "true" causes the button to be clicked.

```
<![CDATA[
  #JSGF V1.0;
  grammar pizza_extra;
  public <yesno> = no {$=false} | nope {$=false} | next {$=false} |
                               yes {$=true} | {$=true};
]]>
```

The grammar for the text, text area, password, hidden, and file input types does not require any semantic interpretation. The contents of the filled VoiceXML field is set to the value attribute of these input types. Here is an example:

```
<![CDATA[
  #JSGF V1.0;
  grammar one_twenty;
  public <onetotwenty> =
1|2|3|4|5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20;
]]>
```

The user should always have the option of saying "none" or "next" to decline updating the HTML control. This is supported by adding a grammar to the VoiceXML field which is outside of the standard grammar used for that field. The sample code below shows an example of a grammar, added to the grammar for a multiple selection list, that allows the user to say "none" or "skip":

```
<grammar>
  <![CDATA[
    #JSGF V1.0;
    grammar meat_toppings;
    <meats> = bacon | chicken | ham | meatball | sausage | pepperoni;
    public <toppings> = <NULL> { $= new Array; }
                  ( <meats> [and] { $.push($meats) } )+;
  ]]>
</grammar>
<grammar>
  <![CDATA[
    #JSGF V1.0;
    grammar no_sel;
    public <no_sel> = none | next | skip;
  ]]>
</grammar>
```

Note that the above example grammars are JSGF, but the grammars can be in any standard format supported by VoiceXML 2.0.

Example

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:vxml="http://www.w3.org/2001/vxml"
   xmlns:ev="http://www.w3.org/2001/xml-events"
   xmlns:xv="http://www.w3.org/2002/xhtml+voice">

   <head><title>Sync Example</title>
      <xv:sync xv:input="in1" xv:field="#result"/>
      <vxml:form id="topform">
         <vxml:field name="result xv:id="result">
            <vxml:prompt>Say a name</vxml:prompt>
            <vxml:grammar src="result.gram"/>
         </vxml:field>
      </vxml:form>
   </head>
   <body ev:event="load" ev:handler="#topform">
      <h1>Sync example</h1>
      <form action="cgi/result.cgi">
         Result: <input type="text name="in1"/>
      </form>
   </body>
</html>
```

<cancel>

Description

The <cancel> element allows a document author to cancel a running speech dialog. It is a stand-alone element with no content that can be referenced as an XML Events event handler.

Syntax

```
<xv:cancel id="string" xv:voice-handler="URI+#+ID"/>
```

Attributes

Attribute	Description
id	Unique document identifier.
voice-handler	A URI reference to a VoiceXML form ID.

Parents

`<head>`

Children

None.

Remarks

The id attribute is required. The optional voice-handler attribute references the id attribute of a voice handler form. If the voice-handler attribute is omitted, then the currently running speech dialog is canceled. If voice-handler is specified, then only the specified voice handler is canceled.

Example

```
<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml"
   xmlns:vxml="http://www.w3.org/2001/vxml"
   xmlns:ev="http://www.w3.org/2001/xml-events"
   xmlns:xv="http://www.w3.org/2002/xhtml+voice">

   <head><title>Sync Example</title>
      <xv:sync xv:input="in1" xv:field="#result"/>
      <xv:cancel id="can1" voice-handler="#topform"/>

      <vxml:form id="topform">
         <vxml:field name="result xv:id="result">
            <vxml:prompt>Say a name</vxml:prompt>
            <vxml:grammar src="result.gram"/>
         </vxml:field>
      </vxml:form>
   </head>
   <body ev:event="load" ev:handler="#topform">
      <h1>Sync example</h1>
```

```
        <form action="cgi/result.cgi">
           Result:  <input type="text name="in1"/><br/>
                     <input type="reset" ev:event="click"
                       ev:handler="#can1"/>
        </form>
     </body>
  </html>
```

XML Events supported in X+V

The following elements and attributes of XML Events are supported and expanded by the X+V language. Refer to the **XML Events specification** for further information on these and other XML Events elements and attributes.

<listener>

Description

Element listener supports a subset of the DOM's EventListener interface. It is used to declare event listeners and register them with specific nodes in the DOM.

Syntax

Attributes

Attribute	Description
event	(NMTOKEN) The required event attribute specifies the event type for which the listener is being registered. As specified by DOM2EVENTS, the value of the attribute should be an XML Name XML.
observer	(IDREF) The optional observer attribute specifies the id of the element with which the event listener is to be registered. If this attribute is not present, the observer is the element that the event attribute is on, or the parent of that element.
target	(IDREF) The optional target attribute specifies the id of the target element of the event (i.e., the node that caused the event). If this attribute is present, only events that match both the event and target attributes will be processed by the associated event handler. Clearly because of the way events propagate, the target element should be a descendent node of the observer element, or the observer element itself. Use of this attribute requires care; for instance, if you specify <listener event="click" observer="para1" target="link1" handler="#clicker"/> where 'para1' is some ancestor of the following node <a id="link1" href="doc.html">The <em>draft</em> document</a> and the user happens to click on the word "draft", the <em> element, and not the <a>, will be the target, and so the handler will not be activated; to catch all mouse clicks on the <a> element and its children, use observer="link1", and no target attribute.
handler	(URI) The optional handler attribute specifies the URI reference of a resource that defines the action that should be performed if the event reaches the observer. If this attribute is not present, the handler is the element that the event attribute is on.

phase	The optional phase attribute specifies when (during which DOM 2 event propagation phase) the listener will be activated by the desired event. **capture** Listener is activated during capturing phase. **default** Listener is activated during bubbling or target phase. The default behavior is phase="default". Note that not all events bubble, in which case with phase="default" you can only handle the event by making the event's target the observer.
propagate	The optional propagate attribute specifies whether after processing all listeners at the current node, the event is allowed to continue on its path (either in the capture or the bubble phase). **stop** Event propagation stops **continue** Event propagation continues (unless stopped by other means, such as scripting, or by another listener). The default behavior is propagate="continue".
defaultAction	The optional defaultAction attribute specifies whether after processing of all listeners for the event, the default action for the event (if any) should be performed or not. For instance, in XHTML the default action for a mouse click on an <a> element or one of its descendents is to traverse the link. **cancel** If the event type is cancelable, the default action is cancelled. **perform** The default action is performed (unless cancelled by other means, such as scripting, or by another listener). The default value is defaultAction="perform". Note that not all events are cancelable, in which case this attribute is ignored.
id	(ID) The optional id attribute is a document-unique identifier. The value of this identifier is often used to manipulate the element through a DOM interface.

Compatibility with the XHTML+Voice Specification

The Multimodal Toolkit and Multimodal Browser in this release are based on the specifications listed in Chapter 6, “References” on page 131.

However, this section describes how the X+V in this release differs from specifications.

XHTML+Voice

For details about XHTML+Voice, see the location of the XHTML+Voice 1.2 specification. In addition, this version of the Multimodal Toolkit and Multimodal Browser supports only JSGF grammars. See the exceptions for JSGF grammars below.

XHTML

For details about XHTML, see the link to the XHTML 1.0 specification. The Multimodal Tools supports the Transitional DTD described in Appendix A.1.2.

VoiceXML

For details about VoiceXML, see the link to the VoiceXML 2.0 specification.

Table 3 lists the VoiceXML elements that are included in the XHTML+Voice spec, along with the attributes for each element. Attributes with a strike-through are not supported in Multimodal Tools.

Table 3. VoiceXML elements/attributes supported in XHTML+Voice

Element	Attributes
`<assign>`	name, expr
`<audio>`	src, fetchtimeout, fetchhint, maxage, maxstale, expr
`<block>`	name, expr, cond
`<catch>`	event, count, cond
`<clear>`	namelist
`<else>`	
`<elseif>`	cond
`<enumerate>`	
`<error>`	count, cond
`<field>`	name, expr, cond, type, slot, modal
`<filled>`	mode, namelist
`<form>`	id, ~~scope~~, xmlns
`<grammar>`	~~version~~, ~~xml:lang~~, ~~mode~~, root, ~~tag-format~~, ~~xml:base~~, src, ~~scope~~, type, ~~weight~~, fetchhint, fetchtimeout, maxage, maxstale
`<help>`	count, cond
`<if>`	cond
`<initial>`	name, expr, cond
`<lexicon>*`	uri
`<log>`	label, expr
`<noinput>`	count, cond
`<nomatch>`	count, cond
`<option>`	~~dtmf~~, accept, value
`<param>`	name, expr, value, ~~valuetype~~, ~~type~~
`<prompt>`	bargein, ~~bargeintype~~, cond, count, timeout, ~~xml:lang~~, ~~xml:base~~
`<property>`	name, value
`<record>`	name, expr, cond, ~~modal~~, ~~beep~~, ~~maxtime~~, ~~finalsilence~~, ~~dtmfterm~~, ~~type~~
`<reprompt>`	

`<return>`	event, ~~eventexpr~~, ~~message~~, ~~messageexpr~~, namelist
`<subdialog>`	name, expr, cond, namelist, src, srcexpr, ~~method~~, ~~enctype~~, ~~fetchaudio~~, fetchtimeout, fetchhint, maxage, maxstale
`<throw>`	event, eventexpr, message, messageexpr
`<value>`	expr
`<var>`	name, expr

All elements except for <lexicon> are described in the VoiceXML 2.0 specification (see the References section). For more information on the <lexicon> element, see the online Help topic Creating a pronunciation pool file (Help > Help Contents > Multimodal Tools > Pronunciations > Tasks).

In addition, the Multimodal Toolkit currently supports only 11 kHz, 16-bit mono WAV audio files. The Multimodal Browser supports 11 kHz, 22 kHz, and 44 kHz 16-bit mono and stereo WAV audio files.

Speech dialog results may be accessed from XHTML in one of the following ways:

- The VoiceXML standard application variables are available to an XHTML+Voice application as global ECMAScript variables. Each variable listed is an array of elements [0..i..n], where each element represents a possible result:
    - application.lastresult$[i].confidence
    - application.lastresult$[i].utterance
    - application.lastresult$[i].inputmode
    - application.lastresult$[i].interpretation
- The XHTML+Voice <sync> element is described in XHTML+Voice Extension Module.

JSGF

For details about JSGF grammars, see the link to the JSGF specification in the References section at the end of this document. We support the specification with the following exceptions:

- Do not use qualified or fully-qualified rulenames in a grammar.
- Rulenames cannot contain the following punctuation symbols:
  + - : ; , = | / \ ( ) [ ] @ # % ! ^ & ~

- The "import" command must specify a URI, plus a rulename or asterisk. For example: "import <http://www.yourcompany.com/grammar.jsgf.rulename>" or "import <http://www.yourcompany.com/grammar.jsgf.*>"

SISR

The Multimodal Browser supports the SISR specification, with the exception of semantic interpretation literals (Section 3.2.2) and global variable declarations and initialization (Section 4.3).

Setting MIME types

Some servers will send only the files that they recognize. Files must be defined using MIME types. Table 4 includes valid file extensions and the corresponding MIME Content types.

Table 4. MIME types

Extension	Content type
`.mxml, .jsm`	`application/x-xhtml+voice+xml`
`jsgf, .jsg, .gram, .gra`	`application/x-jsgf`

The first line is the "official" X+V (XHTML + VoiceXML) document MIME type. However, in the traditional spirit of trying to render whatever the author writes, the browsers are enabling X+V for the standard html MIME type in the first line.

The second line is for Java Speech Grammar Format. Grammar files are only of interest when they are pulled in to X+V as external resources. Generally, in the JSP programming model, the grammars will be inlined in the XHTML+Voice language. See the VoiceXML spec for the grammar tag.

Chapter 3

Adding Grammars

At each point in the multimodal application where users can respond with words, the application will rely on the IBM speech recognition engine to "hear," or recognize, the spoken input. The engine can detect and interpret words and phrases, as long as the programmer tells the engine what words and phrases to expect. The programmer does this by including the expected words in "grammars."

Every word that you want the system to recognize, even "Yes" and "No," must be included in a grammar. Your ability to design the application with simple, tightly controlled grammars will contribute significantly to its usability and customer satisfaction.

This chapter includes the following sections:

- "What is a grammar?" on page 81.
- "Creating JSGF grammars" on page 84.
- "Adding semantic interpretation" on page 87.
- "Creating a pronunciation pool file" on page 88.
- "Importing Reusable Dialog Components" on page 90.
- "Adding mixed initiative applications and form level grammars" on page 90.

Note:

In addition to the grammar specifications referenced in this chapter, for more information on grammars used in VoiceXML applications, see the **VoiceXML Programmer's Guide** (pgmguide.pdf).

What is a grammar?

A grammar is an enumeration, in compact form, of the set of utterances—words and phrases—that constitute the acceptable user response to a given prompt. All the words that you want the speech recognition engine to recognize when users respond to your application must be included in a grammar.

A grammar can be as simple as a list of words, or it can be designed with more flexibility and variability so that it has the capability to recognize natural language, such as phrases and sentences. In the application, as an end-user says words or phrases, the speech recognition engines compare each word or phrase spoken by an end-user with the words and phrases in the active grammar, which can define several ways to say the same thing. The design of grammars is important to achieving accuracy.

Each type of grammar in a voice application uses a particular syntax, or set of rules, to define the words and phrases that can be recognized by the engine. Multimodal browsers support the following grammar formats:

- Java$^{(TM)}$ Speech Grammar Format (JSGF) grammars
- Reusable Dialog Components (subdialogs included with the Multimodal Toolkit)
- Additional or customized pronunciations using pronunciation pool files

Grammars also allow for the specification of semantic return values using the W3C **Semantic Interpretation for Speech Recognition (SISR) 1.0 specification**. Locate the SISR specification in Chapter 6, “References” on page 131.

Grammar considerations

Grammar considerations include the following:

- **Inline vs. external grammars**. You can create grammars inline or in external files (additional information is included in this chapter).
    - An inline grammar is written within the application. For example, create an inline grammar if you want the words to be language-specific or available only at that response point. However, inline grammars are not recommended because you cannot reuse an inline grammar and, if you use the Multimodal Toolkit, the functions provided by the grammar editor are not available, such as validation, content assist, formatting, and execution in the grammar test tool.
    - An external grammar consists of a separate file, such as a JSGF file, that is referenced from the application. For example, create an external grammar if you want the words to be language neutral or if you want to reuse the grammar in other parts of the application.
    - Both external and inline grammars use the <vxml:grammar> tag in the VoiceXML part of the application.

- **Default vs. customized pronunciations**. The IBM speech recognition engine contains default pronunciations for thousands of words, so your grammar will not have to specify expected pronunciations of all words. However, default pronunciations are sometimes based on the spelling and not the common pronunciation. In this case, if testing warrants it, you can customize pronunciations and add them in pool files to your application. For more information, see "Creating a pronunciation pool file" on page 88.
- **Generic vs. customized grammars**. When you write your application, you can use the flexible, but generic, built-in grammars and create one or more of your own. Whether you use a built-in grammar or your own customized grammar, you must decide when each grammar should be active. The speech recognition engine uses only the active grammars to define what it listens for in the incoming speech.
- **Minimizing complexity and size**. Remember that the size and complexity of the grammar will affect performance. During testing, when you click in a field and press the Push-to-Talk button, and it takes a long time to hear the tone, it might mean that your grammar is too complex. Try simplifying the grammar and reducing the number of words.

Using fast match grammar

To improve the recognition response time on a large list grammar (greater than 500 words), you can direct the browser to compile the grammar in fast match mode by setting the property of "com.ibm.speech.asr.vocabtype" to "fastmatch" (default setting is "detailedmatch"). To do this, add the value in the <vxml:property> tag within a <vxml:field> or <vxml:form> element, as shown in the following example:

```
<vxml:property name="com.ibm.speech.asr.vocabtype"
              value="fastmatch"/>
```

The fast match grammar should not contain any branch or contain fewer than 500 words. (Doing so would degrade performance.) If the grammar contains a branch or contains fewer than 500 words, you should always use "detailedmatch."

Only one fast match grammar should be enabled at any given point. Enabling more than one fast match grammar simultaneously will degrade performance.

Grammar features available in the Multimodal Toolkit

The Multimodal Toolkit includes easy-to-use grammar editors that help create, edit, and validate grammars, as well as convert grammars from one format to another.

In addition, the toolkit provides a “Generate Sync” wizard that automatically connects the grammar to the XHTML input element using the XHTML+Voice <sync> tag.

For more information on these and other features, after you install the toolkit, open the online help (from the Help menu, select Help contents > Multimodal developer information > Grammar information).

Creating JSGF grammars

Java Speech Grammar Format (JSGF) is a platform-independent, vendor-independent textual representation of grammars for use in speech recognition. The JSGF format, developed by Sun Microsystems$^{(TM)}$, Inc., adopts the style and conventions of the Java programming language, in addition to use of traditional grammar notations.

For more information, see the **Java Speech Grammar Format specification**. Locate the JSGF specification in Chapter 6, “References” on page 131.

Adding an external JSGF grammar

The default extension for a JSGF grammar file is .jsgf. Other valid extensions include .jsg, .gram, and .gra. The following sample code shows a basic JSGF grammar:

```
#JSGF V1.0 iso-8859-1;

grammar lastnames;

public <lastnames> = Nichols
                      | Smith
                      | Olson
                      ;
```

Type the grammar source code in a text editor.

- Between the equal sign and the semicolon, type a complete list of all the single words that you expect users say, pressing Enter between each word.
- For phrases, add each word in the phrase individually, but without duplication.
- Do NOT use quotation marks or apostrophes.
- Make sure that the last entry is followed immediately by the semicolon.

The following sample code shows a call to an external JSGF grammar file in the VoiceXML part of the multimodal application:

```
<vxml:grammar src="lastnames.jsgf"/>
```

Adding an inline JSGF grammar

To add an inline JSGF grammar, make sure that the grammar is correct and valid. To do this, use the CDATA tag within the <vxml:grammar> tag, as shown in the JSGF example below:

```
<vxml:grammar>
   <![CDATA[
      #JSGF V1.0;
      grammar lastnames;
      public <lastnames> = Nichols | Smith | Olson ;
      ]]>
</vxml:grammar>
```

Exceptions to the JSGF specification

The XHTML+Voice language supports the JSGF specification, with the following exceptions:

- Do not use qualified or fully-qualified rulenames in a grammar.
- Rulenames cannot contain the following punctuation symbols:
  + - : ; , = | / \ ( ) [ ] @ # % ! ^ & ~

The "import" command must specify a URI, plus a rulename or asterisk.

For example,

```
import <http://www.yourcompany.com/grammar.jsgf.rulename>
```

or

```
import <http://www.yourcompany.com/grammar.jsgf.*>
```

Importing a JSGF grammar into another JSGF grammar

To import a JSGF grammar into another JSGF grammar, add the import statement, as shown in the following example, which imports the namelist.jsgf grammar into the names.jsgf grammar.

```
names.jsgf

#JSGF V1.0;
grammar names;
import <namelist.jsgf.*>;
public <names> = <first> <last> | <last> <first> ;
```

```
namelist.jsgf

#JSGF V1.0;
grammar namelist;
public <first> = Tom | Chris | Ann ;
public <last> = Nichols | Smith | Olson ;
```

In the examples above, the import statement: **import <namelist.jsfg.*>;**

makes the <first> and <last> public rules in namelist.jsgf visible to the names.jsgf grammar.

Adding semantic interpretation

You might want to add semantic interpretation tags to your grammar. Semantic interpretation tags can be used to translate recognition results into a format that is more useful to your application. For example, you may want to translate a recognition result into a language-independent format, or reformat dates and numbers into a standard notation.

The Semantic Interpretation for Speech Recognition (SISR) specification describes the format of semantic interpretation tags and specifies how these tags will be used to compute a semantic interpretation result. Section 3.1.6 of the VoiceXML 2.0 spec further describes how that semantic interpretation result will be used to fill in one or more VoiceXML fields.

For more information, see the **Semantic Interpretation for Speech Recognition (SISR) specification**. Locate the SISR specification in Chapter 6, “References” on page 131.

Exceptions to the SISR specification

The Multimodal Browser supports the SISR specification, with the exception of semantic interpretation literals (Section 3.2.2) and global variable declarations and initialization (Section 4.3).

Creating a pronunciation pool file

If you add words to your grammars that are not vocabulary of the speech engine, the IBM Speech Recognition engine automatically creates a default pronunciation based on the spelling of the word.

If you want to customize pronunciations, such as alternative pronunciations or editing a pronunciation, you can add the modified pronunciations in a pool file (file extension .pbs), and add the pool file to the project using the <vxml:lexicon> tag within the body of the <vxml:grammar> tag.

For best results, a pool file should be associated with only one JSGF grammar and should contain all the customized pronunciations for that grammar. You can create a pool file for each JSGF grammar, if needed.

The following sample code shows an example of a pool file created using the IPA phonology. The example includes alternative pronunciations for a last name.:

```
smith S M IH TH
smith S M AY TH
jones JH OW N Z
davis D EY V IX S
```

Adding a pool file for an external grammar

The example below shows how the lastnames.grxml grammar file calls the customized pronunciations specified in the lastnames.pbs pool file.

```
<vxml:grammar src="lastnames.grxml">
     <vxml:lexicon uri="lastnames.pbs"/>
     <vxml:lexicon uri="lastnames.pbs"/>
</vxml:grammar>
```

Adding a pool file for an inline grammar

The example below shows how to create an inline grammar and call the customized pronunciations specified in the lastnames.pbs specified in the lexicon tag.

```
<vxml:grammar>
     <![CDATA[
          #JSGF V1.0;
          grammar lastnames;
          public <lastnames> = Smith | Jones | Davis ;
          ]]>
          <vxml:lexicon uri="lastnames.pbs"/>
</vxml:grammar>
```

Pronunciation features available in the Multimodal Toolkit

The Multimodal Toolkit includes easy-to-use tools that help test and edit pronunciations, as well as create pronunciation pool files to add to your application.

For more information on these and other features, after you install the toolkit, open the online help (from the Help menu, select Help contents > Multimodal developer information > Pronunciation information).

Importing Reusable Dialog Components

The Multimodal Toolkit includes IBM Reusable Dialog Components, which are basic sets of subdialogs that provide VoiceXML source code for common functions, enabling you to quickly and easily add these functions to your applications.

Subdialogs are simple pieces of code that provide basic functions used in typical VoiceXML applications. You can call them from multiple places in the application with only a single instance of code, and you can modify them globally or on a one-time or per-instance basis.

In the toolkit, a wizard can automatically import the dialog component into the application, which adds the external grammar file (JSGF) into the project and creates the reference to the external grammar in the X+V file.

For more information on importing dialog components and other features available in the X+V editor, after you install the toolkit, open the online help (from the Help menu, select Help contents > Multimodal developer information > X+V editor).

Adding mixed initiative applications and form level grammars

In a machine-directed application, the computer controls all interactions by sequentially executing each form item a single time.

However, X+V also supports mixed-initiative applications in which either the system or the user can direct the conversation. One or more grammars in a mixed-initiative application may be active outside the scope of its own dialog; to achieve this, you can use the <link> element and code them as either form-level grammars (scope="dialog") or document-level grammars (scope="document") defined in the application root document. If the user utterance matches an active grammar outside of the current dialog, the application jumps to the dialog specified by the <link>.

When you code a mixed-initiative application, you may also use one or more <initial> elements to prompt for form-wide information, before the user is prompted on a field-by-field basis.

Form-level grammars allow a greater flexibility and more natural responses than field-level grammars because the user can fill in the fields in the form in any order and can fill more than one field as a result of a single utterance. For example, the following city/state grammar:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<grammar version="1.0" xmlns="http://www.w3.org/2001/06/grammar"
    xml:lang="en-US" mode="voice"
    root="citystate" tag-format="semantics/1.0">
<rule id="citystate">
...
<one-of>
```

Chapter 4

Example Applications

Introduction

Developers often learn from example, more than from reading specs and even Getting Started tutorials. For this reason, several example applications are provided that demonstrate increasing complexity in XHTML+Voice development.

The sample code in this chapter includes comments with brief explanations for certain tags. The examples begin with three basic applications, and then progress in increasing complexity. For more information, see the specifications in Chapter 6, “References” on page 131.

Note:

Before you can try these applications, you should install a multimodal browser, one that has been enhanced to provide speech capability, such as the multimodal version of the Opera browser or NetFront(R) browser, which are both packaged with the Multimodal Toolkit V4.3 for WebSphere Studio.

This chapter includes the following sections:

- “Three basic examples to get started” on page 94.
- “Example 1” on page 96.
- “Example 2” on page 99.
- “Example 3” on page 103.
- “Example 4” on page 112.

The following statement applies to all examples in this chapter.

Three basic examples to get started

This section includes three basic applications. They demonstrate VoiceXML applications to familiarize you with VoiceXML structure. The first basic example creates the popular “Hello World.”

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice 1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice12.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ev="http://www.w3.org/2001/xml-events"
      xmlns:vxml="http://www.w3.org/2001/vxml" xml:lang="en_US">

  <head>
    <title>Basic XHTML+Voice example for “Hello World”</title>
        <!--*
        ***** The basic element of a VoiceXML document is the form...
        **-->
        <vxml:form id="vxml_form">
            <!--*
            ***** ...which can then have one or several vxml:block
            *****  elements for solely output tasks (or vxml:field
            *****  elements for input/output tasks).
            **-->
            <vxml:block>
                hello world
            </vxml:block>
        </vxml:form>
  </head>
  <body> </body>
</html>
```

In the sample code above, note the following, which will be used in all the examples to follow:

- The DOCTYPE describes the type of document this is, with the valid DTD for XHTML+Voice. It isn't necessary for voice processing, but is necessary for the document to be valid.
- The <html> tag includes the XHTML and XML Events declarations.
- The <head> tag includes the spoken and visual application. In this application, no recognition is included.

- The <vxml:form> tag is the basic element of a VoiceXML document, to which we should assign an element ID.
- The <vxml:block> tag includes the spoken output. In later examples, we can use this tag to perform more complex tasks with VoiceXML.

The second basic example is an application that prompts for a response, recognizes a spoken response, and repeats it back to you. Note that the application will not actually run because it has no HTML. It is an example of VoiceXML, not XHTML+Voice.

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml xmlns="http://www.w3.org/2001/vxml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.w3.org/2001/vxml
   http://www.w3.org/TR/voicexml20/vxml.xsd"
   version="2.0">
<!--*
***** This simple VoiceXML application takes an input
*****  and kindly plays it back to you!
**-->
  <form>
      <field name="drink">
         <prompt>Would you like coffee, tea, milk, or nothing?</prompt>
         <grammar src="drink.jsgf" type="application/x-jsgf"/>
      </field>
      <block>
         Thank you! Your <value expr="drink"/> order will be processed 
shortly!
      </block>
  </form>
</vxml>
```

The following JSGF grammar is called with the application above.

drink.jsgf

```
 #JSGF V1.0;
 grammar ctm;
 public <ctm> = coffee | tea | milk ;
```

The third basic example shows the most basic XHTML+Voice document, "Hello world," and how we use XHTML+Voice to combine VoiceXML content with an HTML document, at the most basic level.

When we open this page, we see just the text. And if a voice-enabled “multimodal” browser is working correctly, we should also hear “Hello world” spoken to the user.

The basic element of a VoiceXML document is the <vxml:form>, which can then have one or several <vxml:block> elements, for solely output tasks, or vxml:field elements, for input/output tasks.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice 1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice12.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ev="http://www.w3.org/2001/xml-events"
 xmlns:vxml="http://www.w3.org/2001/vxml"
 xml:lang="en_US">
     <head>
        <title>Basic XHTML+VXML Example</title>
        <vxml:form id="vxml_form">
            <vxml:block>
                hello world
            </vxml:block>
        </vxml:form>
      </head>
<!--*
***** We use XML events to point the browser to a vxml form. In this
***** case, we're telling it to enter the form just as the page loads
**-->
    <body id="page.body" ev:event="load" ev:handler="#vxml_form">
        You should hear "Hello, world".
    </body>
</html>
```

Example 1

This following example is very similar to to the last one, except that we show how to tie new events to the completion of a VoiceXML form, something like cause-and-effect.

This X+V document accomplishes two things. When the page loads, the text-to-speech engine says "Hello world" to the user, as in the previous example. Once that is completed, the text value "Hello, world!" is assigned to an HTML text box element on the page. Using speech/audio output together with visual output is the essence of "multimodal" X+V!

Note that we use special XML Events like “vxmldone” to signal to the page that the application should do something.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice 1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice12.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:ev="http://www.w3.org/2001/xml-events"
    xmlns:vxml="http://www.w3.org/2001/vxml"
    xml:lang="en_US">

    <head>
        <title>Basic XHTML+VXML Example</title>
       <!--* When declare="declare" presents, the script element is not
       **** executed until the document has completed loading and has
       **** been called through a user event.
       **-->
        <script type="text/javascript" id="vxml_form_handler"
                declare="declare">
                document.getElementById('page.output_box').value =
                   "Hello, world!";
        </script>
<vxml:form id="vxml_form">
            <vxml:block>
                hello world
            </vxml:block>
        </vxml:form>

    <!--*
    *****  We assign a body element as an XML observer,
    *****  which lets us assign script to be executed when our form
    *****  completes.
    **-->
        <ev:listener ev:observer="page.body" ev:event="vxmldone"
            ev:handler="#vxml_form_handler" ev:propagate="stop" />
    </head>

    <!--*
```

```
      ***** We do two things here: We assign our VoiceXML to be loaded when
      *****  the page loads, and we assign a document ID to our HTML body.
      **-->
      <body id="page.body" ev:event="load" ev:handler="#vxml_form">

      <!--*
      ***** Text will show up on the screen in this text box after
      ***** our page is done "speaking" to the user.
      **-->

          Here it comes...<br/>
          <br/>
          <input type="text" id="page.output_box" value="" size="40"/>
          <br/>

      </body>

  </html>
```

Example 2

In this example, we're going to use an "inline" grammar, which is really just placing the contents of an external file into the actual page.

You should probably try to avoid this practice except in limited cases. We're just doing it for demonstration purposes! It is primarily acceptable for very short grammars that are more than likely not reusuable in other applications.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice 1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice12.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ev="http://www.w3.org/2001/xml-events"
 xmlns:vxml="http://www.w3.org/2001/vxml"
 xml:lang="en_US">

    <head>
        <title>XHTML+VXML Example, with Input</title>

        <script type="text/javascript">
            var planet_var = "";
            function FormatPlanet(oldStr)
            {
         /****
         ***** This function just does some text formatting:
         ***** Make sure the first character is upper-case.
         ****/

                newStr = oldStr.charAt(0).toUpperCase()
                    + oldStr.substring(1, oldStr.length);

                return newStr;
            }
        </script>

        <vxml:form id="vxml_form_prompt">
            <vxml:field name="vxml_field">
<vxml:grammar>
                    <![CDATA[
                    #JSGF V1.0;
                    grammar planet_selection;
                    public <planet_selection> =
                        mercury | venus | earth
                        | mars | jupiter | saturn
                        | uranus | neptune | pluto
                        ;
                ]]>
            </vxml:grammar>

            <vxml:prompt>
                Which world would you like to say hello to?
            </vxml:prompt>

<!--*
 *****  What if users doesn't understand what their options
 *****  are? Well then, they can say "help" and hear them!
 *****  Note: This isn't the only event we can catch, but for
 *****  now it'll work just fine.
 **-->
            <vxml:catch event="help">
                Your options are:
                    mercury, venus, earth, mars, jupiter,
                    uranus, neptune, or pluto.
            </vxml:catch>

  <!--*
  ***** Once the field has recognized a grammar entry, we can
  ***** move along to assigning our variables.
  ***** We can change HTML elements on the page from within our
  ***** VXML forms. See? That's because in most browser
  ***** environments, all the variables that VoiceMXL uses are
  ***** also ECMAscript variables, just like the rest of
  ***** the variables on the page.
  **-->
          <vxml:filled>
            <vxml:assign name="planet_var" expr="vxml_field"/>
            <vxml:assign
                   name="document.getElementById('page.output_box')
                 .value"
                expr="'Hello, ' + FormatPlanet(planet_var) + '!'"/>
```

```
              </vxml:filled>
           </vxml:field>

     <!--*
     ***** We can mix and match as many vxml:field or vxml:block
     ***** elements as we want to, and they'll be visited
     ***** in order. We can actually control how they get
     ***** visited, but that's a more advanced topic that
     ***** we'll talk about later!
     **-->

          <vxml:block>
              Hello <vxml:value expr="planet_var"/>, you sure are a
                wonderful planet!
          </vxml:block>
        </vxml:form>

    </head>

    <body id="page.body" ev:event="load"
          ev:handler="#vxml_form_prompt">

        <input type="text" id="page.output_box" value="Hello?"
               size="18"/>
        <br/>

    </body>

</html>
```

The purpose of this guide is not to teach VoiceXML, so we refer interested readers instead to the VoiceXML spec or to the VoiceXML Programmer's Guide.

However, that said, while we used a grammar to describe our list of options in this example, we could have instead used the tag <vxml:option>. That way, instead of having a <vxml:grammar> tag, we would have had something like this:

```
<vxml:option value="mercury"> mercury </vxml:option>
<vxml:option value="venus"> venus </vxml:option>
<vxml:option value="earth"> earth </vxml:option>
```

and so on. Also in this example we use JavaScript/DOM to assign values to HTML elements on the page. We could also use a helpful tag called xv:sync to tie VoiceXML forms and HTML forms together, which we will discuss later. These are just something to keep in mind for future projects.

Example 3

In this example, we will use an external grammar rather than an inline one, which is the recommended way of using most grammar. Also, pay close attention to the content of the grammar. It highlights some interesting things you can do with VoiceXML grammars (voice grammars can be versatile).

Every 10 seconds before the user fills the form, the application plays the "help" dialogue. That is, once the timeout is reached, a "noinput" event is thrown. Although our example throws this every 10 seconds, we could certainly modify the code so that this is only done once, but that is left as an exercise to the reader.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice 1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice12.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ev="http://www.w3.org/2001/xml-events"
 xmlns:vxml="http://www.w3.org/2001/vxml"
 xml:lang="en_US">

    <head>
        <title>XHTML+VXML Example, with Input</title>
        <script type="text/javascript">
            var drink_selection = "";
            var cream = false;
    /****
    ***** A little bit of visual clean-up for when our form is
    ***** complete, just to provide nice-looking visual feedback.
    ****/
            function FormDone()
            {
                var str = "One "
                    + drink_selection.size
                    + " " + drink_selection.type;
                if (cream === true)
                {
                    str += ", with cream.";
                }
                else
                {
                    str += ".";
                }
                 document.getElementById('page.output_box').value = str;
            }
        </script>
     <vxml:form id="vxml_drink_form">
            <vxml:field name="drink_field">

      <!--*
      ***** Here we're using an external grammar.
      **-->
                <vxml:grammar src="gram/beverage.jsgf"/>

                <vxml:prompt timeout="10s">
                    What kind of drink would you like to order?
                </vxml:prompt>

                <vxml:catch event="nomatch noinput help">
                    This form lets you order a drink.
                    You may order a small, medium, or large drink.
                    The drink may be coffee, lemonade, soda, or milk.
                </vxml:catch>

                <vxml:filled>
     <!--*
     ***** Here, we're assigning the current value of
     ***** the variable drink_field to an external Javascript
     ***** variable. This is NOT necessary; you could just as
     ***** well reference drink_field throughout the rest
     ***** of the program, assuming it is filled. However, it
     ***** is often cleaner this way.
     **-->
                    <vxml:assign name="drink_selection"
                                 expr="drink_field"/>
                </vxml:filled>
            </vxml:field>
        </vxml:form>

        <vxml:form id="vxml_coffee_prompt">
            <vxml:field name="coffee_field">
                <vxml:grammar src="gram/yes_no.jsgf"/>
                <vxml:prompt>
                    Would you like cream with your coffee?
                </vxml:prompt>
                <vxml:catch event="nomatch help">
                    If you would like cream with your coffee,
                    then say "yes". Otherwise, say "no".
                </vxml:catch>

<!--*
***** Note that in this case, we assign a boolean result
*****  to our yes/no prompt response. We also make use
*****  of one of VoiceXML's branching constructs.
**-->
                <vxml:filled>
                    <vxml:if cond="true === coffee_field">
                        <vxml:assign name="cream" expr="true"/>
                    </vxml:if>
                </vxml:filled>
            </vxml:field>
        </vxml:form>
<!--*
***** Our handler script is a little more complex this time,
***** and shows how we can navigate to other forms. If the user
***** has selected coffee, then we actually enter another
***** VoiceXML form to ask if they want cream with their coffee.
***** FormDone() is the function that writes visual output to
***** the HTML text box. We call that when we're done, so for
***** all the paths *except* coffee, we call it. Coffee, on the
***** other hand, calls FormDone() in its vxmldone handler only
***** once, and its also gotten its required input.
**-->
 <script type="text/javascript" id="vxml_drink_form_handler"
        declare="declare">
        if ("small" === drink_selection.size)
            document.getElementById('page.size.small').checked
             = true;
        else if ("medium" === drink_selection.size)
            document.getElementById('page.size.medium').checked
             = true;
        else if ("large" === drink_selection.size)
            document.getElementById('page.size.large').checked
```

```
                  = true;
/****
 ***** In this example, coffee is special, so we treat it
 *****  differently!
 ****/
            if ("coffee" === drink_selection.type)
            {
                document.getElementById('page.drink.coffee').checked
              = true;

  /****
  ***** This is how we "branch" to another form, with an XML
  ***** "click" event to trigger the handler mode for an XML
  ***** listener.
  ****/
                document.getElementById('vxml.coffee').click();
            }
            else
            {
                if ("soda" === drink_selection.type)
                  document.getElementById('page.drink.soda')
                    .checked = true;
                else if ("lemonade" === drink_selection.type)
                    document.getElementById('page.drink.lemonade')
                    .checked = true;
                else if ("milk" === drink_selection.type)
                    document.getElementById('page.drink.milk')
                    .checked = true;

                FormDone();
            }
     </script>

     <script type="text/javascript" id="vxml_coffee_form_handler"
             declare = "declare">

         /****
         ***** In all the other "non-coffee" paths, we called this
         *****  earlier. So now we have to make sure it gets called
         *****  when coffee is done!
         ****/
```

```
                FormDone();

        </script>
    <!--*
    ***** If you compare this listener to the one below it, you'll
    ***** notice that it doesn't have any HTML element to watch
    ***** besides the body element, which only throws an event on
    ***** loading the page. This is a problem!
    *****
    ***** Why? Because we don't really have a way to trigger entry
    ***** into this form again, after the page is loaded! This works
    ***** fine for our example, but in the future, you may wish
    ***** to take a different approach. You could change the body
    ***** ev:event to "load click", as well as on loading, thus
    ***** triggering the voice form by "click"ing the body.
    ***** However, a better solution is probably to add a hidden
    ***** input element like the one we we have for the coffee
    ***** form, and then make our load-event handler "click" this
    ***** new hidden input element when the page. Or better yet, you can
    ***** use DOM Level 2 Event, DOMActivate event, to activate the
    ***** voice form from your JavaScript routine.
    **-->
        <ev:listener ev:observer="page.body" ev:event="vxmldone"
          ev:handler="#vxml_drink_form_handler" ev:propagate="stop" />
   <!--*
   ***** To watch for XML events (i.e. vxmldone events) for most forms,
   ***** we create a hidden HTML element so that our event listener
   ***** has something to watch, and our VoiceXML form sends its
   ***** resultant events to the page through an HTML element.
   *****
   ***** In a more simple form, this would probably work a little
   ***** differently. Instead of using a hidden element, as we do here,
   ***** we could instead use a text element so that when the users
   ***** click on the field to begin typing into it, they would
   ***** activate the voice form that goes with it. However, since
   ***** this is our "interface" to the HTML document, it often
   ***** happens to be convenient (as it is in this case) to make it
   ***** invisible to the user.
   **-->
   <ev:listener ev:observer="vxml.coffee" ev:event="vxmldone"
          ev:handler="#vxml_coffee_form_handler" ev:propagate="stop" />
    </head>

    <body id="page.body" ev:event="load" ev:handler="#vxml_drink_form">

        <input type="hidden" id="vxml.coffee" value="" ev:event="click"
            ev:handler="#vxml_coffee_prompt"/>

        <b>Multimodal Drink Order</b><br/>
        <br/>
        Size options:<br/>
            <input type="radio" name="size" id="page.size.small"/>
                        Small <br/>
            <input type="radio" name="size" id="page.size.medium"/>
                        Medium <br/>
            <input type="radio" name="size" id="page.size.large"/>
                        Large <br/>
        <br/><br/>
  Drink options:<br/>
            <input type="radio" name="type" id="page.drink.soda"/>
                        Soda <br/>
            <input type="radio" name="type" id="page.drink.lemonade"/>
                        Lemonade <br/>
            <input type="radio" name="type" id="page.drink.coffee"/>
                        Coffee <br/>
            <input type="radio" name="type" id="page.drink.milk"/>
                        Milk <br/>
        <br/><br/>

        <input type="text" id="page.output_box"
              value="Waiting for selection." size="40"/>
        <br/>

    </body>

</html>
```

Yes/no JSGF grammar

This grammar includes typical responses for specifying yes or no.

```
#JSGF V1.0 iso-8859-1;

grammar yes_no;

//
// This is a good example of trying to give users as
//  many options as possible for conveying their
//  meaning, while keeping the program constructs
//  as constrained as possible for the programmer (in
//  this case, we only consider a boolean result).
//
// It saves the programmer from having to parse the
//  the utterance string.
//

public <yes_no> =
    <yes> { $ = true; }
    | <no> { $ = false; };

<yes> = yes [please] | sure | okay | fine | yep | yup | affirmative;
<no> = no | nope | no thanks | negative;
```

Beverage JSGF grammar

This grammar provides valid utterances, such as "Give me a regular coffee." Note that we make use of "optional" phrases in this grammar, in order to make the grammar more natural for the average person. However, as an aside, keep in mind that when you start to make your grammars more lenient, people may start to develop higher expectations of what is valid, and get frustrated when your carefully planned grammar does not recognize what they try to respond.

This example also uses "semantic interpretation." This means that we explicitly assign a value to the field/rule, rather than always assigning the utterance (which is the default behavior). This lets us have multiple words signifying the same result. Also, it is usually not a desirable thing to include the optional phrases like "I would like" in the final result, as far as the programmer is concerned.

The code "$.size = $size" might seem confusing. In this context, "$" refers to the field itself, whereas $rule refers to the last rule that was recognized with that name. So what we are doing here is creating a variable called "size" that is a member of the field variable (as we would do with a C structure, for example), and assigning it to the value assigned by the most recent rule (i.e. <size>). In essence, it is saying "$.variable = $lastrule", which is confusing only because in this case the rule name is the same as the variable name.

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

public <beverage> = [I would like | I want | [please] give me]
    [a | an]
    <size> { $.size = $size; }
    <type> { $.type = $type; }
    ;

//
// Semantic interpretation lets us assign "medium" to
//  the utterance "regular", giving us more flexibility.
//
<size> = small { $ = "small"; }
    | (medium | regular) { $ = "medium"; }
    | large { $ = "large"; }
    ;

//
// The default assignment is fine for most of these!
```

```
//
<type> = coffee
    | milk
    | (soda|pop|coke) { $ = "soda"; }
    | lemonade ;
```

Example 4

This final example illustrates how we can use mixed-initiative techniques to control the flow through a VXML form. It also shows a few different types of HTML input types that we can control with X+V, and how we do it.

First of all, we need a short explanation of what we mean by "mixed-initiative". In all XHTML+Voice application, we use the Form Interpretation Algorithm (FIA) to make sure that we visit all the fields and blocks in the form in a certain order. Visually, this order is start to finish.

Basically, when the VoiceXML interpreter goes through a form, it visits only those fields that have the value "undefined." Once those fields are filled with some input, then they are no longer “undefined”, and so they will not be visited again. What we can do is manually set the value of the fields before they would normally be visited, if we already have enough information about that particular input that we do not need to use that field anymore.

In this case, we use a form-level grammar (a grammar that is embedded into the whole form, independent of field/block) to specify a grammar that includes all the grammar entries for each of the following fields. By carefully constructing this "exhaustive" grammar, we can let users say all in one utterance the various input details for our order form, and for each type of input (such as the type of bread), then we set the field for that input to some appropriate value besides “undefined,” so that it won't get visited again. Users do not have to specify all the options; they can say only a partial list of options, and the form will still go through and visit all the remaining fields (those that are still undefined), in effect prompting them for all the information they still might wish to include.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE html PUBLIC "-//VoiceXML Forum//DTD XHTML+Voice 1.2//EN"
"http://www.voicexml.org/specs/multimodal/x+v/12/dtd/xhtml+voice12.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
 xmlns:ev="http://www.w3.org/2001/xml-events"
 xmlns:vxml="http://www.w3.org/2001/vxml"
 xml:lang="en_US">

<head>
   <title>Multimodal Sandwich Order Form</title>

   <script type="text/javascript">
   /****
   ***** The functions in this script block are just some text-
```

```
***** formatting helper functions, so that we can provide
***** nice-looking visual feedback to the user.
****/
         function countToppings()
         {
             var count = 0;

             if (document.getElementById('page.toppings.tomato')
                .checked) count++;
             if (document.getElementById('page.toppings.lettuce')
                .checked) count++;
             if (document.getElementById('page.toppings.onion')
                .checked) count++;

             return count;
         }

         function displayOrder()
         {
             alert(getOrderString());
         }

                   function getOrderString()
                   {
                      var order = "Your order is: A sandwich ";

             var total = countToppings();
             var count = 0;

             if (total > 0)
             {
                 order += "with ";

                 if (document.getElementById('page.toppings.tomato')
                    .checked)
                 {
                     count++;
                     order += "tomatoes";
                     order += getComma(count, total);
                 }
                 if (document.getElementById('page.toppings.lettuce')
           .checked)
        {
            count++;
            order += "lettuce";
            order += getComma(count, total);
        }
        if (document.getElementById('page.toppings.onion')
           .checked)
        {
            count++;
            order += "onions";
            order += getComma(count, total);
        }
    }
    order += "on ";

    if (document.getElementById('page.toasted')
       .checked)
    order += "toasted ";

    if (document.getElementById('page.bread.white')
       .checked)
    order += "white ";
    else if (document.getElementById('page.bread.wheat')
            .checked)
          order += "wheat ";
    else if (document.getElementById('page.bread.spicey')
            .checked)
          order += "spicey ";

             order += "bread.";

             return order;
          }
function getComma(count, total)
{
    if (count === total)
        return " ";
    else if (total-count === 1)
        return " and ";
    else
```

```
                    return ", ";
            }
        </script>

        <vxml:form id="sandwich_order_form">

   <!--*
   ***** We use this variable to control whether
   *****  or not the user wants to completely ignore
   *****  the "exhaustive" form-level grammar, and
   *****  go directly to the specific fields. They
   *****  might want to do this if they are in a
   *****  hands-free environment, in which it is
   *****  convenient to address each prompt
   *****  individually. This way, they can use the
   *****  "help" prompt to get a list of options
   *****  for each menu segment, whereas getting a
   *****  list of all of the menu's options at once
   *****  would be prohibitively long.
   **-->
         <vxml:var name="visitInitial" expr="true"/>

            <vxml:grammar>
   <!--*
   ***** Note that we've tried to give this grammar
   ***** a little extra flexibility in what the user can
   ***** say. We have used the "*" operator to let them
   ***** list as many toppings as they want, or none at
   ***** all. This particular implementation ends up
   ***** allowing some pretty strange phrases, but by
   ***** allowing strange phrases (that will almost never
   ***** be used except by people trying to figure out
   ***** how the grammar works), we make sure that we
   ***** catch more valid ones than we otherwise would.
   **-->
                 <![CDATA[
                        #JSGF V1.0;
                        grammar sandwich_order;
        public <sandwich_order> =
          [
             [<list_menu> {visitInitial = false;} ]
            [I would like | I'd like] [[to] (order|get)] [[please] give me]
            [a sandwich]
         ]
         [
            [ <toppings> {$.voice_field_toppings += $toppings;} ]
            [ [on|with] [a] [ toasted { $.voice_field_toasted = true }]
            [<bread> {$.voice_field_bread = $bread;}] [bread | bun] ]
          ]*;
      <topping> = none
            | (tomato|tomatoes) { $ = "tomato"; }
            | lettuce
            | (onion|onions) { $ = "onion"; }
            ;
      <toppings> = ( [and|with] [a|an|some] <topping> )*;
      <bread> = white | wheat | spicey ;

      <list_menu> = list [ [my|the] [menu] [options|choices] ];
   ]]>
         </vxml:grammar>

         <vxml:block>
               Welcome to the sandwich order form.
         </vxml:block>

  <!--*
  ***** vxml:initial basically specifies the form-level "prompt" for
  ***** our form-level grammar. We use this to handle everything that
  ***** should occur outside of any specific field or block.
  ***** This will NOT be visited more than once if users suggest that
  ***** they want to list the the menu! Take a look at our grammar entry
  ***** for this case to see why. If we didn't have this, then users
  ***** would have to fill in at least one field through the form-level
  ***** grammar before he could proceed into the individual fields.
  **-->
       <vxml:initial cond="visitInitial == true">
                    <vxml:prompt timeout="10s" count="1">
             Please select what you would like on your sandwich
                    </vxml:prompt>
                    <vxml:catch event="nomatch noinput help">
             You may order tomatoes, lettuce, or onions on your sandwich.
             Your bread may be white, wheat, or spicey.
```

```
              You may choose to have your bread toasted.
                          </vxml:catch>
        <vxml:catch event="help" count="2">
             You may order tomatoes, lettuce, or onions on your sandwich.
             Your bread may be white, wheat, or spicey.
             You may choose to have your bread toasted.
             To select each field individually, say the word "list"
        </vxml:catch>
        </vxml:initial>
   <!--*
   ***** Here is our first actual field. There a few
   ***** things to notice here. First, we have 'modal=
   ***** "true"', which tells the interpreter that it
   ***** should NOT also accept entries from the form-
   ***** level grammar along with this field-grammar.
   ***** Otherwise, we could actually mix the form-level
   ***** and field-level grammars together, which can
   ***** often be a very powerful technique.
   *****
   ***** Second, notice that our field-level grammars are
   ***** often just subsets of our "exhaustive" form-
   ***** level grammar. If you're not clear why this is,
   ***** then spend some time looking over the rest of
   ***** this example until it makes better sense.
   **-->
        <vxml:field name="voice_field_toppings" modal="true">
            <vxml:grammar>
                    <![CDATA[
                           #JSGF V1.0;
                           grammar topping_options;
            public <topping_options> = [I would like | I'd like] [no]
                   <toppings> [toppings]
                              { $ += $toppings } ;
                          <topping> =
                              (tomato|tomatoes) { $ = "tomato"; }
                              | lettuce
                              | (onion|onions) { $ = "onion"; }
                              ;
                        <toppings> = ( [and|with] [a|an|some] <topping> )*;
                              ]]>
                           </vxml:grammar>
                    <vxml:prompt>
                            What toppings would you like?
                            You may select tomato, onion, or
                            lettuce.
                    </vxml:prompt>
                    <vxml:catch event="help nomatch noinput">
                           Topping options are: tomato, onion,
                           lettuce.
                    </vxml:catch>
                    <vxml:filled>
<!--*
***** By searching for values within the field's result
*****  string, we allow ourselves to accept an arbitrary
*****  number of values for any particular field input.
*****  This is often helpful for scenarios when we want
*****  to accept a list of input.
**-->
     <vxml:if cond="voice_field_toppings.search(/tomato/i) != -1">
         <vxml:assign name="document.getElementById
                   ('page.toppings.tomato').checked" expr="true"/>
     </vxml:if>
     <vxml:if cond="voice_field_toppings.search(/lettuce/i) != -1">
         <vxml:assign name="document.getElementById
                   ('page.toppings.lettuce').checked" expr="true"/>
     </vxml:if>
     <vxml:if cond="voice_field_toppings.search(/onion/i) != -1">
         <vxml:assign name="document.getElementById
                   ('page.toppings.onion').checked" expr="true"/>
     </vxml:if>
            </vxml:filled>
         </vxml:field>

<!--*
***** Here we use the same boolean JSGF "yes/no" grammar that
*****  was used in earlier examples.
**-->
         <vxml:field name="voice_field_toasted" modal="true">
            <vxml:grammar src="gram/yes_no.jsgf"/>
            <vxml:prompt>
               Would you like your bread toasted?
            </vxml:prompt>
            <vxml:catch event="help nomatch noinput">
                  If you would like your bread toasted, say "yes".
                  Otherwise say "no."
            </vxml:catch>

            <vxml:filled>
                  <vxml:if cond="voice_field_toasted === true">
                  <vxml:assign name="document.getElementById
                              ('page.toasted').checked" expr="true"/>
                  </vxml:if>
            </vxml:filled>
      </vxml:field>

            <!--*
            ***** This is similar to the topping field,
            *****  except that breads are mutually exclusive;
            *****  we should only select one of these, not
            *****  several!
            **-->
      <vxml:field name="voice_field_bread" modal="true">
         <vxml:grammar>
            <![CDATA[
               #JSGF V1.0;
                  grammar bread_options;
                      public <bread_options> = [I would like | I'd like]
                                 <bread> { $ = $bread } [bread] ;
                             <bread> = white | wheat | spicey ;
               ]]>
         </vxml:grammar>
            <vxml:prompt>
               What kind of bread would you like?
               You may choose white, wheat, or spicey bread.
            </vxml:prompt>
            <vxml:catch event="help nomatch noinput">
               You may order white, wheat, or spicey bread.
            </vxml:catch>

            <vxml:filled>
         <vxml:if cond="voice_field_bread.search(/white/i) != -1">
           <vxml:assign name="document.getElementById
                        ('page.bread.white').checked" expr="true"/>
            <vxml:elseif cond="voice_field_bread.search(/wheat/i) != -1"/>
              <vxml:assign name="document.getElementById
                            ('page.bread.wheat').checked" expr="true"/>
            <vxml:elseif cond="voice_field_bread.search(/spicey/i) != -1"/>
              <vxml:assign name="document.getElementById
                            ('page.bread.spicey').checked" expr="true"/>
            </vxml:if>
               </vxml:filled>
            </vxml:field>

         <vxml:block>
            <vxml:value expr="getOrderString()"/>.
            Thank you for your order.
         </vxml:block>

         </vxml:form>

      </head>
      <body ev:event="load" ev:handler="#sandwich_order_form">

   <!--*
   ***** We don't actually want to submit the form,
   *****  we just want to pop up the results for the user
   *****  to read. Note that we use the same function
   *****  to get the text output that we use to read the
   *****  result back to the user.
   **-->
         <form onsubmit="displayOrder(); return false;" action="">
            <b>Multimodal Sandwich Order Form</b><br/>
         <br/>

         <b>Toppings:</b><br/>
      &nbsp;<input type="checkbox"
            id="page.toppings.tomato"/>&nbsp;Tomato<br/>
      &nbsp;<input type="checkbox"
            id="page.toppings.lettuce"/>&nbsp;Lettuce<br/>
      &nbsp;<input type="checkbox"
            id="page.toppings.onion"/>&nbsp;Onion<br/>
         <br/>

         <b>Toasted?</b>&nbsp;&nbsp;
      <input type="checkbox" id="page.toasted"/><br/>
      <br/>

       <b>Bread:</b><br/>
       &nbsp;<input type="radio" name="bread"
             id="page.bread.white" checked="checked"/>White<br/>
       &nbsp;<input type="radio" name="bread"
             id="page.bread.wheat"/>Wheat<br/>
       &nbsp;<input type="radio" name="bread"
             id="page.bread.spicey"/>Spicey<br/>
       <br/><br/>

         <input type="submit" name="submit"
             id="submitButton" value="Complete Order" />
      </form>
   </body>
</html>
<!--*
***** Note that for the sake of keeping this sample reasonably short,
***** we leave out any sort of "order verification". Normally we would
***** ask the user whether or not the order they've entered is correct,
***** and if not, then we would let them change the part of the order
***** that is incorrect (basically, by resetting that field's value so
***** that the form will revisit it).
*****
***** To see how we might do this, take a look at the IBM Pizza Order
***** Form demo, which is an expanded version of this example.
**-->
```

Yes/no JSGF grammar

```
#JSGF V1.0 iso-8859-1;

grammar yes_no;

//
// This is a good example of trying to give the user
//  many options as possible for conveying his or her
//  meaning, while keeping the program constructs
//  as constrained as possible for the programmer (in
//  this case, we only consider a boolean result).
//
// It saves the programmer from having to parse the
//  the utterance string.
//

public <yes_no> =
    <yes> { $ = true }
    | <no> { $ = false; };

<yes> = yes [please] | sure | okay | fine | yep | yup | affirmative;
<no> = no | nope | no thanks | negative;
```

Chapter 5 Multimodal Browser

After you install the Multimodal Browser, the icon for the installed browser, such as the Opera browser, appears on your desktop. You can use it to open the browser and run your multimodal applications.

This chapter includes the following sections:

- "What is a Multimodal Browser?" on page 123.
- "Running the Multimodal Browser" on page 124.
- "Troubleshooting tips" on page 129.

What is a Multimodal Browser?

The Multimodal Browser provides a Web browser in which you can test voice-enabled Web applications. The Multimodal Browser is based on familiar browser technology that is enhanced with extensions that include the IBM automatic speech recognition and text-to-speech technology. This allows you to view and interact with multimodal applications that you have built using XHTML+Voice.

In addition to running and testing your multimodal applications with voice, the browser includes a command and control vocabulary so that you can navigate your browser using voice commands.

Browser features available in the Multimodal Toolkit

- When you develop an application in the toolkit, you can open the application in the browser using the right-click option.
- If you make changes in the .mxml file in the X+V editor, save the changes, and then Reload the application in the browser and test your changes immediately.

In the toolkit, you can set a **Run configuration** that will launch the application in the specified multimodal browser using the **Run** menu:

- Using the **Run** menu or the **Run** toolbar icon, select **Run...** to open the Launch Configurations dialog.
- By default, the Multimodal Browser window opens on the Main page.
- By default, the open X+V file name appears, or you can use the **Browse** button to locate the .mxml file.
- Select the preferred browser from the drop-down list, and click **Apply**.

When you click the **Run** button on the dialog, the file opens in the specified browser. You can launch the application anytime by selecting **Run > Run History**, selecting the configuration name.

Running the Multimodal Browser

To test your multimodal application, you will need a Microsoft Windows 2000 compatible, 16-bit, full-duplex sound card (with a microphone input jack) with good recording quality and a high-quality microphone.

Use one of the following methods to open the Multimodal Browser:

- Double click the desktop icon for the Multimodal Browser, such as the Opera browser.
- Using the **Start** menu, select **Programs** and select the installed browser, such as **Opera**.

Then use the **File > Open** menu to locate the <filename>.mxml or .html file (change the "Files of type" field to show **All files**).

When you give focus to a voice-enabled field (click in the field with the cursor), you will hear the voice prompt for the field.

1. Open a voice-enabled file in the browser. In some applications, a voice prompt begins immediately. In other applications, you should click in each field to hear the voice prompt.
2. Press the **Scroll Lock** key as the Push-to-Talk (or microphone) button. Listen for the tone, and then pause a second before speaking to let the speech recognition engine engage.
3. Speak into the microphone to respond (continue to press the key for a second so that the recognition engine captures all of your response).
4. Release the key, and your response should appear as text in the field.

5. If you make changes to any of the application files, such as the grammar or pool files, you should close and re-open the browser to make sure that the new files are loaded.

Using the Opera browser

At installation, if you selected to install the Opera browser, the desktop icon was added to your desktop. You can use this icon to open and run the Multimodal Browser based on Opera technology. For full documentation on using the Opera browser, refer to the online help included with the browser. Only the Voice preferences added to the browser are described in this document.

Setting Voice preferences

The Opera browser includes a Voice preferences page in which you can change the listening mode, keyboard Push-to-Talk button, and log level. To do this, in the browser, select **Tools > Preferences > Voice**, and use the following settings:

- The **Enable voice** check box is selected by default. Deselect it to disable the voice features with the browser.
- The **Voice setup** area and buttons are reserved for future use.
- The **Stop computer speech if I click mouse button** check box is selected by default. When checked, you can stop voice prompts by clicking the mouse on the screen (anywhere except in a voice-enabled field). Deselect the check box to disable the canceling feature.
- The **Key to talk drop-down** list includes the following options for the keyboard key that will activate the system microphone for input:
    - Scroll Lock (default selection)
    - Insert
- The Talk Key mode drop-down box includes the following options for activating the "listening" function on the browser:
    - In **Hold key while talking** mode, press and hold the button on the device while speaking, and then release the button (default selection).
    - In **Press key, then talk** mode, press and release the button, and then talk. When you finish speaking, it detects silence and automatically stops listening (if there is background noise, it might take a moment for the system to detect the end of speech).

  Note: When using the VoiceXML <record> tag, the Push-to-activate mode has a slightly different behavior. You press and release the button, say the response and then push and release the button to signal the end of the response.

  - In **Key not required to talk** mode, the browser automatically sounds a tone when it is ready to record your response. When you finish speaking, the device detects silence and automatically stops listening (if there is background noise, it might take a moment for the device to detect the end of speech).
    Note: In this mode, the system will not throw a VoiceXML <noinput> event.

- The **Voice log level** drop-down box includes the following preferences for logging:
  - Log disabled (default selection)
  - Verbose
  - Info
  - Warning
  - Severe
- Check **Control Opera user interface using voice** to enable the command, control, and content vocabulary (deselected by default). If you enable it, you can use voice commands to activate controls in the browser, instead of the grammars in the X+V applications. The voice commands must be preceded by the Browser Name ("Browser," by default). For example, to see a list of voice commands, with the browser running and this option enabled, you can press the **Scroll Lock** key and say "**Browser, show voice commands**."

  Voice commands include: **Back, forward, home, refresh, page up, page down, zoom in, zoom out, normal size, show bookmarks, show help,** and **show voice commands** (or **show commands**).
- In the **Browser Name** field, type the command name (browser, by default) that will activate the global command and control vocabulary, instead of the grammars in the X+V applications. Refer to the **Control Opera user interface using voice** option, above.

Other tips:

- If you find that the **Opera** browser has become your default browser, you can reset your preferred browser as the default and continue to use the Opera browser to test your multimodal projects. For example, to reset Microsoft Internet Explorer, from the IE toolbar, select **Tools > Internet Options > Programs**, and click the **Reset Web Settings** button.
- You can control the Memory and Disk caching. To enable or disable caching in the browser, select **Tools > Preferences**, and select **History and cache**. For example, next to **Disk cache**, select the

Empty now button. Note that if you change your application files and they have been cached in the browser, the old files will continue to be used until you clear the cache.

Using the NetFront browser

At installation, if you selected to install the NetFront browser by ACCESS Systems, the desktop icon was added to your desktop. You can use this icon to open and run the Multimodal browser based on ACCESS Systems technology.

For full documentation on using the NetFront browser by ACCESS Systems, refer to the browser's online help included with the browser (**Help > Help Topics**). Only the Voice preferences added to the browser are described in this document.

Setting Voice preferences

To view the Voice preferences, in the browser, select **File > Preferences**, and select **Voice**. If you make changes in the preferences, you should restart the browser to activate the changes.

- The **Enable Voice** check box is selected by default. Deselect it to disable the voice features in the browser.
- The **Listening Mode** drop-down box includes the following options for activating the "listening" function on the browser:
    - In **Push-to-talk** mode, press and hold the button on the device while speaking, and then release the button (default selection).
    - In **Push-to-activate** mode, press and release the button, and then begin speaking. When you finish speaking, it detects silence and automatically stops listening (if there is background noise, it might take a moment for the system to detect the end of speech). **Note**: When using the VoiceXML <record> tag, the Push-to-activate mode has a slightly different behavior. You press and release the button, say the response and then push and release the button to signal the end of the response.
    - In **Auto push to activate** mode, the browser automatically sounds a tone when it is ready to record your response. When you finish speaking, the device detects silence and automatically stops listening (if there is background noise, it might take a moment for the device to detect the end of speech).

      Note: In this mode, the system will not throw a VoiceXML <noinput> event.

- The **PTT Key** drop-down list includes the following options for the keyboard key that will activate the system microphone for input, referred to as the **Push-to-Talk** button:
    - Scroll Lock (default selection)
    - Insert
    - Shift
    - Control
    - F8
    - F12
- The **Voice Log Level** drop-down box includes the following preferences for logging:
    - Log disabled (default selection)
    - Verbose
    - Info
    - Warning
    - Severe
- The **Mouse key cancels voice** check box is selected by default. When checked, you can click the mouse on the screen (anywhere except in a voice-enabled field) to stop voice prompts. Deselect the check box to disable the canceling feature.
- Check **Enable C3N** to enable the **command, control, and content** vocabulary (selected by default). If you enable it, you can use voice commands to activate controls in the browser, instead of the grammars in the X+V applications. The voice commands must be preceded by the Browser Name ("Browser," by default) that you specify in the option below. For example, to see a list of voice commands, with the browser running and this option enabled, you can press the **Scroll Lock** key and say "**Browser, show voice commands**."

  Voice commands include: **Back, forward, home, refresh, page up, page down, zoom in, zoom out, normal size, show bookmarks, show help,** and **show voice commands** (or **show commands**).
- In the **Browser Name** field, type the command name (browser, by default) that will activate the global command and control vocabulary, instead of the grammars in the X+V applications. Refer to the **Enable C3N** option, above.

Other browser preferences

- If you find that the NetFront browser by ACCESS Systems has become your default browser, you can reset your preferred browser as the default and continue to use the NetFront browser to test your multimodal projects. For example, to reset Microsoft Internet Explorer, from the IE toolbar, select **Tools > Internet Options > Programs**, and click the **Reset Web Settings** button.

- You can control the memory and disk caching. To enable or disable caching in the browser, select **File > Preferences**, and select **History and Cache**.

Troubleshooting tips

If you do not hear the voice prompt for the voice-enabled field, try the following testing tips:

- Make sure the system volume is not muted or turned too low.
- If you are using a headset, make sure the plugs are inserted into the correct connections.
- Check to see if multiple voice-enabled pages are open in the browser (open pages appear as blue tabs over the workspace). If so, close the other open pages (right-click on a tab, and select **Close all but active**), and reload the multimodal page.
- Check to see if any other programs are running that use the audio card. If so, close the program and re-start the browser.

If you hear the prompt, but your response is not recognized (the text does not appear in the field), try the following testing tips:

- When responding to a prompt, listen for the tone, and wait another second to let the speech engine engage before speaking.
- After you say your response, continue to press the **Scroll Lock** key (Push-to-Talk button) for another second before releasing it. If you release the button too fast, the response might not be recognized.
- When you click in a field and press the Push-to-Talk button, if it takes a long time to hear the beep, it might mean that your grammar is too complex. Try simplifying the grammar and reducing the number of words.
- Check to make sure that the word you use are included in the grammar for the field.
- Try changing the Push-to-Talk button to another keyboard key, such as the **Insert** key. Compare the results and select the best option.

Other tips for using the browser:

- Widen the browser window to view more toolbar icons.
- If the browser fails to launch, open the **Task Manager** (Ctrl+Alt+Del) and check to see if the process opera.exe is running. If so, end it, and then restart the browser. Also, if the toolkit is closed and you see a javaw.exe program still running, end the process, and restart the browser.
- If you minimize the browser, and then open a second session, the second session starts in minimized view.
- To view keyboard shortcuts in the browser, select **Help > Keyboard**, and for mouse shortcuts, select **Help > Mouse**.
- Although the Multimodal Toolkit supports only 11 kHz, 16-bit mono WAV audio files, the Multimodal Browser supports the following audio files:
    - 11 kHz, 16-bit mono and stereo WAV
    - 22 kHz, 16-bit mono and stereo WAV
    - 44 kHz, 16-bit mono and stereo WAV

Limitations of the Multimodal Browser:

- On the browser, the "load" event occurs only when the actual document is loaded, not if you use the Back or Forward button on the browser. In order to receive this event, you must click the Reload button on the browser.
- If you try to record prompts using the Sound Recorder or other audio recorder while the browser is running, an error.noresource event is thrown because audio input/output resource is not available.

Chapter 6 References

This section contains useful Internet references for information related to multimodal applications.

Note:
Visit the **IBM Multimodal Web site** for Frequently Asked Questions (FAQs), white papers, and other product information:
http://www.ibm.com/software/pervasive/multimodal/

This release of the Multimodal Tools is based on the following versions of specifications (for exceptions to these specifications, see the Compatibility with specifications section):

- **XHTML 1.0 - specification** (using the XHTML 1.0 - Transitional DTD):
  http://www.w3.org/TR/xhtml1/
- **XHTML+Voice 1.2 Specification**:
  http://www.voicexml.org/specs/multimodal/x+v/12/spec.html
- **VoiceXML 2.0 specification**:
  http://www.w3.org/TR/voicexml20/
- **Java Speech Grammar Format specification**:
  http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/
- **Semantic Interpretation for Speech Recognition (SISR) specification**:
  http://www.w3.org/TR/semantic-interpretation/
- **XML Events specification**:
  http://www.w3.org/TR/xml-events/
- **Document Object Model (DOM) – Level 2 specification)**:
  http://www.w3.org/DOM/#what

Other related specifications and Web sites:

- **W3C Web site**, for information on many related topics:
  http://www.w3.org

- **Online tutorials** in many related skills:
  http://www.w3schools.com/
- **HTTP 1.1 Specification**:
  http://www.ietf.org/rfc/rfc2616.txt
- **HTTP State Management Mechanism (Cookie Specification)**:
  http://www.w3.org/Protocols/rfc2109/rfc2109
- **ECMA Standard 262: ECMAScript Language Specification**, 3rd Edition, published by ECMA:
  http://www.ecma-international.org/publications/standards/Ecma-262.htm
- **The International Phonetic Alphabet (IPA)**, published by the International Phonetic Association:
  http://www2.arts.gla.ac.uk/IPA/ipachart.html
- **The Unicode Standard Version 3.0**, The Unicode Consortium, Addison-Wesley Publishing Company, 2000.
- **Other downloadable documents** (in .pdf format) are available on the IBM Publications Center Web site:
  http://www.elink.ibmlink.ibm.com/public/applications/publications/cgibin/pbi.cgi

  To use the Web site, select your country, and Search for keywords such as VoiceXML, Voice Server, or Voice Response (DirectTalk®) to find documents related to your specific connection environment.

Appendix A Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
USA

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

Copyright License

Trademarks