Dialog planning in VoiceXML

Csapó Tamás Gábor <csapot AT tmit.bme.hu>

4 January 2011

2. VoiceXML Programming Guide

VoiceXML is an XML format programming language, describing the interactions between human and computer, with voice-driven dialogues. Its purpose is to include the advantages of the web-based interfaces and content delivery into interactive voice driven applications, to be able to develop without special programming skills. It can be primarily used in telephone environment, using voice browsers [1, 2]. During the measurement you can get acquainted with version 2.0 of VoiceXML. The following Programming Guide is quoted from [2] http://cafe.bevocal.com/docs/tutorial/tutorial.pdf, extended with specific things that can be used in the VoiceXML server used at BME TMIT.

2.1 A typical VoiceXML application

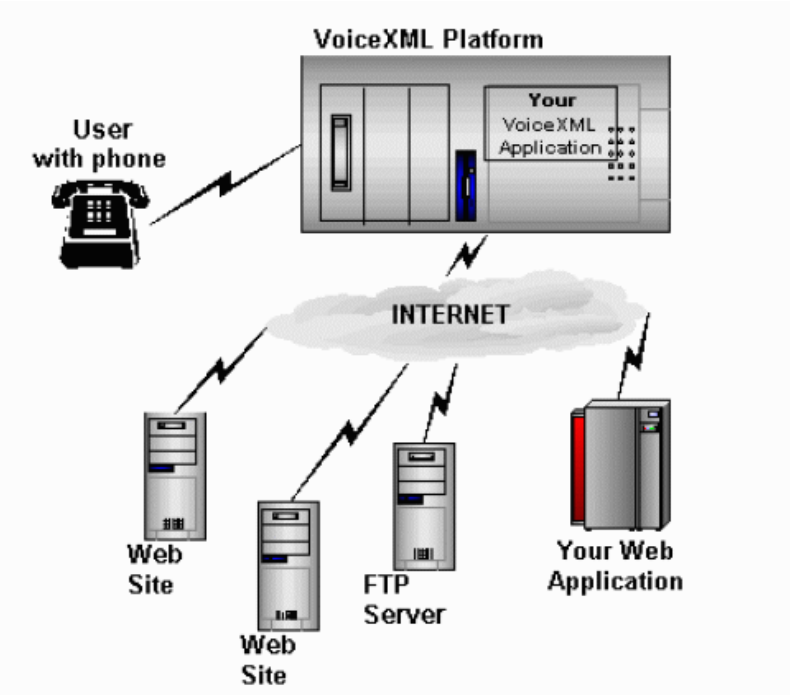

Fig 6. This diagram shows a typical VoiceXML application at work.

A user connects with your application by dialing the appropriate phone number. The VoiceXML interpreter answers the call and starts executing your VoiceXML document. Under the document's control, the interpreter may perform actions such as:

• Sending vocal prompts, messages, or other audio material (such as music or sound effects) to the user.

• Accepting numeric input that the user enters by DTMF (telephone key tone) signals.
• Accepting voice input and recognizing the words.
• Accepting voice input and simply recording it, without trying to recognize any words.
• Sending the user's information to a web site or other Internet server.
• Receiving information from the Internet and passing it to the user.
In addition, VoiceXML documents can perform programming functions such as arithmetic and text manipulation. This allows a document to check the validity of the user's input. Also, a user's session need not be a simple sequence that runs the same way every time. The document may include "if-then-else" decision making and other complex structures.

2.1.1 Voice access to the Web

A complete VoiceXML application usually requires some resources outside the server hosting the VoiceXML application. You may need a web site or other server that is able to accept and process information from users and to send reply information back to them.
This is an advantage of using VoiceXML. A VoiceXML document can access the World Wide Web, acting as a sort of voice-controlled browser. It can send information to web servers and convey the reply to the user. Also, by using VoiceXML as a "front end" to a web application, you minimize the amount of VoiceXML coding that is required. The rest of your application can be based on familiar protocols, such as HTTP and CGI, for which powerful programming tools are available.

2.1.2 Application development

Because VoiceXML documents consist of plain text, you can create and edit them with any simple text editor such as Notepad or Notepad++ on Windows, or EMACS on UNIX systems. You place the documents on your web server, along with related files such as audio data. You test them by calling the appropriate phone number and interacting with them.
Several tools can help you develop your applications:

- VoiceXML interpreter and syntax checker,
- recording dialogs in a SIP client,
- log file, which is generated for each call.

2.2 Basics of VoiceXML

At this point, let's look at a simple VoiceXML document (the line numbers are not part of the document):

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <vxml version="2.0" xmlns="http://www.w3.org/2001/vxml">
3     <form>
4           <field name="selection">
5                 <prompt>
6                       Please choose News, Weather or Sports.
7                 </prompt>
8                 <grammar root="r" mode="voice"
                        type="application/srgs+xml">
9                       <rule id="r">
10                            <one-of>
11                                  <item>news</item>
```

```
12                                  <item>weather</item>
13                                  <item>sports</item>
14                              </one-of>
15                          </rule>
16                  </grammar>
17          </field>
18          <block>
19                  <submit next="process.vxml"/>
20          </block>
21    </form>
22 </vxml>
```

Code: a simple VoiceXML application

This document causes the request, "Please choose News, Weather, or Sports," to be spoken to the user. Then it accepts the user's response and passes the response to another document—actually a server-side script named „`process.vxml`" —which presumably provides the service that the user selected.

2.2.1 Basic syntax

This document illustrates a number of basic points about VoiceXML. It consists of plain text as well as *tags*, which are keywords or expressions enclosed by the angle bracket (< and >) characters. In this example, everything is a tag except for lines 6 and 11-13.
A tag may have *attributes* inside the angle brackets. Each attribute consists of a *name* and a *value*, separated by an equal (=) sign; the value must be enclosed in quotes. Examples are the `version` attribute in line 2 and the `name` attribute in line 4.
Most of the tags in this example are used in pairs. The `<vxml>` tag is matched by `</vxml>`; tags such as `<form>` and `<block>` have similar counterparts. Tags that are used in pairs are called *containers*; they contain content, such as text or other tags, between the paired tags.
Some tags are used singly, rather than in pairs; these are called *stand-alone* or *empty* tags. Empty tags do not have any content (text or other tags), although they may have attributes. The only one in this example is the `<submit... />` tag. Notice that in an empty tag, there is a slash (/) just before the closing > character.
Caution: Compared to HTML, VoiceXML is much stricter about using correct syntax. If you have used HTML, you may be used to taking shortcuts, such as writing attribute values without quotes or omitting the ending tag of a container. In VoiceXML, you must use proper syntax in all documents.
The term *element* is sometimes used in talking about languages such as HTML and VoiceXML. An element is either:
• A stand-alone tag, including its attributes, if any; or
• A container tag, including the start tag and its attributes, as well as the matching end tag, and any other text or tags contained between them.
If an element contains another element, it may be referred to as the *parent* element of the one it contains. Conversely, a contained element is said to be a *child* element of its container.

2.2.2 Header

Lines 1 and 2 are required at the beginning of all VoiceXML documents. For now, always use the lines just as shown above. These lines work with standard VoiceXML (as described in this tutorial). The last line (22) is required at the end of all documents; it matches the `<vxml>` tag in line 2.

2.2.3 Main body

After the header information comes the main body of the VoiceXML document. Typically, the body contains at least one `<form>` tag. The `<form>` tag defines a VoiceXML form, which is a section of a document that can interact with the user. This is the VoiceXML equivalent of an HTML form. For instance, you could write HTML to display a form in a web page:

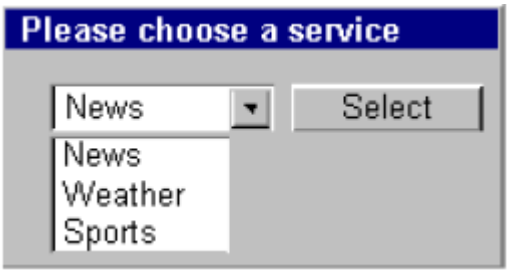

Fig 7. HTML form

In VoiceXML, you use `<form>`, `<field>`, and `<submit>` in a very similar structure. A form usually contains *fields* which accept pieces of information from the user. In this simple example, the form has a single field named selection (lines 4-18) that stores the user's choice.
Inside the field is a `<prompt>` element (lines 5-7), which contains a line of plain text. The VoiceXML interpreter will use its text-to-speech (TTS) translation engine to read this text and send the resulting audio to the user. As we'll see later, instead of relying only on the TTS engine, you can also use `<audio>` tags to play prerecorded audio files.
After the `<prompt>` is a `<grammar>` element (lines 8-16) that defines the three responses the interpreter will accept for this field. With `<rule>` you can give a grammar rule, which means here the enumeration of `<item>` elements within `<one-of>` tag. With more complex forms of the `<grammar>` tag you can give complex grammars, which can be stored in separate files. Grammars usually contain spoken words or DTMF codes, the application can handle both.
After the interpreter sends the prompt to the user, it waits for the user to respond and tries to match the response to the grammar. If the response matches, the interpreter sets the field's selection variable to the response. This value is then available throughout the rest of the form.
Finally, the form ends with a `<block>` element containing a `<submit>` tag (lines 18-20). These lines transfer control to `process.vxml`, passing the `selection` variable for use by that script.

2.3 A basic dialog

Previously, we looked at the fundamentals of VoiceXML as they applied to a very simple script with only one field. In this chapter, we take a closer look at interacting with the user—how to make a voice-response application that is powerful and easy to use.

2.3.1 Example application

Here we'll look at something more like a real-world application. We'll introduce a larger document that accepts several inputs from the user, and also provides some error handling. This document will function as a voice-operated calculator.
The user names an operation (add, subtract, multiply, or divide) and then says two numbers; the script replies with the result. As with the first example, it may be helpful to compare this to an HTML form on a web page, which might look about like this:

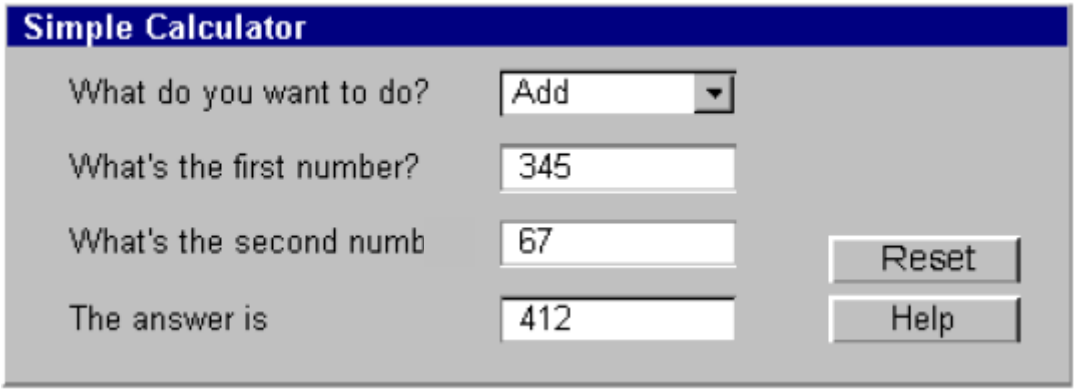

Fig 8. Simple calculator with HTML form

A VoiceXML equivalent to this could be made with a `<form>` containing three `<field>` elements. But to make it easy to use, we'll add some additional features that function similarly to the Reset and Help buttons.
This document is longer than our first example; we'll examine it one part at a time.

2.3.2 Initialization and debugging features

Our calculator starts with the usual standard elements:

```
<?xml version="1.0" encoding="UTF-8"?>
<vxml version="2.0" xmlns="http://www.w3.org/2001/vxml">
```

Comments

VoiceXML comments use the form you are familiar from HTML. Comments start with <!-- and end with -->. Anything between these delimiters is ignored by the interpreter.

```
<!-- Start of calculator code -->
```

Help and error handling

As the last part of our document's initialization, we'll include some tags, called *event handlers* that provide help and error handling for user input. First:

```
<noinput>
      I'm sorry. I didn't hear anything.
      <reprompt/>
</noinput>
```

<noinput> specifies what the document should do if the user fails to provide any voice or keypad input. In this case, the tag specifies a text message and uses the <reprompt> tag to cause the interpreter to repeat the <prompt> for the current form field.

```
<nomatch>
      I didn't get that.
      <reprompt/>
</nomatch>
```

<nomatch> is similar to <noinput>; it specifies what to do if the user's input doesn't match any active grammar.

```
<help>
      I'm sorry. There's no help available here.
</help>
```

If the user asks for help, the message between the <help> tags is read. In this case, the message is not very helpful! But this is the default help message for the entire document.

2.3.3 Forms and fields

We start with the <form> tag, followed by a <block> containing an opening message:

```
<form>
      <block>
            This is the Sample calculator.
      </block>
```

In general, the <block> tag is used to contain a section of executable content that does not expect to interact with the user. For example, you can use it, as here, to play a message that doesn't expect a response. Or, you can use it to contain some program logic, such as the <submit> tag that ended the previous example.
When a <block> tag contains simply text, as in this example, the interpreter treats it as shorthand for having included a <prompt> tag, for example:

```
<block>
      <prompt>
            This is the Sample calculator.
      </prompt>
</block>
```

2.3.4 Basic field format

Now, let's write the first field, where the user chooses the type of calculation:

```
      <!-- operation input -->
      <field name="op">
            <prompt>
                  Choose an operation: add or subtract.
            </prompt>
            <grammar root="r" mode="voice" type="application/srgs+xml">
```

```
            <rule id="r">
                  <one-of>
                        <item>add</item>
                        <item>subtract</item>
                  </one-of>
            </rule>
      </grammar>
      <help>
            Please say what you want to do.
            <reprompt/>
      </help>
      <filled>
            <prompt>
                  Okay, let's <value expr="op"/> two numbers.
            </prompt>
      </filled>
</field>
```

This field is a bit more complex than the one in our first example. It shows the primary parts of a field—prompts to play, grammars to use for recognizing a user's utterances, and actions to perform on successful recognition.

This field has the name "op", which can be used in VoiceXML expressions to reference the field's *input* value, that is, what the user selects. The field contains a `<prompt>` tag to tell the user what's expected and a `<grammar>` tag to listen for the user to say the operation he wants to perform. The grammar, a set of four words enclosed in square brackets, specifies that any one of these words is an acceptable value for this field.

Next is a <help> tag; this provides a specific message that the user will hear if he requests help for this field. Part of the help message (supplied by the `<reprompt>` tag) repeats the field's prompt. You can also add `<noinput>` and `<nomatch>` tags for each individual field, if that helps to make your script more user-friendly.

The last part of the field is a `<filled>` tag. This specifies the action to take once the field is "filled in"; that is, once the user has provided a valid input. In this case, the document plays a message that confirms the user's choice. The `<value>` tag is used to include the input value in the message by referencing the field name, op.

Let's go on to the next field, which will hold the first number for the calculation:

```
<!-- first number input -->
<field name="a" type="number">
      <prompt>
            What's the first number?
      </prompt>
      <help>
            Please say a number.
            This number will be used as the first operand.
      </help>
      <filled>
            <prompt>
                  The first number: <value expr="a"/>.
            </prompt>
      </filled>
</field>
```

This field has the name "a". It also has a type attribute that defines it as a numeric field. This is a useful shortcut—it means that we do not need to specify a grammar for this field.

The field contains <prompt>, <help>, and <filled> tags that act like the ones described for the op field. Notice this time the help prompt does not repeat the initial prompt.

2.3.5 Variables

Next, we need a field to accept the second number for the calculation. The <filled> tag in this field is the one that does the calculation and reports the answer to the user. To have the <filled> element do this work, though, we need to include one more tag before the field:

```
<!-- storing the result -->
<var name="result"/>
```

The <var> tag is used to declare a *variable*. VoiceXML variables can hold numbers, text, and several other types of data. As we've seen, every field has an associated variable, based on its name attribute. Because we have declared fields for the numbers to use for the calculation, they already have associated variables. However, since the script will compute the result, there is no field for it; we must explicitly declare a variable to hold the result.

2.3.6 Field computation

With that done, we can write the code for the last field. It's going to be a long one, so let's start with the first few tags:

```
<!-- second number input -->
<field name="b" type="number">
      <prompt>
            And the second number?
      </prompt>
      <help>
            Please say a number.
            This number will be used as the second operand.
      </help>
      <filled>
            <prompt>
                  <value expr="b"/> Okay.
            </prompt>
```

This field is named b, and its type is number, like the previous field. It has a <help> message, and a <filled> tag that confirms the user's entry. After that, things will get more complex. We need to write code that actually chooses the type of operation to perform, does the calculation, and reports the result. The <filled> element is going to need some more content.

```
            <!-- calculating the result depending on the operand -->
            <if cond="op=='add'">
                  <assign name="result" expr="Number(a) + Number(b)"/>
                  <prompt>
                        <value expr="a"/> plus <value expr="b"/>
                        equals <value expr="result"/>
                  </prompt>
```

The <if> tag is used for decision-making. Its cond attribute specifies the condition to be tested. If the condition is true, the code following the <if> will be executed. If the condition is false, the interpreter will skip over the following text and tags, until it encounters an <else>, <elseif>, or closing </if> tag. The cond value, op=='add', is an expression that is true if the op field contains the word add.
If the condition is true, the interpreter will execute the <assign> tag, which computes the expression Number(a) + Number(b) and places the sum of the two numbers in the result variable (specified by the name attribute). You might have been tempted to write the expression as a + b; however, the + symbol can indicate string concatenation as well as addition. Using the Number() functions guarantees the arguments are numbers and not strings. If a user tries to add 30 and 14, the result should be 44, not 3014!
The <assign> is followed by a <prompt> to report the result to the user. Again, the <value> tag is used to include the values of both input numbers, as well as the result value, in the prompt.

That takes care of addition. Next, we need another tag to handle subtraction:

```
<elseif cond="op=='subtract'"/>
      <assign name="result" expr="a - b"/>
      <prompt>
            <value expr="a"/> minus <value expr="b"/>
            equals <value expr="result"/>
      </prompt>
</if>
```

These tags are very similar to the ones for addition. Instead of <if>, we started with <elseif> because we are building a series of choices, only one of which can be executed each time the script is run. In the <assign>, we can write a – b without using Number(), because the - symbol only has one possible meaning; there is no need to clarify it.

Here we have used <elseif> instead of <if>. After the prompt is the </if> tag that closes the entire conditional sequence.

2.3.7 Form reset

At this point, we have just about finished building our dialog; there are just a few "housekeeping" items to take care of:

```
<!-- form reset -->
<clear/>
```

The <clear> tag is used to reset the form to its initial state; that is, it clears the values of all the fields in the form. When appropriate, you can also use the <clear> tag to reset only some of the variables and fields. Resetting the fields causes execution to resume at the top of the form, allowing the user to make another calculation.
Finally, we have tags to close the <filled> element, the field that contains it, the entire form, and the VoiceXML document:

```
            </filled>
        </field>
    </form>
</vxml>
```

2.3.8 Summary

This chapter has introduced the following main points:
• Use comments to document and maintain your script.
• Use <help>, <noinput>, and <nomatch> to guide your users.
• Use <filled> fields to take action when the user provides input to them.
• Declare variables with <var>, perform calculations with <assign>, and place results into prompts
with <value>.
• Make decisions with <if>, <elseif>, and <else/>.
• Reset a form with <clear/>.

2.4 VoiceXML tags

Based on [4], http://www.vxml.org/

<assign> The assign element is used to explicitly assign a value to a variable.
<block> The block element is simply a form-item container element for executable content, which executes if the condition of the item is equal to 'true'.
<break> The break element is used to designate a pause in the TTS output, with the length being a user-specified time value in either milliseconds, or of a predetermined 'size' variable.
<choice> The choice element is used in conjunction with the menu element to create robust voice menus that allow the caller multiple navigational choices without the need for traditional grammar structures. The menu tag acts as the container element, while the choice element defines the available menu items.
<clear> The clear element is used to set any existing VoiceXML variable or element guard variables to an undefined value, such as any user-defined variable set with the var or assign tags.
<disconnect> The disconnect element is used to programmatically disconnect the caller from the voice application.
<else> The else element is used as the final logic constructor in an array of conditional statements.
<elseif> The elseif element is used to specify additional content when all other else or if statements equate to 'false'.
<field> The field element facilitates a dialog which allows the interpreter to collect information from the user.
<filled> The filled element allows the developer to specify actions to take when a grammar match has occurred.
<form> The form element acts as a container for all field-items, (such as a field or a subdialog), and for all control items, (such as a block or an initial element).
<goto> The goto tag is used to transition application execution to a specific form within the current document, or to an entirely separate document.
<grammar> The grammar element allows the developer to specify a GSL or XML recognition grammar for either voice or DTMF input.
<help> The help element provides a syntactic shorthand for <catch event="help">. This element is used as an error handler for the help event.
<if> The if element, (in conjunction with the else/elseif elements), provides a method to utilize conditional logic expressions which allow the developer to

change the control flow within the application based on user utterances, variable values, or events.

`<item>` The item element defines a valid utterance match within an XML grammar.

`<menu>` The menu tag is another 'shortcut element' that emulates the field, grammar, and goto elements.

`<noinput>` The noinput tag acts as a syntactic shorthand for the expression <catch event="noinput">. It allows the developer to assign event handlers when the application expects voice or DTMF input, but has received none from the caller.

`<nomatch>` The nomatch element is a syntactic shorthand for the expression <catch event="nomatch">. This attribute allows the developer to assign handlers when the caller inputs a value that is not recognized by any of the active grammars.

`<one-of>` The one-of element allows the grammar to be constructed with a series of alternate phrases or rule expansions, each of which is contained within an Item element. This element is the basic building block of any XML grammar that defines more than one valid utterance.

`<prompt>` The prompt element allows the developer to output synthesized text-to-speech content to the caller.

`<reprompt>` The reprompt element, upon execution, will increase the FIA prompt counter by one, and then replay the most recent prompt before listening again for caller input.

`<rule>` The rule element defines the named rule expansion of an XML grammar.

`<script>` The script element is used for enclosing ECMAScript / JavaScript code to execute on the client side.

`<submit>` The use of the submit tag will transition control to a new document, either via the GET or POST methods.

`<var>` The var element is used to declare a VoiceXML variable within the scope specified by its parent element.

`<vxml>` The vxml element is the initial declaration that defines a document as a VoiceXML application.

2.5 References

[1] Chetan Sharma & Jeff Kunins, VoiceXML: Strategies and Techniques for Effective Voice Application Development with VoiceXML 2.0, Wiley 2002

[2] BeVocal, Inc., „VoiceXML Tutorial", 2005, http://cafe.bevocal.com/docs/tutorial/tutorial.pdf

[3] Voice Extensible Markup Language (VoiceXML) Version 2.0, http://www.w3.org/TR/voicexml20/

[4] VoiceXML Development Guide, Version 2.1, http://www.vxml.org/

Further example programs can be found on the further links:

http://cafe.bevocal.com/resources/voicexml_samples/index.html

https://studio.tellme.com/library2/code/

http://developer.voicegenie.com/examples_VoiceGenie.php