“Know how to listen, and you will profit even from those who talk badly.”
- Plutarch

Speech Technologies on the Way to a Natural User Interface

By Jeff Zhuk

This chapter is about speech technologies and related APIs: VoiceXML, SALT, Java Speech API, MS Speech SDK. It looks into unified scenarios with audio/video interface (AVI) definitions, considers design and code examples, and introduces important skills for a new world of wired and wireless speech applications.

What Is a Natural User Interface?

Is a natural user interface (NUI) another set of tags and rules covered by a nice name? Absolutely not!

This time, end users – not a standards committee – make the determination on what they prefer for their methods of interaction. An NUI allows end users to give their preferences at the time of the service request, and to change them flexibly.

Are you a “computer” person? My guess is that you are, because you are reading this book. “Computer literate” folks like you and me enjoy exploring the capacities of computer programs via traditional interfaces. Even so, there are times, such as when we are on vacation, on the go, and in the car, when we prefer “hands-free” conversation rather than using keyboards to access computerized services.

One person prefers handwriting, and someone else is comfortable with typing. One would like to forget keywords and use commonsense terminology instead. Can a computer understand that “find” is the same as “search,” and “Bob” is actually “Robert”? Can it understand that someone has chosen a foreign (non-English) language to interact with it?

An NUI offers all these possibilities. Some of these complex tasks can be addressed in a unified way with AVI scenarios.

First, let us look into the Java details of a voice interface, one part of an NUI. A significant part of the population considers a voice interface to be the most natural and preferred way of interaction.

Speaking with Style

There are open source solutions for text-to-speech conversion. For example, I used the FreeTTS Java Speech application program interface (JSAPI) implementation to write simple Java code for a voice component, but the sound of the voice may not have been terribly impressive. What can the JSAPI [1] offer to enhance a voice component?

Here are two lines of the source (where I use the FreeTTS) that actually speak for themselves:

```
Voice talker = new CMUDiphoneVoice();

talker.speak(speech); // speech is a text string
```

Remember that the *Voice* class is the main talker.

We can create an object of the *Voice* class with four features: voice name, gender, age, and speaking style. The voice name and speaking style are both String objects, and the synthesizer determines the allowed contents of those strings.

The gender of a voice can be GENDER_FEMALE, GENDER_MALE, GENDER_NEUTRAL, or GENDER_DONT_CARE. Gender neutral means some robotic or artificial voices. We can use the "don't care" value if the feature is not important and we are OK with any available voice.

The age can be AGE_CHILD (up to 12 years), AGE_TEENAGER (13 to 19 years), AGE_YOUNGER_ADULT (20 to 39 years), AGE_MIDDLE_ADULT (40 to 60 years), AGE_OLDER_ADULT (60+ years), AGE_NEUTRAL, and AGE_DONT_CARE.

Here is the example of a woman's voice selection with a "greeting" voice style:

```
Voice("Julie", GENDER_FEMALE, AGE_YOUNGER_ADULT, "greeting");
```

Not all the features of JSAPI had been implemented at the time of this book's publication. Here is the reality check: the *match()* method of the *Voice* class can test whether an engine-implementation has suitable properties for the voice (Fig. 12.1).

[Fig.12-1]

The *getVoices()* method creates a set of voices according to initial age and gender parameters. If the requested age or gender is not available, the default voice parameters are set.

We can use this method in the scenario of multiple actors that have different genders and ages.

```
<actors name="age" value=" AGE_TEENAGER | AGE_MIDDLE_ADULT" />
<actors name="gender" value="GENDER_FEMALE | GENDER_MALE" />
```

These two scenario lines define arrays of age and gender arguments that produce four actor voices.

Java Speech API Markup Language

An even more intimate control of voice characteristics can be achieved by using the Java Speech API Markup Language (JSML) [2]. JSML is a subset of XML that allows applications to annotate text that is to be spoken, with additional information. We can set prosody rate or speed of speech. For example:

```
Your <emphasis>United Airlines</emphasis> flight is scheduled tonight
<prosody rate="-20%">at 8:80pm</prosody> It is almost 4pm now. Good
time to get ready.
```

This friendly reminder emphasizes the airline's name and slows down the voice speed 20% while pronouncing the departure time.

According to the JSAPI, the synthesizer's *speak()* method understands JSML. JSML has more element names or tags in addition to *emphasis* and *prosody*. For example, the *div* marks text content structures such as paragraph and sentences.

```
Hello <div type="paragraph">How are you</div>
```

The *sayas* tag provides important hints to the synthesizer on how to treat the text that follows. For example, "3/5" can be treated as a number or as a date.

```
<sayas class="date:dm">3/5</sayas>
<!-- spoken as "May third" -->
<sayas class="number">1/2</sayas>
<!-- spoken as "half" -->
<sayas class="phone">12345678901</sayas>
<!-- spoken as "1-234-567-8901" -->
<sayas class="net:email">jeff.zhuk@javaschool.com</sayas>
<!-- spoken as "Jeff dot Zhuk at Javaschool dot com" -->
<sayas class="currency">$9.95</sayas>
<!-- spoken as "nine dollars ninety five cents" -->
```

```
<sayas class="literal">IRS</sayas>

<!-- spoken as character-by-character "I.R.S" -->

<sayas class="measure">65kg</sayas>

<!-- spoken as "sixty five kilograms" -->
```

In addition, a voice tag specifies the speaking voice. For example:

```
<voice gender="male" age="20"> Do you want to send fax?</voice>
```

Notice that we can define the speaking voice in our Java code (see Fig. 12.1) or in JSML. Which way is preferable?

For the famous *Hello World* application, it is easier to specify voice characteristics directly in the code. JSML is a better choice for real-life production-quality applications; it gives more control of speech synthesis.

JSML Factory as One of the AVIFactory *Implementations*

JSML-based text can be generated on the fly by an appropriate lightweight presentation layer component of the application. No adjustment of the core services is required. We can use XML Stylesheet Language for Transformations (XSLT) to automatically convert core service contents into HTML or JSML format. Fig.12-2 shows the Object Model Diagram for the enterprise application capable to execute XML based scenarios with audio and video interfaces (AVI).
[Fig.12-2]

The *AVIFactory* interface on the left side of Fig. 12.2 has multiple implementations for audio and video presentation formats. Any *AVIFactory* implementation transforms data into JSML, HTML, or other presentation formats.

The *ScenarioPlayer* class plays a selected scenario and uses the *ServiceConnector* object to invoke any service that retrieves data according to a scenario. The *ScenarioPlayer* class creates the proper *AVIFactory* presenter object to transform data into the proper audio or video presentation format.

Speech Synthesis Markup Language

Is JSML the best way to represent voice data? We have to ask several more questions before we can answer this one. For example, what are the current standards in the speech synthesis and recognition area?

Speech Synthesis Markup Language (SSML) [3] is an upcoming W3C standard; SSML Specification Version 1.0 may be approved by the time this book is published.

JSML and SSML are not exactly the same (surprise!). Both are XML-based definitions for voice synthesis characteristics. Do not panic! It is relatively easy to map SSML to JSML for at least some voice characteristics. An example of an SSML document is provided in Fig. 12.3.

[Fig.12-3]

The differences between SSML and JSML are very apparent, but the similarities are even more impressive. Here is a brief review of basic SSML elements.

audio – allows insertion of recorded audio files

break – an empty element to set the prosodic boundaries between words

emphasis – increases the level of stress with which the contained text is spoken

mark – a specific reference point in the text

paragraph – indicates the paragraph structure of the document

phoneme – provides the phonetic pronunciation for the contained text

prosody – controls the pitch, rate, and volume of the speech output

say-as – indicates the type of text contained in the element; for example, date or number

sentence – indicates the sentence structure of the document

speak – includes the document body, the required root element for all SSML documents

sub – substitutes a string of text that should be pronounced in place of the text contained in the element

voice – describes speaking voice characteristics

Here is an example of another SSML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<speak version="1.0"
       xmlns="http://www.w3.org/2001/10/synthesis"
       xml:lang="en-US">
<sentence>
Your friendly reminder
<prosody pitch=”high” rate=”-20” volume=”load”>it is only
 <say-as type=”number”> 5 </say-as> days left till the Valentine
day</prosody>
</sentence>
</speak>
```

From SSML to JSML

A simple example of mapping SSML to JSML is provided with the *SSMLtoJSMLAudioFactory* class, which implements the *AVIFactory* interface and represents a version of the *AudioFactory* (Fig. 12.4).

[Fig.12-4]

The *SSMLtoJSMLAudioFactory* class implements the *AVIFactory* interface. There are at least six methods that must be provided in this class. Fig. 12.4 displays four of them.

The *init()* method initializes an original source. The *initComponents()* method creates the synthesizer object and can use the *getVoices()* method considered in Fig. 12.1 to initialize voice components. The *getHeader()* method returns the standard JSML header. The *getFooter()* method returns the standard JSML footer.

The *getBody()* method extracts a speakable text from the original SSML source. The “speak” tags frame this text.

And finally, we can map SSML tags to appropriate JSML tags and provide a standard-based implementation.

Play JSML

Note that the *SSMLtoJSMLAudioFactory* class implements the *Speakable* interface. The *Speakable* interface is the spoken version of the *toString()* method of the *Object* class. Implementing the *Speakable* interface means implementing the *getJSMLText()* method. The *getJSMLText()* method, as well as the *play()*, method is shown in Fig.12.5.

[Fig.12-5]

With the source code presented in Fig. 12.4, the *getJSMLText()* method can be implemented as a single line that concatenates the header, transformed body, and footer of the JSML string.

The *play()* method uses the factory properties to present the JSML content. The *play()* method starts by checking whether the synthesizer is ready to talk. Then it uses the synthesizer object to invoke its *speak()* method, passing the JSML string as one of the arguments.

The other argument is a *SpeakableListener* object. This object can be used to receive and handle events associated with the spoken text. We do not plan to use the *SpeakableListener* object in our example, so we passed the null object as the second argument. It is possible to use the normal mechanisms for attachment and removal of listeners with the *addSpeakableListener()* and *removeSpeakableListener()* methods.

Speech Recognition with Java

Speech technologies are not limited by speech synthesis. Speech recognition technologies have matured to the point that the default message “Please repeat your selection” is not so commonplace anymore, and human-computer conversation can go beyond multiple-choice menus.

There are recognizer programs for personal use, corporate sales, and other purposes. Most personal recognizers support dictation mode. They are speaker-dependent, requiring “program training” that creates a "speaker profile" with a detailed map of the user's speaking patterns and accent. Then, the program uses this map to improve recognition accuracy.

The JSAPI offers a recognizer that provides an optional *SpeakerManager* object that allows an application to manage the *SpeakerProfiles* of that recognizer. The *getSpeakerManager()* method of the recognizer interface returns the *SpeakerManager* if this option is available for this recognizer. Recognizers that do not maintain speaker profiles – known as speaker-independent recognizers – return *null* for this method. A single recognizer may have multiple *SpeakerProfiles* for one user and may store the profiles of multiple users.

The *SpeakerProfile* class is a reference to data stored with the recognizer. A profile is identified by three values: its unique ID, its name, and its variant. The ID and the name are self-explanatory. The variant identifies a particular enrollment of a user and becomes useful when one user has more than one enrollment, or *SpeakerProfile*.

Additional data stored by a recognizer with the profile may include:

- Speaker data such as name, age, and gender
- Speaker preferences
- Data about the words and word patterns of the speaker (language models)
- Data about the pronunciation of words by the speaker (word models)
- Data about the speaker's voice and speaking style (acoustic models)

- Records of previous training and usage history

Speech recognition systems (SRS) can listen to users and, to some degree, recognize and translate their speech to words and sentences. Current speech technologies have to constrain speech context with grammars. Today, the systems can achieve “reasonable” recognition accuracy only within these constraints.

The Java Speech Grammar Format

The Java Speech Grammar Format (JSGF) [4] is a platform and vendor-independent way of describing a **rule grammar** (also known as a **command and control grammar** or **regular grammar**).

A rule grammar specifies the types of **utterances** a user might say. For example, a service control grammar might include *Service* and *Action* commands.

A voice application can be based on a set of scenarios. Each scenario knows the context and provides appropriate grammar rules for the context.

Grammar rules may be provided in a multilingual manner. For example:

```
<greetings.english.hello>

<greetings.russian.privet>

<greetings.deutsch.gutenTag>
```

Hello, *Privet*, and *GutenTag* are tokens in the grammar rules. Tokens define expected words that may be spoken by a user. The world of tokens forms a **vocabulary** or **lexicon**. Each record in the vocabulary defines the **pronunciation** of the token.

A single file defines a single grammar with its header and body. The header defines the JSGF version and (optional) encoding. For example:

```
#JSGF V1.0 ISO8859-5;
```

The grammar starts with the grammar name and is similar to Java package names. For example:

```
grammar com.its.scenarios.examples.greetings;
```

We can also import grammar rules and packages, as is usually done in Java code:

```
import <com.its.scenarios.examples.cyc.*> ; // talk to knowledge base
```

The grammar body defines *rules* as a rule name followed by its definition-token.

The definition can include several alternatives separated by "|" characters. For example:

```
<login> = login ;

<find> = find | search | get | lookup ;

<new> = new | create | add ;

<command> = <find> | <new> | <login>  ;
```

We can use the Kleene star (named after Stephen Cole Kleene, its originator) or the "+" character to set expectations that a user can repeat a word multiple times.

```
<agree> = I * agree | yes | OK ; //  "I agree" and "agree" - both
covered

<disagree> = no +  ; // no can be spoken 1 or more times
```

The Kleene star and the + operator are both **unary operators** in the JSGF. There is also the tag unary operator that helps to return application-specific information as the result of recognition. For example:

```
<service> = (mail | email ) {email} | (search | research | find) {find}
;
```

The system returns the word *email* if *mail* or *email* is spoken. When *search, research*, or *find* is uttered, the system returns the word *find*.

Today, the mainstream of speech recognition technology lies outside the JSAPI. (This may be different next year.) One example of a current technology is the open source Sphinx [5] project written in C++ by a group from Carnegie Mellon University.

In the Sphinx system, recognition takes place in three passes (the last two are optional): lexical-tree Viterbi search, flat-structured Viterbi search, and global best-path search.

Improving Sphinx Recognition Rate with Training

Sphinx can be trained to better satisfy a specific client with the SphinxTrain program. Even after training, the rate of accuracy for Sphinx-2 is about 50%; for Sphinx-3, delivered at the end of 2002, the rate is about 70%. In comparison, the rate of accuracy in Microsoft's Speech Software Development Kit (SDK) recognition engine is 95% after voice training and microphone calibration. There are more products on the market today, like IBM ViaVoice Speech SDK, or Dragon NaturallySpeaking products that produce even better results although they are not freely available.

Microsoft Speech SDK

The Microsoft Speech SDK [6] is based on the Microsoft Speech API (SAPI), a layer of software that allows applications and speech engines to communicate in a standardized way. The MS Speech SDK provides both text-to-speech (TTS) and speech recognition functions.

Fig. 12.6 illustrates the TTS synthesis process. The main elements participating in TTS conversion are:

ISpVoice – the interface used by the application to access the TTS function

ISpTTSEngineSite – the engine interface to speak data and queue events

IspObjectWithToken* – *the interface to create and initialize the engine

ISpTTSEngine* – *the interface to call the engine

IspTokenUI* – *the way for the SAPI control panel to access the User Interface

[Fig.12-6]

The speech recognition architecture looks even simpler (Fig. 12.7). The main speech recognition elements interact in the following way:

1. The engine uses the *ISpSREngineSite* interface to call the SAPI to read audio and returns the recognition results.
2. The SAPI calls the engine using the methods of the *ISpSREngine* interface to pass details of recognition grammars. SAPI also uses these methods to start and stop recognition.
3. The *IspObjectWithToken* interface provides a mechanism for the engine to query and edit information about the object token.
4. *ISpTokenUI* represents User Interface components that are callable from an application.

[Fig. 12.7]

SAPI 5 synthesis markup is not exactly SSML; it is closer to the format published by the Sable Consortium. SAPI XML tags provide functions such as volume control and word emphasis. These tags can be inserted into text passed into *ISpVoice::Speak* and text streams of format SPDFID_XML, which are then passed into *ISpVoice::SpeakStream* and auto-detected (by default) by the SAPI XML parser. In the case of an invalid XML structure, a speak error may be returned to the application. We can change rate and volume attributes in real time using *ISpVoice::SetRate* and *ISpVoice::SetVolume*.

Volume

The Volume tag controls the volume of a voice and requires just one attribute: Level, an integer between 0 and 100. The tag can be empty, to apply to all following text, or it can frame content to which alone it applies.

```
<volume level="50">This text should be spoken at volume level fifty.
   <volume level="100">
      This text should be spoken at volume level one hundred.
   </volume>
</volume>
<volume level="80"/>All text which follows should be spoken at volume
level eighty.
```

Rate

The Rate tag defines the rate of a voice with one of two attributes: Speed and AbsSpeed. The Speed attribute defines the relative increase or decrease of the speed value, whereas AbsSpeed defines its absolute rate value: an integer between –10 and 10. The tag can be empty, to apply to all following text, or it can frame content to which alone it applies.

```
<rate absspeed="5">
   This text should be spoken at rate five.
   <rate speed="-5">
      Decrease the rate to level zero.
   </rate>
</rate>
<rate absspeed="10"/> Speak the rest with the rate 10.
```

Pitch

In a very similar manner, the Pitch tag controls the pitch of a voice with one of two attributes: Middle and AbsMiddle; an integer between –10 and 10 can represent an absolute as well as relative value.

```
<pitch absmiddle="5">
This text should be spoken at pitch five.
   <pitch middle="-5">
      This text should be spoken at pitch zero.
   </pitch>
</pitch>
<pitch absmiddle="10"/> All the rest should be spoken at pitch ten.
```

Zero represents the default level for rate, volume, and pitch values.

Emph

The Emph tag instructs the voice to emphasize a word or section of text. The Emph tag cannot be empty.

```
Your <emph>American Airline </emph> flight departs at <emph>eight
</emph> tonight
```

Voice

The Voice tag defines a voice based on its Age, Gender, Language, Name, Vendor, and VendorPreferred attributes, which can be required or optional. These correspond exactly to the required and optional attribute parameters for the *ISpObjectTokenCategory_EnumerateToken* and *SpFindBestToken* functions.

If no voice matching all the required attributes is found, a voice change will not occur. Optional attributes are treated differently. In this case, an exact match is not necessarily expected. A voice that is closer to the provided attributes will be selected over one that is less similar. For example:

```
The default voice should speak this sentence.
<voice required="Gender=Female;Age!=Child">
A female non-child should speak this sentence, if one exists.
<voice required="Age=Teen">
   A teen should speak this sentence. If a female, non-child teen voice
is present; this voice will be selected over a male teen voice, for
example.
   </voice>
</voice>
```

Let us consider a demonstration program that uses the TTS and speech recognition facilities of the Microsoft Speech SDK.

Speech Technology to Decrease Network Bandwidth

The application is a conference between multiple clients over the Internet. The application utilizes speech technology to significantly decrease network bandwidth. Client programs intercept a user's speech, translate it to text, and send ASCII text over the Internet to the server-dispatcher. The server broadcasts the text it receives to the other clients (a regular chat schema). Client programs receive text from the server and convert it back to speech. Fig. 12.8 illustrates the application with a diagram.

[Fig.12-8]

Other application details are:

- Speech recognition and TTS are done on the client side.
- The client recognizes a phrase.
- Plain text is transmitted between the client and the server.
- The client program appends metadata, such as the user's name, in SSML format.
- The server side can be implemented in C++, C#, or Java, using TCP/IP sockets.

An example of the *TalkingClient* program can be found in Fig. 12.9. The *TalkingClient* program takes the user's name as an optional argument. If the argument is not provided, the default voice is used.

[Fig.12-9]

The main routine starts with the socket connection to a recipient. We then initialize the speech engine and define the break signal that will be used to indicate when to send to the server what the user has said. In the example, the break signal is the word "okay."

The main processing happens in the *while* loop. The speech engine recognizes the user's sentence and prints it on the screen. Then, the program sends the resulting text to the server with the additional SAPI XML Voice tag:

```
<voice optional="Gender=userGender;Age=userAge;Name=userName">
resultingText</voice>
```

The other part of the application is presented in Fig. 12.10. The requirements of the *ListeningClient* are to receive text from the server and to transform text to speech using the voice profile, if it is available

[Fig.12-10]

The *ListeningClient* program starts in a very similar manner. It uses the *ReceiveLine* method of the *SocketClient* to listen to messages coming from the server. The program converts every unit of speech into voice and displays the line on the screen. Examples of *Socket* classes for Windows can be found online [7].

Standards for Scenarios for Speech Applications

Let us stop this overview of the parts of speech recognition technology for a minute. All of the pieces are important: some are more important for system programmers who do the groundwork, whereas others target application developers. Application developers can use this groundwork to describe application flow and write interpretation scenarios.

Our next step is to write scenarios for speech applications. Let us consider current and future standards that may help. Note that the Microsoft .NET Speech SDK uses SSML, which is a part of the W3C Speech Interface Framework (unlike the Microsoft Speech SDK that uses SAPI XML). SSML is a markup language that defines TTS processing, which is the simplest part of speech technology. There are two markup languages that can describe a complete speech interface: Speech Application Language Tags (SALT) [8], a relatively new standard, and VoiceXML [9], a well-established technology with many implementations.

VoiceXML was developed for telephony applications as a high-level dialog markup language that integrates speech interface with data and provides a full control of the application flow. Unlike VoiceXML, SALT offers a lower-level interface that strictly focuses on speech tags but targets several devices, including but not limited to telephone systems.

VoiceXML, as well as SALT, uses standards of the W3C Speech Interface Framework such as SSML and Speech Recognition Grammar Standard (SRGS) [10]. SALT also includes recommendations on Natural Language Semantics Markup Language (NLSML) [11] as a recognition result format, and Call Control XML (CCXML) [12] as a call control language.

In a nutshell: NLSML is an XML-based markup language for representing the meaning of a natural language utterance, and CCXML provides telephony call control support for VoiceXML or SALT, and other dialog systems.

NLSML uses an XForms data model for the semantic information that is returned in the interpretation. (See the NLSML and XForms overviews in Appendix 2.)

SALT provides facilities for multimodal applications that may include not only voice but also screen interfaces. SALT also gives developers the freedom to embed SALT tags into other languages. This allows for more flexibility in writing speak-and-display scenarios.

Speech Application Language Tags

SALT consists of a relatively small set of XML elements. Each XML element has associated attributes and Document Object Model (DOM) properties, events, and methods. One can write speech interfaces for voice-only and multimodal applications using SALT with HTML, XHTML, and other standards. SALT controls dialog scenarios through the DOM event model that is popular in Web software.

The three top-level elements in SALT are <*listen…*>, <*prompt…*>, and <*dtmf…*>. The first two XML elements define speech engine parameters: <*listen…*> configures the speech recognizer, executes recognition, and handles speech input events; <*prompt…*> configures the speech synthesizer and plays out prompts

The third XML element plays a significant role in call controls for telephony applications. <*dtmf…*> configures and controls dual-tone multifrequency (DTMF) signaling in telephony applications. Telephony systems use DTMF to signal which key has been pressed by a client. Regular phones usually have twelve keys: ten decimal digit keys and "#" and "*" keys. Each key corresponds to a different pair of frequencies.

The <*listen*> and <*dtmf*> elements may contain <*grammar*> and <*bind*> elements. The <*listen*> element may also include the <*record*> element.

The <*grammar*> element defines grammars. A single <*listen*> element may include multiple grammars. The <*listen*> element may have methods to activate an individual grammar before starting recognition. SALT itself is independent of the grammar formats, but for interoperability, it recommends supporting at least the XML form of the W3C SRGS.

The <*bind*> element can inspect the results of recognition and provide conditional copy-actions. The <*bind*> element can cause the relevant data to be copied to values in the containing page. A single <*listen*> element may contain multiple <*bind*>s. <*bind*> may have a *conditional test* attribute as well as a *value* attribute. <*bind*> uses XPath syntax (see Appendix 2 and other XML standards mentioned in this book) in its *value* attribute to point to a particular node of the result. <*bind*> uses an XML pattern query in its *conditional test* attribute. If the condition is true, the content of the node is bound into the page element specified by the *targetElement* attribute. The *onReco* event handler with script programming can provide even more complex processing. The <*onReco*> and <*bind*> elements are triggered on the return of a recognition result.

The <*record*> element can specify parameters related to speech recording. <*bind*> or scripted code can process the results of the recording, if necessary.

A Spoken Message Scenario

Fig. 12.11 demonstrates a scenario in which dialog flow is provided with a client-side script. The scenario is actually an HTML page with embedded SALT tags and script functions. The *askForService()* script activates the SALT <*listen*> and <*prompt*> tags. For example, *askName.Start()* prompts the user with “What is your name?” and the following *nameRecognition.Start()* examines the recognition results. The *askForService()*

script executes the relevant prompts and recognitions until all values are obtained. Successful message recognition triggers the *submit()* function, which submits the message to the recipient.

[Fig.12-11]

The user’s name not only serves as the user’s signature but can also invoke a chosen voice profile, if available, on the recipient side.

Did you notice the reference to the *spokenMessage.xml* grammar file that supports the scenario in the code? How do we define grammar?

Grammar Definition

First, let us look into the existing Command and Control features of the MS Speech SDK. The Command and Control features of SAPI 5 are based on context-free grammars (CFGs). A CFG defines a specific set of words and the sentences that are valid for recognition by the speech recognition engine.

The CFG format in SAPI 5 uses XML to define the structure of grammars and grammar rules. SAPI 5–compliant speech recognition engines expect grammar definitions in a binary format produced by any CFG/Grammar compiler – for example, *gc.exe*, the SAPI 5 grammar compiler included in Speech SDK. Compilation is usually done before application run-time but can be done at run-time.

Here is an example of a file that provides grammar rules to navigate through mail messages (“next”, “previous”) and to retrieve the currently selected email (“getMail”).

```
<GRAMMAR LANGID="409">
    <DEFINE>
        <ID NAME="VID_MailNavigationRules" VAL="1"/>
        <ID NAME="VID_MailReceiverRules" VAL="2"/>
    </DEFINE>
    <RULE ID="VID_MailNavigationRules" >
      <L>
       <P VAL="next">
          <o>Please *+</o>
            <p>next</p>
            <o>message\email\mail</o>
         </P>
       <P VAL="previous">
          <o>Please *+</o>
            <p>previous\last\back</p>
            <o>message\email\mail</o>
         </P>
      </L>
    </RULE>
    <RULE ID="VID_MailReceiverRules" TOPLEVEL="ACTIVE">
        <O>Please</O>
        <P>
            <L>
                <P val="getMail">Retrieve</P>
                <P val="getMail">Receive</P>
                <P val="getMail">Get</P>
            </L>
        </P>
        <O>the mail</O>
    </RULE>
</GRAMMAR>
```

Appendix 3 provides more examples (along with C# program source code) for a speech application based on SAPI 5. The grammar file can be dynamically loaded and compiled at run-time, decreasing the number of choices for any *current* recognition and improving recognition quality.

VoiceXML

The last, but definitely not least important, technology on the list is VoiceXML. Although SALT and VoiceXML have different targets, in some ways, they compete in the speech technology arena. Unlike SALT, which is relatively new, VoiceXML started in 1995, as part of an AT&T project called Phone Markup Language (PML).

The VoiceXML Forum was formed in 1998-1999 by AT&T, IBM, Lucent, and Motorola. At that time, Motorola had developed VoxML and IBM was developing its own SpeechML. The VoiceXML Forum helped integrate these two efforts. Since then, VoiceXML has had a history of successful implementations by many vendors.

Unlike SALT, which is a royalty-free, upcoming standard, VoiceXML may be subject to royalty payments. Several companies, including IBM, Motorola, and AT&T, have indicated they may have patent rights to VoiceXML.

This brief overview of VoiceXML is based on the VoiceXML2.0 Specification submitted to W3C at the beginning of 2003.

What Is VoiceXML?

VoiceXML is designed for creating dialog scenarios with digitized audio, speech recognition, and DTMF key input. VoiceXML can record spoken input, telephony, and mixed-initiative conversations. The mixed conversation is an extended case of the most common type of computer-human conversations directed by the computer. The main target of VoiceXML is Web-based development and content delivery to interactive speech applications.

The VoiceXML interpreter renders VoiceXML documents audibly, just as a Web browser renders HTML documents visually. However, standard Web browsers run on a local machine, whereas the VoiceXML interpreter runs at a remote hosting site. Fig. 12.12 displays the enterprise application with multimodal access to business services.

[Fig.12-12]

Like HTML Web pages, VoiceXML documents have Web URLs and can be located on any Web server. VoiceXML pages deliver the service content via speech applications.

Main Components of Speech Recognition Systems

Speech recognition systems in general and VoiceXML systems in particular rely on high-performance server-side hardware and software located on or connected to the Web container. The Web container is the architecture tier responsible for correspondence to clients over HTTP and dispatching client requests to proper business services. In this case, speech recognition services become the client that intercepts voice flow and translates it into HTTP streams. The key hardware factors for delivering reliable, scalable VoiceXML applications are:Telephony connectivity

- Internet connectivity
- Scalable architecture
- Caching and media streaming
- CODECs – combinations of analog-to-digital (A/D) and digital-to-analog (D/A) signal converters

Progress in hardware technologies such as the high-speed, low–power consumption digital signal processor (DSP) has substantially contributed to improving CODEC conversion efficiency.

The SRS platform contains intelligent caching technology that minimizes network traffic by caching VoiceXML, audio files, and compiled grammars. The VoiceXML platform makes extensive use of load balancing, resource pooling, and dynamic resource allocation. SRS servers use multithreaded C++ implementations, delivering the most performance from available hardware resources. To prevent unnecessary recompilation of grammars, the VoiceXML platform uses a high-performance indexing technique to cache and reuse previously compiled grammars.

Voice services offer the following software components to implement an end-to-end solution for phone-accessible Web content:

- Telephony platform – software modules for TTS, voice recognition
menu systems, parsing engines, and DTMF Standard support

– Open-system architecture in compliance with industry standards: VoiceXML, Wireless Application Protocol (WAP), Wireless Markup Language (WML), XHTML, SSML, SRGS, NLSML, etc.

- WAP solution – support for using WAP to deliver Web and audio content to new Web phones and enabling seamless integration between Web and audio content.

- Voice application and activation – user interface and logic (such as personalization) for accessing back-end audio content, and Web and email databases for easy phone access

What Is the VoiceXML Architecture, and How Does It Work? A document (Web) server contains VoiceXML documents or VXML pages with dialog based scenarios. (I try to use the word “scenario” on every other page, but sometimes the word sneaks in-between.)

The document server responds to a client request by sending the VoiceXML document to an SRS or a VoiceXML implementation platform (the VoiceXML interpreter). A voice service scenario is a sequence of interaction dialogs between a user and an implementation platform.

Document servers perform business logic, database, and legacy system operations, and produce VoiceXML documents that describe interaction dialogs. User input affects dialog interpretation by the VoiceXML interpreter. The VoiceXML interpreter transforms user input into requests submitted to a document server. The document server replies with other VoiceXML documents describing new sets of dialogs.

How Does the VoiceXML Document Look Like?

A VoiceXML document can describe the:

- Output of synthesized speech (TTS)
- Output of audio files
- Recognition of spoken input
- Recognition of DTMF input
- Recording of spoken input
- Control of dialog flow

VoiceXML requires a common grammar format – namely, the XML Form of the W3C SRGS – to facilitate interoperability.

A voice application is a collection of one or more VoiceXML documents sharing the same **application root document**. A VoiceXML document is composed of one or more dialogs. The application entry point is the first VoiceXML document that the VoiceXML interpreter loads when it starts the application. The developer's task is to provide voice commands to the user in the most comfortable way while offering clearly distinguished possibilities of responses expected from the user through voice and/or telephone keys.There are two kinds of dialogs: **forms** and **menus**. Forms define an interaction that collects field values. Each field may specify a grammar with expected inputs for that field. A menu commonly asks the user to choose one of several options and then uses the choice to transition to another dialog.

Fig. 12.13 presents a very simple example of a VoiceXML document. This VoiceXML document provides a form dialog that offers users a choice of services to fill the service field. Expected answers are provided in the grammar document *com.its.services.grxml*.

[Fig.12-13]

Each dialog has one or more speech and/or DTMF grammars associated with it. Most of the speech applications today are **machine directed**. A single dialog grammar is active at any current time for machine-directed applications, the grammar associated with a current user dialog. In **mixed-initiative** applications, the user and the machine alternate in determining what to do next. In this case, more than one dialog grammar may be active and the user can say something that matches another dialog's grammar. Mixed initiative adds flexibility and power to voice applications.

VoiceXML can handle events not covered by the form mechanism described above. There are default handlers for the predefined events; plus, developers can override these handlers with their own event handlers in any element that can throw an event. The platform throws events, for example, when the user does not respond, does not respond intelligibly, or requests help.

The <*catch*>, <*error*>, <*help*>, <*noinput*>, and <*nomatch*> elements are examples of event handlers. For example, the <*catch*> element can detect a disconnect event and provide some action upon the event:

```
 <catch event="connection.disconnect.hangup">
    <submit namelist="disconnect"
next="http://javaschool.com/school/public/speech/vxml/exit.jsp"/>
</catch>
```

Applications can support help by putting the help keyword in a grammar in the application root document.

```
<help>  <prompt>Say "Retry" to retry authorization, or "Register" to
hear the registration instructions. Say "Exit" or "Goodbye" to exit.
  </prompt>   <listen/>
</help>
```

A List of VoiceXML Elements

VoiceXML elements include:

<assign> – assigns a value to a variable

<audio> – plays an audio clip within a prompt

<block> – a container of (noninteractive) executable code

<catch> – catches an event

<choice> – defines a menu item

<clear> – clears one or more form item variables

<disconnect> – disconnects a session

<else> – used in <if> elements

<elseif> – used in <if> elements

<enumerate> – shorthand for enumerating the choices in a menu

<error> – catches an error event

<exit> – exits a session

<field> – declares an input field in a form

<filled> – an action executed when fields are filled

<form> – a dialog for presenting information and collecting data

<goto> – goes to another dialog in the same or a different document

<grammar> – specifies a speech recognition or DTMF grammar

<help> – catches a help event

<if> – simple conditional logic

<initial> – declares initial logic upon entry into a mixed initiative form

<link> – specifies a transition common to all dialogs in the link's scope

<log> – generates a debug message

<menu> – a dialog for choosing among alternative destinations

<meta> – defines a metadata item as a name/value pair

<metadata> – defines metadata information using a metadata schema

<noinput> – catches a noinput event

<nomatch> – catches a nomatch event

<object> – interacts with a custom extension

<option> – specifies an option in a <field>

<param> – parameter in <object> or <subdialog>

<prompt> – queues speech synthesis and audio output to the user

<property> – controls implementation platform settings

<record> – records an audio sample

<reprompt> – plays a field prompt when a field is revisited after an event

<return> – returns from a subdialog

<script> – specifies a block of ECMAScript client-side scripting logic

<subdialog> – invokes another dialog as a subdialog of the current one

<submit> – submits values to a document server

<throw> – throws an event

<transfer> – transfers the caller to another destination

<value> – inserts the value of an expression in a prompt

<var> – declares a variable

<vxml> – top-level element in each VoiceXML document

Service Order with VoiceXML Example

Fig. 12.14 introduces a typical VoiceXML document that initiates a brief phone conversation. The source code starts with the standard XML and then has VoiceXML reference lines. The next thing we see is a *form* that looks almost exactly like an HTML form. In fact, the form has exactly the same purpose – to collect information from a user into the form fields. This form has a single field named *course*.

[Fig.12-14]

The program prompts the user to choose one of the training courses.

```
<prompt>Which course do you want to take?</prompt>
```

The grammar line above the prompt defines a grammar rules file that will try to resolve the answer.

```
   <grammar type="application/srgs+xml"
src="/grammars/training.grxml"/>
```

The user might want to talk to a human being. In this case, the grammar rules might resolve the user's desire and return the word *operator* as the user's selection. The program uses an <*if*> element to check on this condition.

```
   <if cond="course == 'operator' ">
```

If this condition is true, the program will use the <*goto*> element to jump to another document that transfers the caller to the operator. Note that all tags are properly closed, as they should be in any XML file.

Looking down the code below the <*if*> element, we find <*noinput*> and <*nomatch*> event handlers. If the user produces no input during the default time, the program plays the prompt again using the <*reprompt*/> element.

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

The most interesting script starts when a user selection is not expected. In this case, the <*nomatch*> event handler is fired. This element may have a counter, which we use here to try to provide a more appropriate response and possibly decrease the user's discomfort.

The very first <*nomatch*> element provides an additional hint to the user and reprompts the original message.

```
    <nomatch count="1">
      Please select a Java, Wireless, or Ontology course from the list.
      <reprompt/>
    </nomatch>
```

The next time the user makes a strange selection, the program offers its candid advice.

```
    <nomatch count="2">
      <prompt>
        I am sorry, we have so many types of training courses, but not
this one.
        I recommend you start with the Ontology Introduction course at
this time.
        Will that work for you?
      </prompt>
    </nomatch>
```

The third *nomatch* event switches the user to the operator.

```
<nomatch count="3">
      I will switch you to the operator.
      Hopefully, you will find the course you want.
      Good luck.
```

```
    <goto
next="http://javaschool.com/school/public/speech/vxml/operator.vxml" />
 </nomatch>
```

But what if the user was successful in the course selection? In this case, the selected course value fills the *course* field and the value is submitted to the training page.

```
   <block>
     <submit
next="http://javaschool.com/school/public/speech/vxml/training.jsp"/>
    </block>
```

The last two lines close the form and the VoiceXML document.

```
  </form>
</vxml>
```

Wow!

Play Audio and Transfer Phone Calls with VoiceXML

How does VoiceXML do the transfer operation? Here is the code:

```
<!-- Transfer to the operator -->
 <!-- Say it first -->
 Transferring to the operator according your request.
 <!-- Play music while transfer -->
 <!-- Wait up to 60 seconds for the transfer  -->
 <transfer dest="tel:+1-234-567-8901"
   transferaudio="music.wav" connecttimeout="60s">
 </transfer>
```

The code extract first says, “Transferring to the operator according to your request” and then actually tries to transfer the user to the operator. The <*transfer*> element in our example turns on some music and sets the timeout to 60 seconds for the transfer. There is another essential part of the <*transfer*> element – the telephone number of the operator.

Here is a transfer example that shows what happens when the program catches the *busy* event:

```
  <transfer maxlength="60" dest="8005558355">
   <catch event="event.busy">
    <audio> busy </audio>
    <goto next="_home"/>
   </catch>
</transfer>
```

Listen to E-Mail Messages with VoiceXML

The <*link*> element below navigates to *mail.vxml* whenever the user says "mail."

```
<link next="mail.vxml">
   <grammar type="application/srgs+xml" root="root" version="1.0">
     <rule id="root" scope="public">mail</rule>
  </grammar>
 </link>
```

Composee and Send a New Message with VoiceXML in-line grammar rules

This example provides in-line grammar rules, unlike most of the following examples, in which we reference grammar rule files.

The <*subdialog*> element helps to create reusable dialog components and decompose an application into multiple documents.

```
<subdialog name="compose" src="newmail.vxml">
      <filled>
        <!-- The "compose" subdialog returns 3 variables below.
          These variables must be specified in the "return" element
          of the “compose” -->
        <assign name="to_address" expr=" compose.to_address"/>
        <assign name="subject" expr=" compose.subject"/>
        <assign name="message" expr=" compose.body"/>
      </filled>
 </subdialog>
```

Fig. 12.15 provides an example of the *new_mail* service request. The example uses the *compose* subdialog to fill two fields for the new mail form. The *compose* subdialog returns its status and two requested fields. If the returned status is not OK, the service says the message cannot be delivered and exits. Otherwise, the service assigns returned values to the *to_name* and *message* fields, and prompts the user for the message subject. It often happens that mail goes out without any subject.

[Fig.12-15]

The subject can serve as communication metadata, which makes even more sense today when computer systems are increasingly involved in the communication process. With this last field, the service is ready to rock and roll and submits all the data to the long URL provided in the <*submit*> element.

The *new_mail* service listing also illustrates the use of *if-else* elements with the conditional actions described above.

Fig. 12.16 displays the *compose* subdialog. In the *compose* dialog, the prompt asks for a recipient's name. Apparently, the grammar rules behind the scenes are working hard to recover the email address from the list of available names. The user can ask for help to hear more detailed prompt messages. If the name recognition fails, the <*nomatch*> element returns the status value *not_known_name*, back to the *new_mail* service.

[Fig.12-16]

In the best-case scenario, when name recognition succeeds, the *compose* dialog sets the status value to OK and prompts the user to fill (answer) the *message* field. The *compose* dialog then returns the OK status and two variables (the *to_name* and the *message*), back to the *new_mail* service.

The *forward_mail* service reuses the same *compose* subdialog to collect the *to_address* and *message* fields. Fig. 12.17 shows the *forward_mail* VoiceXML page. The *forward_mail* listing includes two additional variables: *subject* and *old_message*. These variables passed as parameters extracted by the *mail_service* dialog from the original mail. The *forward_mail* service behaves like the *new_mail* service.

[Fig.12-17]

If the *compose* subdialog returns an OK status with the two requested fields (*to_address* and *message*), the *forward_mail* service will submit all data, including the two additional fields (*subject* and *old_message*) that came as parameters from the original email, to the final URL. If the status returned by the *compose* subdialog is not as cheerful, the *forward_mail* service will not forward the message but will exit instead.

Parameters can be passed with the <*param*> elements of a <*subdialog*>. These parameters must be declared in the subdialog using <*var*> elements, as displayed in Fig. 12.17.

The *mail_service* dialog passes the parameters to the *forward_mail* service with the following lines:

```
<form>
<subdialog name="forward_mail" src="forward_mail.vxml">
   <param name="subject" expr=" ‘ Hello’ "/>
   <param name=”old_message” expr=” ‘How are you?’ “/>
   <filled>
     <submit
next="http://javaschool.com/school/public/speech/mail.jsp"/>
   </filled>
  </subdialog>
</form>
```

SSML Elements in VoiceXML

Looking into the *prompt* examples in Fig. 12.16, we can see the tags we learned before as SSML elements. No wonder. The VoiceXML 2.0 Specifications model the content of the <*prompt*> element based on W3C SSML 1.0 and makes available the following SSML elements:

<audio> – specifies audio files to be played and text to be spoken

<break> – specifies a pause in the speech output

<desc> – provides a description of a nonspeech audio source in <audio>

<emphasis> – specifies that the enclosed text should be spoken with emphasis

<lexicon> – specifies a pronunciation lexicon for the prompt

<mark> – ignored by VoiceXML platforms

<metadata> – specifies XML metadata content for the prompt

<paragraph> (or <p>) – identifies the enclosed text as a paragraph, containing zero or more sentences

<phoneme> – specifies a phonetic pronunciation for the contained text

<prosody> – specifies prosodic information for the enclosed text

<say-as> – specifies the type of text construct contained within the element

<sentence> (or <s>) – identifies the enclosed text as a sentence

<sub> – specifies replacement spoken text for the contained text

<voice> – specifies voice characteristics for the spoken text

The example in Fig. 12.18 uses the <*record*> element to collect an audio recording from the user. This example also uses the *bargein* property that controls whether a user can interrupt a prompt. Setting the *bargein* property to “true” allows the user to interrupt the program, introducing a mixed initiative.

[Fig.12-18]

Record Audio and Accept Telephone Keys with VoiceXML

The program prompts the user to record her or his message. A reference to the recorded audio is stored in the *msg* variable. There are several important settings in the <*record*> element, including timeouts and DTMFTERM.

The recording stops under one of the following conditions: a final silence for more than three seconds occurs, a DTMF key is pressed, the maximum recording time – ten seconds – is exceeded, or the caller hangs up. The audio message will be sent to the Web server via the HTTP POST method with the *enctype="multipart/form-data".*

Another example, in Fig. 12.19, demonstrates a VoiceXML feature that allows the user to enter text messages using a telephone keypad. VoiceXML supports platforms with telephone keys. In Fig. 12.19, the user is prompted to type the message. The <*block*> element copies the message to the variable *document.key_message*. The example Fig. 12.9 below shows the use of the <*object*> element, a part of ECMAScript [13].

[Fig.12-19]

ECMAScript

Developed by the European Computer Manufacturers Association (ECMA), ECMAScript was modeled after JavaScript but designed to be application independent. The language was divided into two parts: a domain-independent core and a domain-specific object model. ECMAScript defines a language core, leaving the design of the domain object model to specific vendors.

An ECMAScript object, presented in the example, may have the following attributes:

name - When the object is evaluated, it sets this variable to an ECMAScript value whose type is defined by the object.

expr - The initial value of the form item variable; default is the ECMAScript value "*undefined*". If initialized to a value, then the form item will not be visited unless the form item variable is cleared.

cond - An expression that must evaluate to true after conversion to boolean in order for the form item to be visited.

classid - The URI specifying the location of the object's implementation. The URI conventions are platform-dependent.

codebase - The base path used to resolve relative URIs specified by classid, data, and archive. It defaults to the base URI of the current document.

codetype - The content type of data expected when downloading the object specified by *classid*. The default is the value of the type attribute.

data - The URI specifying the location of the object's data. If it is a relative URI, it is interpreted relative to the *codebase* attribute.

type - The content type of the data specified by the data attribute.

archive - A space-separated list of URIs for archives containing resources relevant to the object, which may include the resources specified by the *classid* and data attributes.

ECMAScript provides scripting capabilities for Web-based client-server architecture and makes it possible to distribute computation between the client and server. Each Web browser and Web server that supports ECMAScript supports (in its own way) the ECMAScript execution environment. Some of the facilities of ECMAScript are similar to Java and Self [14] languages.

An ECMAScript program is a cluster of communicating objects that consist of an unordered collection of *properties* with their ***attributes.*** Attributes, such as *ReadOnly*, *DontEnum*, *DontDelete,* and *Internal*, determine how each property can be used. For example, a property with the *ReadOnly* attribute is not changeable and not executable by ECMAScript programs, the *DontEnum* properties cannot be enumerated in the programming loops, your attempts to delete the *DontDelete* properties will be ignored, and the *Internal* properties are not directly accessible via the property access operators.

ECMAScript properties are containers for objects, primitive values, or methods. A primitive value is a member of one of the following built-in types: *Undefined, Null, Boolean, Number,* and *String*.

ECMAScript defines a collection of **built-in objects** that include the following object names: *Global*, *Object, Function, Array*, *String* (yes, there are objects with the same names as built-in primitive types), *Boolean, Number, Math, Date, RegExp*, and several *Error* object types.

ECMAScript in VoiceXML Documents

Fig. 12-20 presents ECMAScript embedded into the *forward_mail* subdialog. Several lines of the ECMAScript give the final touch to the *forward_mail* dialog. The subject of the forwarded message starts with *FW:* and the body of the message includes not only the current message provided by the user, but also the original message the user wants to forward to another recipient.

[Fig.12-20]

Grammar Rules

According to the VoiceXML 2.0 Specification, platforms should support the Augmented BNF (ABNF) Form of the W3C SRGS, although VoiceXML platforms may choose to support grammar formats other than SRGS.

The <*grammar*> element may specify an **inline** grammar or an **external** grammar. Fig. 12.21 demostrates an example of inline grammar. This simple example provides inline grammar rules for the selection of one of many items.

[Fig.12-21]

In a similar manner, VoiceXML allows developers to provide DTMF grammar rules.

```
<grammar mode="dtmf" weight="0.3"
src="http://javaschool.com/school/public/speech/vxml/dtmf.number"/>
```

The grammar above includes references to the *dtmf* grammar file. The extract below shows inline *dtmf* grammar rules.

```
<grammar mode="dtmf" version="1.0" root="code">
  <rule id="root" scope="public">
    <one-of>
      <item> 1 2 3 </item>
      <item> # </item>
    </one-of>
  </rule>
</grammar>
```

The VoiceXML Interpreter Evaluates Its Own Performance

The *application.lastresult$* variable holds information about the last recognition. The *application.lastresult$[i].confidence* may vary from 0.0 to 1.0. A value of 0.0 indicates minimum confidence. The *application.lastresult$[i].utterance* keeps the raw string of words (or digits for DTMF) that were recognized for this interpretation. The *application.lastresult$[i].inputmode* stores the last mode value (DTMF or voice). The *application.lastresult$[i].interpretation* variable contains the last interpretation result. This self-evaluation feature can be used to provide additional confirmational prompts when necessary.

```
<if cond="application.lastresult$.confidence &lt; 0.7">
              <goto nextitem="confirmationdialog"/>
```

```
         <else/>
```

VoiceXML Interpreter's Resources and Caching

A VoiceXML interpreter fetches VoiceXML documents and other resources, such as audio files, grammars, scripts, and objects, using powerful caching mechanisms. Unlike a visual browser, a VoiceXML interpreter lacks end-user controls for cache refresh, which is controlled only through appropriate use of the *maxage* and *maxstale* attributes in VoiceXML documents.

The *maxage* attribute indicates that the document is willing to use content whose age is no greater than the specified time in seconds. If the *maxstale* attribute 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.

Metadata in VoiceXML Documents

VoiceXML does not require metadata information. However, it provides two elements in which metadata information can be expressed: <*meta*> and <*metadata*>, with the recommendation that metadata is expressed using the <*metadata*> element, with information in Resource Description Framework (RDF).

As in HTML, the <*meta*> element can contain a metadata property of the document expressed by the pair of attributes, *name* and *content*.

```
<meta name="generator" content="http://JavaSchool.com"/>
```

The <*meta*> element can also specify HTTP response headers with *http-equiv* and *content* attributes.

```
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1"
/>
```

A VoiceXML document can include the <*metadata*> element using the Dublin Core version 1.0 RDF Schema [15].

Fig. 12.22 provides an example of a VoiceXML document with the <*metadata*> element. The <*metadata*> element provides hidden (and silent) information about the document, which nonetheless serves (or will serve) an extremely important role in the interconnected world. This information feeds search engines and helps end users find the document.

[Fig.12-22]

The metadata element ends our voyage into VoiceXML technology, and also ends this chapter.

Summary

This chapter reviewed voice technologies, speech synthesis and recognition, related standards, and some implementations.

VoiceXML-based technology is the most mature, and is prime-time ready for what it was designed for: telephony applications that offer menu-driven voice dialogs that eventually lead to services.

Data communication is growing, and wireless devices will begin to exchange more data packets outside than inside the telephony world. At that point, the lightness and multimodality of SALT will make it a stronger competitor.

Neither of these technologies was designed for a natural language user interface. One common limitation is the grammar rules standard defined by the SRGS. They fit perfectly into the multiple-choice world, but have no room for the thoughtful process of understanding.

Integrating Questions

What are the common features of speech application architectures?

What role does XML play in speech applications?

Case Study

1. Create a SALT file, similar to Fig. 12.11, that is related to a book order.

2. Create a grammar file to support the ordering of a book.

3. Describe an application at your workplace that can benefit from speech technology.

References

1. The Java Speech API: *http://java.sun.com/products/java-media/speech*.
2. The Java Speech API Markup Language: *http://java.sun.com/products/java-media/speech/forDevelopers/JSML*.
3. Speech Synthesis Markup Language: *http://www.w3.org/TR/speech-synthesis/*.
4. The Java Speech Grammar Format Specification: *http://java.sun.com/products/java-media/speech/forDevelopers/JSGF/*.
5. Sphinx, open source speech recognition project: *http://sourceforge.net/projects/cmusphinx/*.
6. Microsoft Speech SDK: *http://download.microsoft.com/download/speechSDK/*.
7. Nyffenegger, R. “A C++ Socket Class for Windows.“ Available at: *http://www.adp-gmbh.ch/win/misc/sockets.html.* Accessed Dec. 20, 2002.
8. “Speech Application Language Tags,” Technical White Paper, SALT Forum, Jan. 20, 2002. Available at: *http://www.saltforum.org/spec.asp*.

9. VoiceXML: *www.voicexml.org/spec.html*.
10. Speech Recognition Grammar Standard: *http://www.w3.org/TR/speech-grammar*.
11. Natural Language Semantics Markup Language: *http://www.w3.org/TR/nl-spec/*.
12. Call Control XML: *http://www.w3.org/TR/ccxml/*.
13. Standard ECMA-262 ECMAScript Language Specification: *http://www.ecma-international.org/*.
14. Ungar, D., and R. Smith. 1987. “Self: The Power of Simplicity.” In Proceedings of OOPSLA Conference, Orlando, FL, October 1987: pp. 227–241.
15. "Dublin Core Metadata Initiative," a Simple Content Description Model for Electronic Resources: *http://purl.org/DC/*.