VoiceXML FORUM

Voice eXtensible Markup Language

VoiceXML

Version: 1.00
Date: 07 March 2000

About the VoiceXML Forum

The VoiceXML Forum is an industry organization founded by AT&T, IBM, Lucent and Motorola. It was established to develop and promote the Voice eXtensible Markup Language (VoiceXML), a new computer language designed to make Internet content and information accessible via voice and phone.

With the backing and technology contributions of its four world-class founders, and the support of leading Internet industry players, the VoiceXML Forum has made speech-enabled applications on the Internet a reality.

For more information on the VoiceXML Forum please visit the website at http://www.voicexml.org

Implementation Scope

VoiceXML 1.0 was designed for speech-based telephony applications. Where a specific application environment does not require all the features of v1.0, exceptions should be clearly noted, and publicly documented as a subset of VoiceXML 1.0. Any vendor-specific additions or changes should be similarly noted as proprietary extensions to VoiceXML 1.0. The VoiceXML Forum provides no support for, and make no guarantee of, future compatibility with such changes.

Disclaimers

Revision History

Version	Date	Description
0.9	17 Aug 1999	Initial release. Provided as baseline in support of comment period from supporters.
1.0 RC	02 Mar 2000	Release Candidate (released to Forum Supporters)
1.0	07 Mar 2000	Released to public - editorial corrections from 1.0 RC

Table of Contents

1 INTRODUCTION

This document introduces VoiceXML, the Voice Extensible Markup Language. VoiceXML is designed for creating audio dialogs that feature synthesized speech, digitized audio, recognition of spoken and DTMF key input, recording of spoken input, telephony, and mixed-initiative conversations. Its major goal is to bring the advantages of web-based development and content delivery to interactive voice response applications.

Here are two short examples of VoiceXML. The first is the venerable "Hello World":

```
<?xml version="1.0"?>
<vxml version="1.0">
  <form>
    <block>Hello World!</block>
  </form>
</vxml>
```

The top-level element is `<vxml>`, which is mainly a container for *dialogs*. There are two types of dialogs: *forms* and *menus*. Forms present information and gather input; menus offer choices of what to do next. This example has a single form, which contains a block that synthesizes and presents "Hello World!" to the user. Since the form does not specify a successor dialog, the conversation ends.

Our second example asks the user for a choice of drink and then submits it to a server script:

```
<?xml version="1.0"?>
<vxml version="1.0">
  <form>
    <field name="drink">
      <prompt>Would you like coffee, tea, milk, or nothing?</prompt>
      <grammar src="drink.gram" type="application/x-jsgf"/>
    </field>
    <block>
      <submit next="http://www.drink.example/drink2.asp"/>
    </block>
  </form>
</vxml>
```

A *field* is an input field. The user must provide a value for the field before proceeding to the next element in the form. A sample interaction is:

C *(computer)*: Would you like coffee, tea, milk, or nothing?

H *(human)*: Orange juice.

C: I did not understand what you said.

C: Would you like coffee, tea, milk, or nothing?

H: Tea

C: *(continues in document drink2.asp)*

2 BACKGROUND

This section contains a high-level architectural model, whose terminology is then used to describe the goals of VoiceXML, its scope, its design principals, and the requirements it places on the systems that support it.

2.1 Architectural Model

The architectural model assumed by this document has the following components:

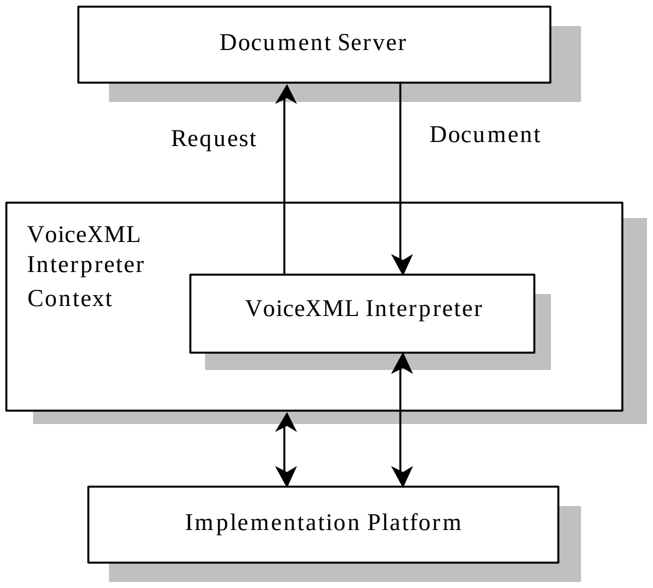

Figure 1 Architectural model.

A *document server* (e.g. a web server) processes *requests* from a client application, *the VoiceXML Interpreter*, through the *VoiceXML interpreter context*. The server produces *VoiceXML documents* in reply, which are processed by the VoiceXML Interpreter. The VoiceXML interpreter context may monitor user inputs in parallel with the VoiceXML interpreter. For example, one VoiceXML interpreter context may always listen for a special escape phrase that takes the user to a high-level personal assistant, and another may listen for escape phrases that alter user preferences like volume or text-to-speech characteristics.

The *implementation platform* is controlled by the VoiceXML interpreter context and by the VoiceXML interpreter. For instance, in an interactive voice response application, the VoiceXML interpreter context may be responsible for detecting an incoming call, acquiring the initial VoiceXML document, and answering the call, while the VoiceXML interpreter conducts the dialog after answer. The implementation platform generates events in response to user actions (e.g. spoken or character input received, disconnect) and system events (e.g. timer expiration). Some of these events are acted upon by the VoiceXML interpreter itself, as specified by the VoiceXML document, while others are acted upon by the VoiceXML interpreter context.

2.2 Goals of VoiceXML

VoiceXML's main goal is to bring the full power of web development and content delivery to voice response applications, and to free the authors of such applications from low-level programming and resource management. It enables integration of voice services with data services using the familiar client-server paradigm. A voice service is viewed as a sequence of interaction dialogs between a user and an implementation platform. The dialogs are provided by document servers, which may be external to the implementation platform. Document servers maintain overall service logic, perform database and legacy system operations, and produce dialogs. A VoiceXML document specifies each interaction dialog to be conducted by a VoiceXML interpreter. User input affects dialog interpretation and is collected into requests submitted to a document server. The document server may reply with another VoiceXML document to continue the user's session with other dialogs.

VoiceXML is a markup language that:

- Minimizes client/server interactions by specifying multiple interactions per document.
- Shields application authors from low-level, and platform-specific details.
- Separates user interaction code (in VoiceXML) from service logic (CGI scripts).
- Promotes service portability across implementation platforms. VoiceXML is a common language for content providers, tool providers, and platform providers.
- Is easy to use for simple interactions, and yet provides language features to support complex dialogs.

While VoiceXML strives to accommodate the requirements of a majority of voice response services, services with stringent requirements may best be served by dedicated applications that employ a finer level of control.

2.3 Scope of VoiceXML

The language describes the human-machine interaction provided by voice response systems, which includes:

- Output of synthesized speech (text-to-speech).
- Output of audio files.
- Recognition of spoken input.
- Recognition of DTMF input.
- Recording of spoken input.
- Telephony features such as call transfer and disconnect.

The language provides means for collecting character and/or spoken input, assigning the input to document-defined request variables, and making decisions that affect the interpretation of documents written in the language. A document may be linked to other documents through Universal Resource Identifiers (URIs).

2.4 Principles of Design

VoiceXML is an XML schema. For details about XML, refer to the Annotated XML Reference Manual.

1. The language promotes portability of services through abstraction of platform resources.
2. The language accommodates platform diversity in supported audio file formats, speech grammar formats, and URI schemes. While platforms will respond to market pressures and support common formats, the language per se will not specify them.
3. The language supports ease of authoring for common types of interactions.
4. The language has a well-defined semantics that preserves the author's intent regarding the behavior of interactions with the user. Client heuristics are not required to determine document element interpretation.
5. The language has a control flow mechanism.
6. The language enables a separation of service logic from interaction behavior.
7. It is not intended for heavy computation, database operations, or legacy system operations. These are assumed to be handled by resources outside the document interpreter, e.g. a document server.
8. General service logic, state management, dialog generation, and dialog sequencing are assumed to reside outside the document interpreter.
9. The language provides ways to link documents using URIs, and also to submit data to server scripts using URIs.
10. VoiceXML provides ways to identify exactly which data to submit to the server, and which HTTP method (get or post) to use in the submittal.
11. The language does not require document authors to explicitly allocate and deallocate dialog resources, or deal with concurrency. Resource allocation and concurrent threads of control are to be handled by the implementation platform.

2.5 Implementation Platform Requirements

This section outlines the requirements on the hardware/software platforms that will support a VoiceXML interpreter.

Document acquisition. The interpreter context is expected to acquire documents for the VoiceXML interpreter to act on. In some cases, the document request is generated by the interpretation of a VoiceXML document, while other requests are generated by the interpreter context in response to events outside the scope of the language, for example an incoming phone call.

Audio output. An implementation platform can provide audio output using audio files and/or using text-to-speech (TTS). When both are supported, the platform must be able to freely sequence TTS and audio output. Audio files are referred to by a URI. The language does not specify a required set of audio file formats.

Audio input. An implementation platform is required to detect and report character and / or spoken input simultaneously and to control input detection interval duration with a timer whose length is specified by a VoiceXML document.

- It must report characters (for example, DTMF) entered by a user.
- It must be able to receive speech recognition grammar data dynamically. Some VoiceXML elements contain speech grammar data; others refer to speech grammar data through a URI. The speech recognizer must be able to accommodate dynamic update of the spoken input for which it is listening through either method of speech grammar data specification.
- It should be able to record audio received from the user. The implementation platform must be able to make the recording available to a *request* variable.

3 CONCEPTS

A VoiceXML *document* (or a set of documents called an *application*) forms a conversational finite state machine. The user is always in one conversational state, or *dialog*, at a time. Each dialog determines the next dialog to transition to. *Transitions* are specified using URIs, which define the next document and dialog to use. If a URI does not refer to a document, the current document is assumed. If it does not refer to a dialog, the first dialog in the document is assumed. Execution is terminated when a dialog does not specify a successor, or if it has an element that explicitly exits the conversation.

3.1 Dialogs and Subdialogs

There are two kinds of dialogs: *forms* and *menus*. Forms define an interaction that collects values for a set of field item variables. Each field may specify a grammar that defines the allowable inputs for that field. If a form-level grammar is present, it can be used to fill several fields from one utterance. A menu presents the user with a choice of options and then transitions to another dialog based on that choice.

A *subdialog* is like a function call, in that it provides a mechanism for invoking a new interaction, and returning to the original form. Local data, grammars, and state information are saved and are available upon returning to the calling document. Subdialogs can be used, for example, to create a confirmation sequence that may require a database query; to create a set of components that may be shared among documents in a single application; or to create a reusable library of dialogs shared among many applications.

3.2 Sessions

A *session* begins when the user starts to interact with a VoiceXML interpreter context, continues as documents are loaded and processed, and ends when requested by the user, a document, or the interpreter context.

3.3 Applications

An *application* is a set of documents sharing the same *application root document*. Whenever the user interacts with a document in an application, its application root document is also loaded. The application root document remains loaded while the user is transitioning between other documents in the same application, and it is unloaded when the user transitions to a document that is not in the application. While it is loaded, the application root document's variables are available to the other documents as *application variables*, and its grammars can also be set to remain active for the duration of the application.

Figure 2 shows the transition of documents (D) in an application that share a common application root document (root).

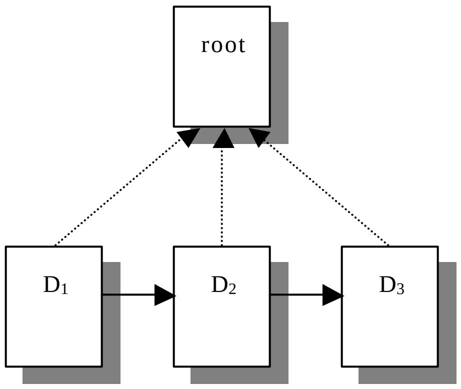

Figure 2 Transitioning between documents in an application.

3.4 Grammars

Each dialog has one or more speech and/or DTMF *grammars* associated with it. In *machine directed* applications, each dialog's grammars are active only when the user is in that dialog. In *mixed initiative* applications, where the user and the machine alternate in determining what to do next, some of the dialogs are flagged to make their grammars *active* (i.e., listened for) even when the user is in another dialog in the same document, or on another loaded document in the same application. In this situation, if the user says something matching another dialog's active grammars, execution transitions to that other dialog, with the user's utterance treated as if it were said in that dialog. Mixed initiative adds flexibility and power to voice applications.

3.5 Events

VoiceXML provides a form-filling mechanism for handling "normal" user input. In addition, VoiceXML defines a mechanism for handling events not covered by the form mechanism.

Events are thrown by the platform under a variety of circumstances, such as when the user does not respond, doesn't respond intelligibly, requests help, etc. The interpreter also throws events if it finds a semantic error in a VoiceXML document. Events are caught by catch elements or their syntactic shorthand. Each element in which an event can occur may specify catch elements. Catch elements are also inherited from enclosing elements "as if by copy". In this way, common event handling behavior can be specified at any level, and it applies to all lower levels.

3.6 Links

A *link* supports mixed initiative. It specifies a grammar that is active whenever the user is in the scope of the link. If user input matches the link's grammar, control transfers to the link's destination URI. A <link> can be used to throw an event to go to a destination URI.

4 VOICEXML ELEMENTS

Element	Purpose	Page
<assign>	Assign a variable a value.	71
<audio>	Play an audio clip within a prompt.	46
<block>	A container of (non-interactive) executable code.	54
<break>	JSML element to insert a pause in output.	44
<catch>	Catch an event.	38
<choice>	Define a menu item.	28
<clear>	Clear one or more form item variables.	72
<disconnect>	Disconnect a session.	76
<div>	JSML element to classify a region of text as a particular type.	44
<dtmf>	Specify a touch-tone key grammar.	35
<else>	Used in <if> elements.	72
<elseif>	Used in <if> elements.	72
<emp>	JSML element to change the emphasis of speech output.	44
<enumerate>	Shorthand for enumerating the choices in a menu.	28
<error>	Catch an error event.	39
<exit>	Exit a session.	75
<field>	Declares an input field in a form.	50
<filled>	An action executed when fields are filled.	64
<form>	A dialog for presenting information and collecting data.	17
<goto>	Go to another dialog in the same or different document.	73
<grammar>	Specify a speech recognition grammar.	35
<help>	Catch a help event.	39
<if>	Simple conditional logic.	72
<initial>	Declares initial logic upon entry into a (mixed-initiative) form.	55
<link>	Specify a transition common to all dialogs in the link's scope.	30
<menu>	A dialog for choosing amongst alternative destinations.	28
<meta>	Define a meta data item as a name/value pair.	66
<noinput>	Catch a noinput event.	39
<nomatch>	Catch a nomatch event.	39
<object>	Interact with a custom extension.	60
<option>	Specify an option in a <field>	53
<param>	Parameter in <object> or <subdialog>.	69
<prompt>	Queue TTS and audio output to the user.	44
<property>	Control implementation platform settings.	66
<pros>	JSML element to change the prosody of speech output.	44
<record>	Record an audio sample.	61
<reprompt>	Play a field prompt when a field is re-visited after an event.	73
<return>	Return from a subdialog.	75
<sayas>	JSML element to modify how a word or phrase is spoken.	44
<script>	Specify a block of ECMAScript client-side scripting logic.	77
<subdialog>	Invoke another dialog as a subdialog of the current one.	56

Element	Purpose	Page
`<submit>`	Submit values to a document server.	74
`<throw>`	Throw an event.	38
`<transfer>`	Transfer the caller to another destination.	63
`<value>`	Insert the value of a expression in a prompt.	46
`<var>`	Declare a variable.	71
`<vxml>`	Top-level element in each VoiceXML document.	14

5 DOCUMENT STRUCTURE AND EXECUTION

A VoiceXML document is primarily composed of top-level elements called *dialogs*. There are two types of dialogs: *forms* and *menus*. A document may also have `<meta>` elements, `<var>` and `<script>` elements, `<property>` elements, `<catch>` elements, and `<link>` elements.

Execution within one document. Document execution begins at the first dialog by default. As each dialog executes, it determines the next dialog. When a dialog doesn't specify a successor dialog, document execution stops.

Here is "Hello World!" expanded to illustrate some of this. It now has a document level variable called "hi" which holds the greeting. Its value is used as the prompt in the first form. Once the first form plays the greeting, it goes to the form named "say_goodbye", which prompts the user with "Goodbye!" Because the second form does not transition to another dialog, it causes the document to be exited.

```
<?xml version="1.0"?>
<vxml version="1.0">
  <meta name="author" content="John Doe"/>
  <meta name="maintainer" content="hello-support@hi.example"/>
  <var name="hi" expr="'Hello World!'"/>
  <form>
    <block>
      <value expr="hi"/>
      <goto next="#say_goodbye"/>
    </block>
  </form>
  <form id="say_goodbye">
    <block>
      Goodbye!
    </block>
  </form>
</vxml>
```

Stylistically it is best to combine the forms:

```
<?xml version="1.0"?>
<vxml version="1.0">
  <meta name="author" content="John Doe"/>
  <meta name="maintainer" content="hello-support@hi.example"/>
  <var name="hi" expr="'Hello World!'"/>
  <form>
    <block><value expr="hi"/>Goodbye!</block>
  </form>
</vxml>
```

Attributes of `<vxml>` include:

`version`	The version of VoiceXML of this document (required). The initial version number is 1.0.
`base`	The base URI.
`lang`	The language and locale type for this document.
`application`	The URI of this document's application root document, if any.

Executing a multi-document application. Normally, each document runs as an isolated application. In cases where you want multiple documents to work together as one application, you select one document to be the *application root document*, and refer to it in the other documents' `<vxml>` elements.

When this is done, every time the interpreter is told to load a document in this application, it also loads the application root document if it is not already loaded. The application root document remains loaded until the interpreter is told to load a document that belongs to a different application. Thus one of the following two conditions always holds during interpretation:

- The application root document (or a stand-alone document) is loaded and the user is executing in it.
- The application root document and one other document in the application are both loaded and the user is executing in the non-root document.

There are two benefits to multi-document applications. First, the application root document's variables are available for use by the other documents in the application, so that information can be shared and retained. Second, the grammars of the application root document may be set to remain active even when the user is in other application documents, so that the user can always interact with common forms, links, and menus.

Here is a two-document application illustrating this:

<u>Application root document</u> (app-root.vxml)

```
<?xml version="1.0"?>
<vxml version="1.0">
  <var name="bye" expr="'Ciao'"/>
  <link next="operator_xfer.vxml"> <grammar> operator </grammar> </link>
</vxml>
```

<u>Main document</u> (main.vxml)

```
<?xml version="1.0"?>
<vxml version="1.0" application="app-root.vxml">
  <form id="say_goodbye">
    <field name="answer" type="boolean">
      <prompt>Shall we say <value expr="application.bye"/>?</prompt>
      <filled>
        <if cond="answer">
          <exit/>
        </if>
        <clear namelist="answer"/>
      </filled>
    </field>
  </form>
</vxml>
```

In this example, the application is designed so that main.vxml must be loaded first. Its application attribute specifies that app-root.vxml should be imported as the application root document. So, app-root.vxml is then loaded, which creates the application variable bye and also defines a link that navigates to /operator-xfer.vxml whenever the user says "operator". The user starts out in the say_goodbye form:

C: Shall we say Ciao?

H: Si.

C: I did not understand what you said. *(a platform-specific default message.)*

H: Ciao

C: I did not understand what you said.

H: Operator.

C: *(Goes to operator_xfer.vxml, which transfers the caller to a human operator.)*

Note that when the user is in a multi-document application, at most two documents are loaded at any one time: the application root document, and unless the user is actually interacting with the application root document, one other application document.

If a document refers to a non-existent application root document, or if an application root document itself has a reference to another application root document, an error.semantic event is thrown.

Subdialogs. A subdialog is a mechanism for decomposing complex sequences of dialogs to better structure them, or to create reusable components. For example, the solicitation of account information may involve gathering several pieces of information, such as account number, and home telephone number. A customer care service might be structured with several independent applications that could share this basic building block, thus it would be reasonable to construct it as a subdialog. This is illustrated in the example below. The first document, app.vxml, seeks to adjust a customer's account, and in doing so must get the account information and then the adjustment level. The account information is obtained by using a subdialog element that invokes another VoiceXML document to solicit the user input. While the second document is being executed, the calling dialog is suspended, awaiting the return of information. The second document provides the results of its user interactions using a <return> element, and the resulting values are accessed through the variable defined by the name attribute on the <subdialog> element.

Customer Service Application (app.vxml)

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

<form id="billing_adjustment">
   <var name="account_number"/>
   <var name="home_phone"/>

   <subdialog name="accountinfo" src="acct_info.vxml#basic">
      <filled>
         <!-- Note the variable defined by "accountinfo" is returned as
              a an ECMAScript object and it contains two properties defined
              by the variables specified in the "return" element of the
              subdialog. -->
         <assign name="account_number" expr="accountinfo.acctnum"/>
         <assign name="home_phone"     expr="accountinfo.acctphone"/>
      </filled>
   </subdialog>

   <field name="adjustment_amount" type="currency">
      <prompt> What is the value of your account adjustment?</prompt>
      <filled>
         <submit  next="/cgi-bin/updateaccount"/>
      </filled>
   </field>
</form>

</vxml>
```

Document Containing Account Information Subdialog (acct_info.vxml)

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

   <form id="basic">
      <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>
            <!-- The values obtained by the two fields are supplied to the
               calling dialog by the "return" element. -->
            <return namelist="acctnum acctphone"/>
         </filled>
      </field>
   </form>

   </vxml>
```

Subdialogs add a new execution context when they are invoked. The subdialog could be a new dialog within the existing document, or a new dialog within a new document. The invocation of a subdialog limits the scope of active grammars to the subdialog only.

Figure 3 shows the execution flow when a sequence of documents (D) transitions to a subdialog (SD) and then back.

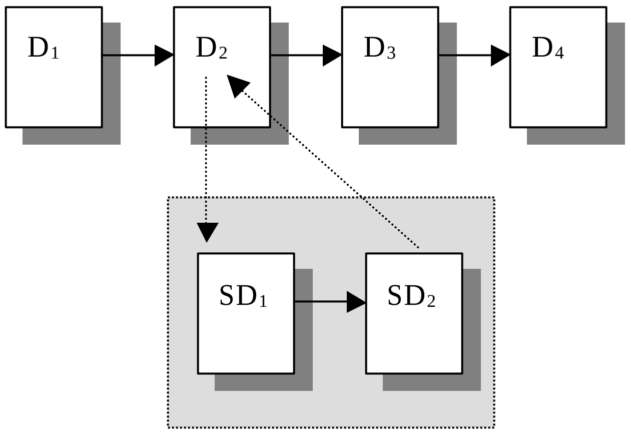

Figure 3 Execution flow when invoking a subdialog composed of two documents.

6 FORMS

Forms are the key component of VoiceXML documents. A form contains:

- A set of *form items*, elements that are visited in the main loop of the form interpretation algorithm. Form items are subdivided into *field items*, those that define the form's field item variables, and *control items*, those that help control the gathering of the form's fields.
- Declarations of non-field item variables.
- Event handlers.
- "Filled" actions, blocks of procedural logic that execute when certain combinations of field items are filled in.

Form attributes are:

`id`	The name of the form.
`scope`	The default scope of the form's grammars. If it is `dialog` then the form grammars are active only in the form. If the scope is `document`, then the form are active during any dialog in the same document. If the scope is `document` and the document is an application root document, then the form grammars are active during any dialog in any document of this application. A form grammar that has `dialog` scope is active only in its form.

This section describes some of the concepts behind forms, and then gives some detailed examples of their operation.

6.1 Form Interpretation

Forms are interpreted by an implicit form interpretation algorithm (FIA). The FIA has a main loop that repeatedly selects a form item and then visits it. The selected form item is the lexically first whose guard condition is not satisfied. For instance, a field item's default guard condition tests to see if the field item variable has a value, so that if a simple form contains only field items, the user will be prompted for each field item in turn.

Interpreting a form item generally involves:

- Selecting and playing one or more prompts;
- Collecting a user input, either a response that fills in one or more fields, or a throwing of some event (`help`, for instance); and
- Interpreting any `<filled>` actions that pertained to the newly filled in fields.

The FIA ends when it interprets a transfer of control statement (e.g. a `<goto>` to another dialog or document, a `<submit>` of data to the document server). It also ends with an implied `<exit>` when no form item remains eligible to select.

6.2 Form Items

A form's form items are the elements that can be visited in the main loop of the form interpretation algorithm. Field items direct the FIA to gather a specific field. When the FIA selects a control item, the control item may contain a block of procedural code to execute, or it may tell the FIA to set up the initial prompt-and-collect for a mixed initiative form.

6.2.1 Field Items

A field item specifies a *field item variable* to gather from the user. Field items have prompts to tell the user what to say or key in, grammars that define the allowed inputs, and event handlers that process any resulting events. A field item may also have a `<filled>` element that defines an action to take just after the field item variable is filled in. Field items are subdivided into:

`<field>`	A field item whose value is obtained via ASR or DTMF grammars.
`<record>`	A field item whose value is an audio clip recorded by the user. A `<record>` element could collect a voice mail message, for instance.
`<transfer>`	A field item which transfers the user to another telephone number. If the transfer returns control, the field variable will be set to the result status.
`<object>`	This field item invokes a platform-specific "object" with various parameters. The result of the platform object is an ECMAScript Object with one or more properties. One platform object could be a built-in dialog that gathers credit card information. Another could gather a text message using some proprietary DTMF text entry method. There is no requirement for implementations to provide platform-specific objects, although support for the `<object>` element is required.
`<subdialog>`	A `<subdialog>` field item is roughly like a function call. It invokes another dialog on the current page, or invokes another VoiceXML document. It returns an ECMAScript `Object` as its result.

6.2.2 Control Items

There are two types of control items:

`<block>`	A sequence of procedural statements used for prompting and computation, but not for gathering input. A block has a (normally implicit) form item variable that is set to `true` just before it is interpreted.
`<initial>`	This element controls the initial interaction in a mixed initiative form. Its prompts should be written to encourage the user to say something matching a form level grammar. When at least one field item variable is filled as a result of recognition during an `<initial>` element, the form item variable of `<initial>` becomes true, thus removing it as an alternative for the FIA.

6.3 Form Item Variables and Conditions

Each form item has an associated *form item variable*, which by default is set to `undefined` when the form is entered. This form item variable will contain the result of interpreting the form item. A field item's form item variable is also called a *field item variable*, and it holds the value collected from the user. A form item variable can be given a name using the name attribute, or left nameless, in which case an internal name is generated.

Each form item also has a *guard condition*, which governs whether or not that form item can be selected by the form interpretation algorithm. The default guard condition just tests to see if the form item variable has a value. If it does, the form item will not be visited.

Typically, field items are given names, but control items are not. Generally form item variables are not given initial values and additional guard conditions are not specified. But sometimes there is a need for more detailed control. One form may have a form item variable initially set to hide a field, and later cleared (e.g., using <clear>) to force the field's collection. Another field may have a guard condition that activates it only when it has not been collected, and when two other fields have been filled. A block item could execute only when some condition holds true. Thus, fine control can be exercised over the order in which form items are selected and executed by the FIA, however in general, many dialogs can be constructed without resorting to this level of complexity.

In summary, all form items have the following attributes:

name	The name of a dialog-scoped form item variable that will hold the value of the form item.
expr	The initial value of the form item variable; default is ECMAScript undefined. If initialized to a value, then the form item will not be executed unless the form item variable is cleared.
cond	An expression to evaluate in conjunction with the test of the form item variable. If absent, this defaults to true, or in the case of <initial>, a test to see if any field item variable has been filled in.

6.4 Directed Forms

The simplest and most common type of form is one in which the form items are executed exactly once in sequential order to implement a computer-directed interaction. Here is a weather information service that uses such a form.

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

```
</form>
```

This dialog proceeds sequentially:

> C (computer): Welcome to the weather information service. What state?
>
> H (human): Help
>
> C: Please speak the state for which you want the weather.
>
> H: Georgia
>
> C: What city?
>
> H: Tblisi
>
> C: I did not understand what you said. What city?
>
> H: Macon
>
> C: The conditions in Macon Georgia are sunny and clear at 11 AM ...

The form interpretation algorithm's first iteration selects the first block, since its (hidden) form item variable is initially `undefined`. This block outputs the main prompt, and its form item variable is set to `true`. On the FIA's second iteration, the first block is skipped because its form item variable is now defined, and the `state` field is selected because the dialog variable `state` is `undefined`. This field prompts the user for the state, and then sets the variable `state` to the answer. The third form iteration prompts and collects the `city` field. The fourth iteration executes the final block and transitions to a different URI.

Each field in this example has a prompt to play in order to elicit a response, a grammar that specifies what to listen for, and an event handler for the `help` event. The `help` event is thrown whenever the user asks for assistance. The help event handler catches these events and plays a more detailed prompt.

Here is a second directed form, one that prompts for credit card information:

```
<form id="get_card_info">
  <block> We now need your credit card type, number, and expiration date.</block>

  <field name="card_type">
    <prompt count="1">What kind of credit card do you have?</prompt>
    <prompt count="2">Type of card?</prompt>
    <!-- This is an in line grammar. -->
    <grammar>
        visa                {visa}
      | master [card]       {mastercard}
      | amex                {amex}
      | american [express]  {amex}
    </grammar>
    <help> Please say Visa, Mastercard, or American Express. </help>
  </field>

  <!-- The grammar for type="digits" is built in. -->
  <field name="card_num" type="digits">
    <prompt count="1">What is your card number?</prompt>
    <prompt count="2">Card number?</prompt>
    <catch event="help">
      <if cond="card_type == 'amex'">
        Please say or key in your 15 digit card number.
      <else/>
        Please say or key in your 16 digit card number.
      </if>
    </catch>
    <filled>
```

```
      <if cond="card_type == 'amex' && card_num.length != 15">
          American Express card numbers must have 15 digits.
          <clear namelist="card_num"/>
          <throw event="nomatch"/>
      <elseif cond="card_type != 'amex' && card_num.length != 16"/>
          Mastercard and Visa card numbers have 16 digits.
          <clear namelist="card_num"/>
          <throw event="nomatch"/>
      </if>
    </filled>
  </field>

  <field name="expiry_date" type="digits">
    <prompt count="1">What is your card's expiration date?</prompt>
    <prompt count="2">Expiration date?</prompt>
    <help>
      Say or key in the expiration date, for example one two oh one.
    </help>
    <filled>
      <!-- validate the mmyy -->
      <var name="mm"/>
      <var name="i" expr="expiry_date.length"/>
      <if cond="i == 3">
          <assign name="mm" expr="expiry_date.substring(0,1)"/>
      <elseif cond="i == 4"/>
          <assign name="mm" expr="expiry_date.substring(0,2)"/>
      </if>
      <if cond="mm == '' || mm < 1 || mm > 12">
          <clear namelist="expiry_date"/>
          <throw event="nomatch"/>
      </if>
    </filled>
  </field>

  <field name="confirm" type="boolean">
    <prompt>I have <value expr="card_type"/> number <value expr="card_num"/>,
        expiring on <value expr="expiry_date"/>.  Is this correct? </prompt>
    <filled>
        <if cond="confirm">
            <submit next="place_order.asp"
              namelist="card_type card_num expiry_date"/>
        </if>
        <clear namelist="card_type card_num expiry_date acknowledge"/>
    </filled>
  </field>
</form>
```

The dialog might go something like this:

C: We now need your credit card type, number, and expiration date.	
C: What kind of credit card do you have?	
H: Discover	
C: I did not understand what you said.	*(a platform-specific default message.)*
C: Type of card?	*(the second prompt is used now.)*
H: Shoot.	*(fortunately treated as "help" by this platform)*
C: Please say Visa, Master card, or American Express.	
H: Uh, Amex.	*(this platform ignores "uh")*
C: What is your card number?	

H: One two three four ... wait ...

C: I did not understand what you said.

C: Card number?

H: *(uses DTMF)* 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 #

C: What is your card's expiration date?

H: one two oh one

C: I have Amex number 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 expiring on 1 2 0 1. To go on say yes, to reenter say no.

H: Yes

Fields are the major building blocks of forms. A field declares a variable and specifies the prompts, grammars, DTMF sequences, help messages, and other event handlers that are used to obtain it. Each field declares a VoiceXML field item variable in the form's `dialog` scope. These may be submitted once the form is filled, or copied into other variables.

Each field has its own speech and/or DTMF grammars, specified explicitly using `<grammar>` and `<dtmf>` elements, or implicitly using the `type` attribute. The type attribute is used for standard built-in grammars, like `digits`, `boolean`, or `number`. The type attribute also governs how that field's value is spoken by the speech synthesizer.

Each field can have one or more prompts. If there is one, it is repeatedly used to prompt the user for the value until one is provided. If there are many, they must be given `count` attributes. These determine which prompt to use on each attempt. In the example, prompts are become shorter. This is called *tapered prompting*.

The `<catch event="help">` elements are event handlers that define what to do when the user asks for help. Help messages can also be tapered. These can be abbreviated, so that the following two elements are equivalent:

```
<catch event="help">
  Please say visa, mastercard, or amex.
</catch>

<help>Please say visa, mastercard, or amex.</help>
```

The `<filled>` element defines what to do when the user provides a recognized input for that field. One use is to specify integrity constraints over and above the checking done by the grammars, as with the date field above.

6.5 Mixed Initiative Forms

The last section talked about forms implementing rigid, computer-directed conversations. To make a form *mixed initiative*, where both the computer and the human direct the conversation, it must one or more `<initial>` form items and one or more form-level grammars.

If a *form* has form-level grammars:

- Its fields can be filled in any order.
- More than one field can be filled as a result of a single user utterance.

Also, the form's grammars can be active when the user is in other dialogs. If a document has two forms on it, say a car rental form and a hotel reservation form, and both forms have grammars that are active for that document, a user could respond to a request for hotel reservation information with information about the car rental, and thus direct the computer to talk about the car rental instead. The user can speak to any active grammar, and have fields set and actions taken in response.

Example. Here is a second version of the weather information service, showing mixed initiative. It has been “enhanced” for illustrative purposes with advertising and with a confirmation of the city and state:

```
<form id="weather_info">
  <grammar src="cityandstate.gram" type="application/x-jsgf"/>

  <!-- Caller can't barge in on today's advertisement. -->
  <block>
    <prompt bargein="false">
      Welcome to the weather information service.
      <audio src="http://www.online-ads.example/wis.wav"/>
    </prompt>
  </block>

  <initial name="start">
    <prompt> For what city and state would you like the weather? </prompt>
    <help> Please say the name of the city and state for which you
           you would like a weather report. </help>
    <!-- If user is silent, reprompt once, then try directed prompts. -->
    <noinput count="1"> <reprompt/> </noinput>
    <noinput count="2"> <reprompt/> <assign name="start" expr="true"/> </noinput>
  </initial>

  <field name="state">
    <prompt>What state?</prompt>
    <help>Please speak the state for which you want the weather.</help>
  </field>

  <field name="city">
    <prompt>Please say the city in <value expr="state"/> for which
            you want the weather.</prompt>
    <help>Please speak the city for which you want the weather.</help>
    <filled>
      <!-- Most of our customers are in LA. -->
      <if cond="city == 'Los Angeles' && state == undefined">
        <assign name="state" expr="'California'"/>
      </if>
    </filled>
  </field>

  <field name="go_ahead" type="boolean" modal="true">
    <prompt>Do you want to hear the weather for
         <value expr="city"/>, <value expr="state"/>?
    </prompt>
    <filled>
      <if cond="go_ahead">
        <prompt bargein="false">
          <audio src="http://www.online-ads.example/wis2.wav"/>
        </prompt>
        <submit next="/servlet/weather" namelist="city state"/>
      </if>
      <clear namelist="start city state go_ahead"/>
    </filled>
  </field>
</form>
```

Here is a transcript showing the advantages for even a novice user:

> C: Welcome to the weather information service. Buy Joe's Spicy Shrimp Sauce.
>
> C: For what city and state would you like the weather?
>
> H: Uh, California.
>
> C: Please say the city in California for which you want the weather.
>
> H: San Francisco, please.
>
> C: Do you want to hear the weather for San Francisco, California?
>
> H: No
>
> C: What state?
>
> H: Los Angeles.
>
> C: Do you want to hear the weather for Los Angeles, California?
>
> H: Yes
>
> C: Don't forget, buy Joe's Spicy Shrimp Sauce tonight!
>
> C: Mostly sunny today with highs in the 80s. Lows tonight from the low 60s ...

The `go_ahead` field has its `modal` attribute set to `true`. This causes all grammars to be disabled except the ones defined in the current form item, so that the only grammar active during this field is the built-in grammar for `boolean`.

An experienced user can get things done much faster (but is still forced to listen to the ads):

> C: Welcome to the weather information service. Buy Joe's Spicy Shrimp Sauce.
>
> C: What ...
>
> H *(barging in)*: LA
>
> C: Do you ...
>
> H *(barging in)*: Yes
>
> C: Don't forget, buy Joe's Spicy Shrimp Sauce tonight!
>
> C: Mostly sunny today with highs in the 80s. Lows tonight from the low 60s ...

Controlling the order of field collection. The form interpretation algorithm can be customized in several ways. One way is to assign a value to a form item variable, so that its form item will not be selected. Another is to use `<clear>` to set a form item variable to `undefined`; this forces the FIA to revisit the form item again.

Another method is to explicitly specify the next field item to visit using `<goto nextitem>`. This forces an immediate transfer to that field item. If the `<goto nextitem>` occurs in a `<filled>` action, the rest of the `<filled>` action and any pending `<filled>` actions will be skipped.

Here is an example `<goto nextitem>` executed in response to the `exit` event:

```
<form id="survey_2000_03_30">
   <catch event="exit">
      <goto nextitem="confirm_exit"/>
   </catch>
   <block>
      <prompt>
         Hello, you have been called at random to answer questions
         critical to U.S. foreign policy.
      </prompt>
   </block>
```

```
      <field name="q1" type="boolean">
         <prompt>Do you agree with the IMF position on privatizing certain
                 functions of Burkina Faso’s agriculture ministry?</prompt>
      </field>
      <field name="q2" type="boolean">
         <prompt>If this privatization occurs, will its effects be beneficial
                 mainly to Ouagadougou and Bobo-Dioulasso?</prompt>
      </field>
      <field name="q3" type="boolean">
         <prompt>Do you agree that sorghum and millet output might thereby
                increase by as much as four percent per annum?</prompt>
      </field>
      <block>
         <submit next="register" namelist="q1 q2 q3"/>
      </block>
      <field name="confirm_exit" type="boolean">
         <prompt>You have elected to exit.  Are you sure you want to do
                 this, and perhaps adversely affect U.S. foreign policy
                 vis-à-vis sub-Saharan Africa for decades to come?</prompt>
         <filled>
            <if cond="confirm_exit">
               Okay, but the U.S. State Department is displeased.
               <exit/>
            <else/>
               Good, let’s pick up where we left off.
               <clear namelist="confirm_exit"/>
            </if>
         </filled>
      </field>
   </form>
```

If the user says “exit” in response to any of the survey questions, an `exit` event is thrown by the platform and caught by the `<catch>` event handler. This handler directs that `confirm_exit` be the next visited field. The `confirm_exit` field would not be visited during normal completion of the survey because the preceding `<block>` element transfers control to the registration script.

6.6 Form Interpretation Algorithm

We’ve presented the form interpretation algorithm (FIA) at a conceptual level. In this section we describe it in more detail.

6.6.1 Initialization Phase

Whenever a form is entered, it is initialized. Internal prompt counter variables (in the form’s `dialog` scope) are reset to 1. Each variable (form-level `<var>` elements and form item variables) is initialized, in document order, to `undefined` or to the value of the relevant `expr` attribute.

6.6.2 Main Loop

The main loop of the FIA has three phases:

- The *select* phase: the next form item is selected for visiting.
- The *collect* phase: the next unfilled form item is visited, which prompts the user for input, enables the appropriate grammars, and then waits for and collects an *input* (such as a spoken phrase or DTMF key presses) or an *event* (such as a request for help or a no input timeout).

- The *process* phase: an input is processed by filling form items and executing `<filled>` elements to perform actions such as input validation. An event is processed by executing the appropriate event handler for that event type.

Note that the FIA may be given an input (a set of grammar slot/slot value pairs) that was collected while the user was in a different form's FIA. In this case the first iteration of the main loop skips the select and collect phases, and goes right to the process phase with that input.

Select phase. The purpose of the select the next form item to visit. This is done as follows:

- If a `<goto>` from the last main loop iteration's process phase specified a `<goto nextitem>`, then the specified form item is selected.
- Otherwise the first form item whose guard condition is false is chosen to be visited.
- If no guard condition is false, then the last iteration completed the form without encountering an explicit transfer of control, so the FIA does an implicit `<exit>` operation.

Collect phase. The purpose of the collect phase is to collect an input or an event. The selected form item is *visited*, which performs actions that depend on the type of form item:

- If a field item is visited, the FIA selects and queues up any prompts based on the field item's prompt counter and the prompt conditions. Then it listens for the field level grammar(s) and any active higher-level grammars, and waits for a grammar recognition or for some event.
- If an `<initial>` is visited, the FIA selects and queues up prompts based on the `<initial>`'s prompt counter and prompt conditions. Then it listens for the form level grammar(s) and any active higher-level grammars. It waits for a grammar recognition or for an event.
- A `<block>` element is visited by setting its form item variable to `true`, evaluating its content, and then bypassing the process phase. No input is collected, and the next iteration of the FIA's main loop is entered.

Process phase. The purpose of the process phase is to process the input or event collected during the collect phase, as follows:

- If an event (such as a `noinput` or a `hangup`) occurred, then the applicable catch element is identified and executed. This can cause the FIA to terminate (e.g. if it transitions to a different dialog or document or it does an `<exit>`), or it can cause the FIA to go into the next iteration of the main loop (e.g. as when the default help event handler is executed).
- If an input matches a grammar from a `<link>` then that link's transition is executed, or its event is thrown. If the `<link>` throws an event, the event is processed in the context of the current form item.
- If an input matches a grammar in a form other than the current form, then the FIA terminates, the other form is initialized, and that form's FIA is started with this input in its process phase.
- If an input matches a grammar in this form, then:
    - The input's grammar slot values are assigned to the corresponding field item variables.
    - The `<filled>` actions triggered by these assignments are identified as described in section 15.

  - Each identified <filled> action is executed in document order. If a <goto> or <throw> is encountered, the remaining <filled> elements are not executed, and the FIA either terminates or continues in the next main loop iteration.

After completion of the process phase, interpretation continues by returning to the select phase.

A more detailed form interpretation algorithm can be found in Appendix C.

7 MENUS

A *menu* is a convenient syntactic shorthand for a form containing a single anonymous field that prompts the user to make a choice and transitions to different places based on that choice. Like a regular form, it can have its grammar scoped such that it is active when the user is executing another dialog. The following menu offers the user three choices:

```
<menu>
  <prompt>Welcome home. Say one of: <enumerate/></prompt>
  <choice next="http://www.sports.example/vxml/start.vxml">
     Sports </choice>
  <choice next="http://www.weather.example/intro.vxml">
     Weather </choice>
  <choice next="http://www.stargazer.example/voice/astronews.vxml">
     Stargazer astrophysics news </choice>
  <noinput>Please say one of <enumerate/></noinput>
</menu>
```

This dialog might proceed as follows:

C: Welcome home. Say one of: sports; weather; Stargazer astrophysics news.

H: Astrology.

C: I did not understand what you said. *(a platform-specific default message.)*

C: Welcome home. Say one of: sports; weather; Stargazer astrophysics news.

H: sports.

C: *(proceeds to http://www.sports.example/vxml/start.vxml)*

Menu element. This identifies the menu, and determines the scope of its grammars. Menu attributes are:

id	The identifier of the menu. It allows the menu to be the target of a <goto> or a <submit>.
scope	The menu's grammar scope. If it is dialog – the default – the menu's grammars are only active when the user transitions into the menu. If the scope is document, its grammars are active over the whole document (or if the menu is in the application root document, any loaded document in the application).
dtmf	When set to true, any choices that do not have explicit DTMF elements are given the implicit ones "1", "2", etc.

Choice element. The <choice> element serves several purposes:

- It specifies a speech grammar fragment and/or a DTMF grammar fragment that determines when that choice has been selected.

- The contents are used to form the `<enumerate>` prompt string.
- It specifies the URI to go to when the choice is selected.

Choice attributes are:

`dtmf`	The DTMF sequence for this choice.
`next`	The URI of next dialog or document.
`event`	Specify an event to be thrown instead of specifying a `next`.
`expr`	Specify an expression to evaluate instead of specifying a `next`.
`caching`	See Section 12.1.
`fetchaudio`	See Section 12.1.
`fetchhint`	See Section 12.1. This defaults to the `documentfetchhint` property.
`fetchtimeout`	See Section 12.1.

DTMF in menus. Menus can rely purely on speech, purely on DTMF, or both in combination by including a `<property>` element in the `<menu>`. Here is a DTMF-only menu with explicit DTMF sequences given to each choice, using the choice's `dtmf` attribute:

```
<menu>
  <property name="inputmodes" value="dtmf"/>
  <prompt>
    For sports press 1, For weather press 2, For Stargazer astrophysics press 3.
  </prompt>
  <choice dtmf="1" next="http://www.sports.example/vxml/start.vxml"/>
  <choice dtmf="2" next="http://www.weather.example/intro.vxml"/>
  <choice dtmf="3" next="http://www.stargazer.example/voice/astronews.vxml"/>
</menu>
```

Alternatively, you can set the `<menu>`'s `dtmf` attribute to `true` to assign sequential DTMF digits to each of the first nine choices: the first choice has DTMF "1", and so on:

```
<menu dtmf="true">
  <property name="inputmodes" value="dtmf"/>
  <prompt>
    For sports press 1, For weather press 2, For Stargazer astrophysics press 3.
  </prompt>
  <choice next="http://www.sports.example/vxml/start.vxml"/>
  <choice next="http://www.weather.example/intro.vxml"/>
  <choice next="http://www.stargazer.example/voice/astronews.vxml"/>
</menu>
```

Enumerate element. The `<enumerate>` element is an automatically generated description of the choices available to the user. It specifies a template that is applied to each choice in the order they appear in the menu. If it is used with no content, a default template that lists all the choices is used, determined by the interpreter context. If it has content, the content is the template specifier. This specifier may refer to two special variables: `_prompt` is the choice's prompt, and `_dtmf` is the choice's assigned DTMF sequence. For example, if the menu were rewritten as

```
<menu dtmf="true">
  <prompt>
    Welcome home.
    <enumerate>
      For <value expr="_prompt"/>, press <value expr="_dtmf"/>.
    </enumerate>
  </prompt>
  <choice next="http://www.sports.example/vxml/start.vxml">
```

```
      sports </choice>
    <choice next="http://www.weather.example/intro.vxml">
      weather </choice>
    <choice next="http://www.stargazer.example/voice/astronews.vxml">
      Stargazer astrophysics news </choice>
  </menu>
```

then the menu’s prompt would be:

C: Welcome home. For sports, press 1. For weather, press 2. For Stargazer astrophysics news, press 3.

The <enumerate> element may also be used analogously in prompts for <field> elements that contain a set of <option> elements as discussed in Section 14.1.3

Grammar Generation. Any *choice phrase* specifies a set of words and phrases to listen for. The user may say any phrase consisting of any subset of the words of the choice phrase in the same order in which they occur in the choice phrase. A choice phrase is constructed from the PCDATA of the elements contained directly or indirectly in the <choice> element. For example, in response to the prompt “Stargazer astrophysics news” a user could say “Stargazer”, “astrophysics”, “Stargazer news”, “astrophysics news”, and so on. The equivalent JSGF rule would be “[Stargazer] [astrophysics] [news]” (where [...] indicates optionality).

As an example of the use of PCDATA contained in descendants of the <choice> element, consider the following example:

```
    <choice next="http://www.stargazer.example/voice/astronews.vxml">
      <prompt>
        <audio src="http://www.stargazer.example/space.wav">
          Stargazer
          <emp>astrophysics</emp>
          news
        </audio>
      </prompt>
    </choice>
```

This choice would be read from the audio file, or as “Stargazer *Astrophysics* News” if the file could not be played. The grammar for the choice would be the equivalent of “[Stargazer] [astrophysics] [news] ” gleaned from the PCDATA of the <choice> element’s descendants.

Interpretation model. A menu behaves like a form with a single field that does all the work. The menu prompts become field prompts. The menu event handlers become the field event handlers. The menu grammars become *form* grammars.

Upon entry, the menu’s grammars are built and enabled, and the prompt is played. When the user input matches a choice, control transitions according to the value of the next, expr, or event attribute of the <choice>, only one of which may be specified.

8 LINKS

A <link> element has one or more grammars, which are scoped to the element containing the <link>. Grammar elements contained in the <link> are not permitted to specify scope. When one of these grammars is matched, the link activates, and either:

- Transitions to a new document or dialog (like <goto>), or
- Throws an event (like <throw>).

For instance, this link activates when you say “books” or press “2”.

```
<link next="http://www.voicexml.org/books/main.vxml">
  <grammar type="application/x-jsgf"> books | Voice XML books </grammar>
  <dtmf> 2 </dtmf>
</link>
```

This link takes you to a dynamically determined dialog in the current document:

```
<link expr="'#' + document.helpstate">
  <grammar type="application/x-jsgf"> help </grammar>
</link>
```

The `<link>` element can be a child of `<vxml>`, `<form>`, or of a form item. A link at the `<vxml>` level has grammars that are active throughout the document. A link at the `<form>` level has grammars active while the user is in that form. If an application root document has a document-level link, its grammars are active no matter what document of the application is being executed.

If execution is in a modal form item, then link grammars at the form or document level are not active.

You can also define a link that, when matched, throws an event instead of going to a new document. This event is thrown at the current location in the execution, not at the location where the link is specified. For example, if the user matches this link’s grammar, a `help` event is thrown in the form item the user was visiting:

```
<link event="help">
  <grammar type="application/x-jsgf">
    arrgh | alas all is lost | fie ye froward machine | I don't get it
  </grammar>
</link>
```

Attributes of `<link>` are:

`next`	The URI to go to. This URI is a document (perhaps with an anchor to specify the starting dialog), or a dialog in the current document (just a bare anchor).
`expr`	Like `next`, except that the URI is dynamically determined by evaluating the given ECMAScript expression.
`event`	The event to throw when the user matches one of the link grammars. Note that only one of `next`, `expr`, or `event` may be specified.
`caching`	See Section 12.1.
`fetchaudio`	See Section 12.1.
`fetchhint`	See Section 12.1. This defaults to the `documentfetchhint` property.
`fetchtimeout`	See Section 12.1.

9 VARIABLES AND EXPRESSIONS

VoiceXML variables are in all respects equivalent to ECMAScript variables. The variable naming convention is as in ECMAScript, but names beginning with the underscore character (“_”) are reserved for internal use.

9.1 Declaring Variables

Variables are declared by <var> elements:

```
<var name="home_phone"/>
<var name="pi"      expr="3.14159"/>
<var name="city"    expr="'Sacramento'"/>
```

They are also declared by form items:

```
<field name="num_tickets" type="number">
   <prompt>How many tickets do you wish to purchase?</prompt>
</field>
```

Variables declared without an explicit initial value are initialized to the ECMAScript `undefined` value. Variables must be declared before being used.

In a form, the variables declared by `<var>` and those declared by form items are initialized when the form is entered. The initializations are guaranteed to take place in document order, so that this, for example, is legal:

```
<form id="test">
   <var name="one" expr="1"/>
   <field name="two" expr="one+1" type="number">
   </field>
   <var name="three" expr="two+1"/>
   <field name="go_on" type="boolean">
      <prompt>Say yes or no to continue</prompt>
   </field>
   <filled>
      <goto next="#tally"/>
   </filled>
</form>
```

When the user visits this `<form>`, the form's initialization first declares the variable `one` and sets its value to 1. Then it declares the field item variable `two` and gives it the value 2. Then the initialization logic declares the variable `three` and gives it the value 3. The form interpretation algorithm then enters its main interpretation loop and begins at the `go_on` field.

9.2 Variable Scopes

Variables can be declared in following scopes:

`session`	These are read-only variables that pertain to an entire user session. They are declared and set by the interpreter context. New session variables cannot be declared by VoiceXML documents. See Section 9.4.
`application`	These are declared with `<var>` elements that are children of the application root document's `<vxml>` element. They are initialized when the application root document is loaded. They exist while the application root document is loaded, and are visible to the root document and any other loaded application leaf document.
`document`	These variables are declared with `<var>` elements that are children of the document's `<vxml>` element. They are initialized when the document is loaded. They exist while the document is loaded, and are visible only within that document.

`dialog`	Each dialog (`<form>` or `<menu>`) has a dialog scope that exists while the user is visiting that dialog, and which is visible to the element of that dialog. Dialog variables are declared by `<var>` child elements of `<form>`, by `<var>` elements inside executable content (e.g. `<block>` content or catch element content), and by the various form item elements. The child `<var>` elements of `<form>` are initialized when the form is first visited. The `<var>` elements inside executable content are initialized when the executable content is executed. The form item variables are initialized when the form item is collected.
(anonymous)	Each `<block>`, `<filled>`, and catch element defines a new anonymous scope to contain variables declared in that element.

The following diagram shows the scope hierarchy:

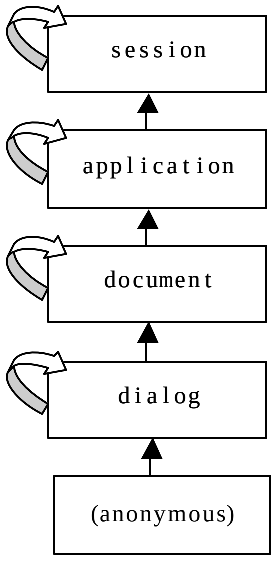

Figure 4 The scope hierarchy.

The curved arrows in this diagram show that each scope contains a variable whose name is the same as the scope that refers to the scope itself. This allows you for example in the anonymous, dialog, and document scopes to refer to a variable *X* in the document scope using `document`.*X*.

9.3 Referencing Variables

Variables are referenced in `cond` and `expr` attributes:

```
<if cond="city == 'LA'">
   <assign name="city" expr="'Los Angeles'"/>
<elseif cond="city == 'Philly'"/>
   <assign name="city" expr="'Philadelphia'"/>
<elseif cond="city == 'Constantinople'"/>
   <assign name="city" expr="'Istanbul'"/>
</if>

<assign name="var1" expr="var1 + 1"/>

<if cond="i > 1">
```

```
      <assign name="i" expr="i-1"/>
   </if>
```

The expression language used in `cond` and `expr` is precisely ECMAScript. Note that the `cond` operators “>”, “<”, “>=”, “<=”, and “&&” must be escaped in XML (to “`&gt;`” and “`&lt;`” and so on). For clarity, examples in this document do not use XML escapes.

Variable references match the closest enclosing scope according to the scope chain given above. You can prefix a reference with a scope name for clarity or to resolve ambiguity. For instance to save the value of a form field item variable for use later on in a document:

```
   <assign name="document.ssn" expr="dialog.ssn"/>
```

If the application root document has a variable `x`, it is referred to as `application.x` in non-root documents, and either `application.x` or `document.x` in the application root document.

9.4 Standard Session Variables

`session.telephone.ani`

Automatic Number Identification. This variable provides the result from the Automatic Number Identification service that provides the receiver of a telephone call with the number of the calling phone. This information is provided only if the service is supported, and is undefined otherwise.

`session.telephone.dnis`

Dialed Number Identification Service. This variable provides the result from the Dialed Number Identification Service that identifies for the receiver of a call the number that the caller dialed. This information is provided only if the service is supported, and is `undefined` otherwise.

`session.telephone.iidigits`

Information Indicator Digit. This variable provides information about the originating line (e.g. payphone, cellular service, special operator handling, prison) of the caller. Telecordia publishes the complete list of II digits in Section 1 of each volume of the "Local Exchange Routing Guide". This information is provided only if the service is supported, and is `undefined` otherwise.

`session.telephone.uui`

User to User Information. This variable returns supplementary information provided as part of an ISDN call set-up from a calling party. This information is provided only if the service is supported, and is `undefined` otherwise.

10 GRAMMARS

10.1 Speech Grammars

The `<grammar>` element is used to provide a speech grammar that

- specifies a set of utterances that a user may speak to perform an action or supply information, and
- provides a corresponding string value (in the case of a field grammar) or set of attribute-value pairs (in the case of a form grammar) to describe the information or action.

The `<grammar>` element is designed to accommodate any grammar format that meets these two requirements. At this time, VoiceXML does not specify a grammar format nor require support of a particular grammar format. This is similar to the situation with recorded audio formats for VoiceXML, and with media formats in general for HTML.

The `<grammar>` element may be used to specify an *inline* grammar or an *external* grammar. An inline grammar is specified by the content of a `<grammar>` element:

```
<grammar type="mime-type">
     inline speech grammar
</grammar>
```

It may be necessary in this case to enclose the content in a CDATA section. For inline grammars the type parameter specifies a MIME type that governs the interpretation of the content of the `<grammar>` tag.

An external grammar is specified by an element of the form

```
<grammar src="URI" type="mime-type"/>
```

The MIME type is optional in this case because this information may be obtained via the URI protocol (as in the case of HTTP), and may be inferred from the filename extension. If the type is not specified, and cannot be inferred, the default type is platform specific. However, if the type is specified using the type attribute, it overrides other information about the type.

See Appendix D for notes on using the Java™ Speech API Grammar Format (JSGF) with VoiceXML. (Note: Java is a trademark of Sun Microsystems Inc.)

Attributes of `<grammar>` include:

src	The URI specifying the location of the grammar, if it is external.
scope	Either document, which makes the grammar active in all dialogs of the current document (and relevant application leaf documents), or dialog, to make the grammar active throughout the current form. If omitted, the grammar scoping is resolved by looking at the parent element.
type	The MIME type of the grammar. If this is omitted, the interpreter context will attempt to determine the type dynamically.
caching	See Section 12.1.
fetchhint	See Section 12.1. This defaults to the grammarfetchhint property.
fetchtimeout	See Section 12.1.

10.2 DTMF Grammars

The `<dtmf>` element is used to specify a DTMF grammar that

- defines a set of key presses that a user may use to perform an action or supply information, and
- defines the corresponding string value that describes that information or action.

The `<dtmf>` element is designed to accommodate any grammar format that meets these two requirements. VoiceXML does not specify nor require support for any particular grammar format: as with `<grammar>`, it is expected that standards efforts and market pressures will cause each widely used VoiceXML interpreter context to support a common set of formats.

The `<dtmf>` element can refer to an external grammar:

```
<dtmf src="URI" type="mime-type"/>
```

or to an inline grammar:

```
<dtmf type="mime-type">
     <!-- inline dtmf grammar -->
</dtmf>
```

The attributes of `<dtmf>` are precisely those of `<grammar>`:

src	The URI specifying the location of the grammar, if it is external.
scope	Either `document`, which makes the grammar active in all dialogs of the current document (and relevant application leaf documents), or `dialog`, to make the grammar active throughout the current form. If omitted, the grammar scoping is resolved by looking at the parent element.
type	The MIME type of the grammar. If this is omitted, the interpreter context will attempt to determine the type dynamically.
caching	See Section 12.1.
fetchhint	See Section 12.1. This defaults to the `grammarfetchhint` property.
fetchtimeout	See Section 12.1.

10.3 Scope of Grammars

Field grammars are always scoped to their fields, that is, they are not active unless the interpreter is visiting that field. Grammars contained in fields cannot specify a scope.

Link grammars are given the scope of the element that contains the link. Thus, if they are defined in the application root document, links are also active in any other loaded application document. Grammars contained in links cannot specify a scope.

Form grammars are by default given `dialog` scope, so that they are active only when the user is in the form. If they are given scope `document`, they are active whenever the user is in the document. If they are given scope `document` and the document is the application root document, then they are also active whenever the user is in another loaded document in the same application. A grammar in a form may be given `document` scope either by specifying the `scope` attribute on the form element or by specifying the scope attribute on the `<grammar>`

element. If both are specified, the grammar assumes the scope specified by the <grammar> element.

<menu> grammars are also by default given dialog scope, and are active only when the user is in the menu. But they can be given the document scope and be active throughout the document, and if their document is the application root document, also be active in any other loaded document belonging to the application. Grammars contained in menu choices cannot specify a scope.

Sometimes a form may need to have some grammars active throughout the document, and other grammars that should be active only when in the form. One reason for doing this is to minimize grammar overlap problems. To do this, each individual <grammar> and <dtmf> element can be given its own scope if that scope should be different than the scope of the <form> element itself:

```
<form scope="document">
  <grammar> … </grammar>
  <grammar scope="dialog"> … </grammar>
</form>
```

10.4 Activation of Grammars

When the interpreter waits for input as a result of visiting a field, the following grammars are active:

- grammars for that field, including grammars contained in links in that field;
- grammars for its form, including grammars contained in links in that form;
- grammars contained in links in its document, and grammars for menus and other forms in its document which are given document scope;
- grammars contained in links in its application root document, and grammars for menus and forms in its application root document which are given document scope.

In the case that an input matches more than one active grammar, the list above defines the precedence order. If the input matches more than one active grammar with the same precedence, the precedence is determined using document order. Menus behave with regard to grammar activation like their equivalent forms (see Section 7).

If the form item is modal (i.e., its modal attribute is set to true), all grammars except its own are turned off while waiting for input. If the input matches a grammar in a form or menu other than the current form or menu, control passes to the other form or menu. If the match causes control to leave the current form, all current form data is lost.

11 EVENT HANDLING

The platform throws events when the user does not respond, doesn't respond intelligibly, requests help, etc. The interpreter throws events if it finds a semantic error in a VoiceXML document, or when it encounters a <throw> element. Events are identified by character strings.

Each element in which an event can occur has a set of *catch elements*, which include:

- <catch>

- <error>
- <help>
- <noinput>
- <nomatch>

An element inherits the catch elements ("as if by copy") from each of its ancestor elements, as needed. If a field, for example, does not contain a catch element for nomatch, but its form does, the form's nomatch catch element is used. In this way, common event handling behavior can be specified at any level, and it applies to all descendents.

11.1 Throw

The <throw> element throws an event. These can be the pre-defined ones:

```
<throw event="nomatch"/>
<throw event="telephone.disconnect.hangup"/>
```

or application-defined events:

```
<throw event="com.att.portal.machine"/>
```

Attributes of <throw> are:

event	The event being thrown.

11.2 Catch

The catch element associates a catch with a document, dialog, or form item. It contains executable content.

```
<form id="launch_missiles">
  <field name="password">
    <prompt>What is the code word?</prompt>
    <grammar>rutabaga</grammar>
    <help>It is the name of an obscure vegetable.</help>
    <catch event="nomatch noinput" count="3">
      <prompt>Security violation!</prompt>
      <submit next="apprehend_felon" namelist="user_id"/>
    </catch>
  </field>
  <block>
    <goto next="#get_city"/>
  </block>
</form>
```

Attributes of <catch> are:

event	The event or events to catch.
count	The occurrence of the event (default is 1). The count allows you to handle different occurrences of the same event differently. Each form item and <menu> maintains a counter for each event that occurs while it is being visited; these counters are reset each time the <menu> or form item's <form> is re-entered.
cond	An optional condition to test to see if the event may be caught by this element. Defaults to true.

11.3 Shorthand Notation

The <error>, <help>, <noinput>, and <nomatch> elements are shorthands for very common types of <catch> elements.

The <error> element is short for <catch event="error"> and catches all events of type error:

```
<error>An error has occurred -- please call again later.<exit/></error>
```

The <help> element is an abbreviation for <catch event="help">:

```
<help>No help is available.</help>
```

The <noinput> element abbreviates <catch event="noinput">:

```
<noinput>I didn't hear anything, please try again.</noinput>
```

And the <nomatch> element is short for <catch event="nomatch">:

```
<nomatch>I heard something, but it wasn't a known city.</nomatch>
```

These elements take the attributes:

count	The event count (as in <catch>).
cond	An optional condition to test to see if the event is caught by this element (as in <catch>). Defaults to true.

11.4 Catch Element Selection

An element inherits the catch elements ("as if by copy") from each of its ancestor elements, as needed. When an event is thrown, the scope in which the event is handled and its enclosing scopes are examined to find the *best qualified* catch element, according to the following algorithm:

1. Form an ordered list of catches consisting of all catches in the current scope and all enclosing scopes (form item, form, document, application root document, interpreter context), ordered first by scope (starting with the current scope), and then within each scope by document order.
2. Remove from this list all catches whose event name does not match the event being thrown or whose cond evaluates to false.
3. Find the "correct count": the highest count among the catch elements still on the list less than or equal to the current count value.
4. Select the first element in the list with the "correct count".

The name of a thrown event matches the catch element event name if it is either an exact match or a prefix match. A prefix match occurs when the catch element event attribute has a prefix in common with the name of the event being thrown. For example,

```
<catch event="telephone.disconnect">
```

will prefix match the event telephone.disconnect.transfer.

11.5 Default Catch Elements

The interpreter is expected to provide implicit default catch handlers for the noinput, help, nomatch, cancel, exit, and error events if the author did not specify them.

The system default behavior of catch handlers for various events and errors is summarized by the definitions below that specify (1) whether any audio response is to be provided, and (2) how execution is affected. Note: where an audio response is provided, the actual content is platform dependent.

Event Type	Audio Provided	Action
cancel	no	don't reprompt
error	yes	exit interpreter
exit	no	exit interpreter
help	yes	reprompt
noinput	no	reprompt
nomatch	yes	reprompt
telephone.disconnect	no	exit interpreter
all others	yes	exit interpreter

Specific platforms and locales will differ in the default prompts presented.

11.6 Event Types

There are pre-defined events and application-defined events. Events are also subdivided into plain events (things that happen normally), and error events (abnormal occurrences). The error naming convention allows for multiple levels of granularity.

The pre-defined events are:

cancel	The user has requested to cancel playing of the current prompt.
telephone.disconnect.hangup	The user has hung up.
telephone.disconnect.transfer	The user has been transferred unconditionally to another line and will not return.
exit	The user has asked to exit.
help	The user has asked for help.
noinput	The user has not responded within the timeout interval.
nomatch	The user input something, but it was not recognized.

The predefined errors are:

error.badfetch	A failed fetch. This may be the result, for example, of a missing document, a malformed URI, a communications error during the process of fetching the document, a timeout, a security violation, or a malformed document.
error.semantic	A run-time error was found in the VoiceXML document, e.g. a divide by 0, substring bounds error, or an undefined variable was referenced.

`error.noauthorization`	The user is not authorized to perform the operation requested (such as dialing an invalid telephone number, or one for which the user is not allowed to call).
`error.unsupported.format`	The requested resource has a format that is not supported by the platform, e.g. an unsupported grammar format, audio file format, object type, or MIME type.
`error.unsupported.`*`element`*	The platform does not support the given *element*. For instance, if a platform does not implement <`record`>, it must throw `error.unsupported.record`. This allows an author to use event handling to adapt to different platform capabilities.

Application-specific error types should follow the following format:

`error.com.mot.mix.noauth`	Access to personal profile information is not authorized.
`error.com.ibm.portal.restricted`	The document tried to access a restricted resource.

Catches can catch specific events (`cancel`) or all those sharing a prefix (`error.unsupported`).

12 RESOURCE FETCHING

12.1 Fetching

Fetching of content from a URI occurs in a VoiceXML interpreter context to: (1) fetch VoiceXML documents to interpret, or (2) fetch other document types, such as audio files, objects, grammars, and scripts. All occasions for fetching content in a VoiceXML interpreter context are governed by the following three attributes:

`caching`	Either `safe` to force a query to fetch the most recent copy of the content, or `fast` to use the cached copy of the content if it has not expired. If not specified, a value derived from the innermost `caching` property is used.
`fetchtimeout`	The interval to wait for the content to be returned before throwing an `error.badfetch` event. If not specified, a value derived from the innermost `fetchtimeout` property is used.
`fetchhint`	Defines when the interpreter context 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. In the case of a very large file (implying long download times) or a streaming audio source, `stream` indicates to the interpreter context to begin processing the content as it arrives and should not wait for full retrieval of the

content. If not specified, a value derived from the innermost relevant *fetchhint property is used.

When content is fetched from a URI, the caching attribute determines where it is located (in the cache or not), the fetchtimeout attribute determines how long to wait for the content (starting from the time when the resource is needed), and fetchhint determines when the content is fetched. The caching policies for a VoiceXML interpreter context are explained in more detail in the next section.

The fetchhint attribute is used to help interpreter contexts that can improve their performance by exploiting information about when content can be fetched. There is no requirement that an interpreter context must actually change when it fetches documents from other than a safe setting. However, any interpreter context that is capable of operating in a prefetch or stream setting, must also be able to operate under the safe setting.

When transitioning from one dialog to another, through either a <subdialog>, <goto>, <submit>, <link>, or <choice> element, there are additional rules that affect interpreter behavior. If the referenced URI names a document (e.g. "doc#dialog") or query data is provided (through POST or GET), then a new document is obtained (either from the local cache or from a server). When it is obtained, the document goes through its initialization phase (i.e., obtaining and initializing a new application root document if needed, initializing document variables, and executing document scripts). The requested dialog (or first dialog if none is specified) is then initialized and execution of the dialog begins. If the referenced URI names only a fragment (e.g. "#dialog") then no document is obtained, and no initialization of the document is performed. The requested dialog is processed as before.

Elements that fetch VoiceXML documents also support the following additional attribute:

fetchaudio	The URI of the audio clip to play while the fetch is being done. If not specified, the fetchaudio property is used, and if that property is not set, no audio is played during the fetch.

The fetchaudio attribute is useful for enhancing a user experience when there may be noticeable delays while the next document is retrieved. This can be used to play background music, or a series of announcements. When the document is retrieved, the audio file is interrupted if it is still playing.

12.2 Caching

The VoiceXML interpreter context, just like HTML visual browsers, can use caching to improve performance in fetching documents and other resources; audio recordings (which can be quite large) are as common to VoiceXML documents as images are to HTML pages. In a visual browser it is common to include end user controls to update or refresh content that is perceived to be stale. This is not the case for theVoiceXML interpreter context, since it lacks equivalent end user controls. Thus enforcement of cache refresh is at the discretion of the applications program through appropriate use of the caching policies employed by VoiceXMLinterpreter contexts.

The default caching policy for VoiceXML interpreter contexts is one commonly employed in HTML browsers:

- If the document referenced by a URI is unexpired in the cache, then use the cached copy.

- If the document referenced by a URI is expired or not present in the cache, then fetch it from the server using get. Note: it is an optimization to perform a “get if modified” on an expired document still present in the cache.

In VoiceXML this caching policy is known as fast. But because fast cache usage can lead to anomalous results, VoiceXML interpreter contexts also implement a safe caching policy:

- Even if the document referenced by a URI is in the cache and is unexpired, still do a “get if modified” operation. This will force a more recent version of the document to replace the cached version, if a more recent version exists. If no more recent version exists, the server does not go to the expense of transferring the document.
- If the document referenced by a URI is expired or not present in the cache, then fetch it from the server using get. Note: it is an optimization to perform a “get if modified” on an expired document still present in the cache.

The safe caching policy ensures that the VoiceXML interpreter context always has the most up to date version of a document, at the expense of performance (due to the extra access to the document server). The safe policy is similar to the effect of always reloading or refreshing a web page in an HTML visual browser.

VoiceXML allows the author to select which caching policy to use. The caching attribute of certain elements may be set to safe or fast to determine what default policy to use for that element. If the attribute is not specified, the policy is determined a <property> element that specifies a value for the caching property (see Section 17).

For example:

```
<?xml version="1.0"?>
<vxml version="1.0">
   <!-- Elements in this document will by default use caching="fast". -->
   <property name="caching" value="fast"/>
   …
   <form id="test">
      <block>
         <!-- Welcome rarely changes, so fast caching is fine. -->
         <audio src="http://www.weather4U.example/vxml/welcome.wav"/>
         <!-- Ads change all the time, so safe caching is needed. -->
         <audio caching="safe"
           src="http://www.onlineads.example/weather4U/ad17"/>
      </block>
      …
   </form>
   …
</vxml>
```

One common practice will be to use safe caching during development, when documents and resources change continually, and then use fast caching with selected resources fetched “safely” as the application goes into system test and then production.

It is also possible, though perhaps less likely, to have a production application that uses safe caching by default and fetches some resources using the fast caching policy.

13 PROMPT

The prompt element controls the output of synthesized speech and prerecorded audio. Conceptually, prompts are instantaneously queued for playing, so interpretation proceeds until the user needs to provide an input. At this point, the prompts are played, and the system waits for user input. Once the input is received from the speech recognition subsystem (or the DTMF recognizer), interpretation proceeds.

Prompts have the following attributes:

bargein	Control whether a user can interrupt a prompt. Default is true.
cond	An expression telling if the prompt should be spoken. Default is true.
count	A number that allows you to emit different prompts if the user is doing something repeatedly. If omitted, it defaults to "1".
timeout	The timeout that will be used for the following user input. The default noinput timeout is platform specific.

13.1 Basic Prompts

You've seen prompts in the previous examples:

```
<prompt>Please say your city.</prompt>
```

You can leave out the <prompt> … </prompt> if:

- There is no need to specify a prompt attribute (like bargein), and
- The prompt consists entirely of PCDATA (contains no speech markups) or consists of just an <audio> element.

For instance, these are also prompts:

```
Please say your city.
<audio src="say_your_city.wav"/>
```

But the <prompt> … </prompt> cannot be removed from this prompt due to the embedded speech markups:

```
<prompt>Please <emp>say</emp> your city.</prompt>
```

13.2 Speech Markup

Prompts can have markup to indicate emphasis, breaks, and prosody:

```
<prompt> This is <emp>also</emp> computer-generated text.
   <break size="medium"/> Do you like it? </prompt>
```

VoiceXML supports the following speech markup elements:

13.2.1 <break>

Specifies a pause in the speech output. Attributes of <break> are:

msecs	The number of milliseconds to pause.

size	A relative pause duration. Possible values are: none, small, medium or large.

At most one of msecs and size must be specified. If neither are specified, size="medium" is assumed.

13.2.2 <div>

Identifies the enclosed text as a particular type. Attributes of <div> are:

type	Possible values are sentence or paragraph.

13.2.3 <emp>

Specifies that the enclosed text should be spoken with emphasis. Attributes of <emp> are:

level	Specifies the level of emphasis. Possible values are: strong, moderate (default), none or reduced.

13.2.4 <pros>

Specifies prosodic information for the enclosed text. For details about the format of attribute values, see the *Java™ API Speech Markup Language specification* (v0.5 - August 28, 1997)

Attributes of <pros> are:

rate	Specifies the speaking rate.
vol	Specifies the output volume.
pitch	Specifies the pitch.
range	Specifies the pitch range.

13.2.5 <sayas>

Specifies how a word or phrase is spoken. Attributes of <sayas> are:

phon	The representation of the Unicode International Phonetic Alphabet (IPA) characters that are to be spoken instead of the contained text.
sub	Defines substitute text to be spoken instead of the contained text.
class	Possible values are phone, date, digits, literal, currency, number and time.

Sometimes text needs to be rendered using a particular style. For example, a telephone number adhering to the North American Dialing Plan needs a break after the first three digits, and another break after the second three digits. To effect this, use the class attribute:

```
<prompt>You are calling <value expr="home_num" class="phone"/></prompt>
<prompt>You are calling
     <sayas class="phone">312-555-1212</sayas>
</prompt>
```

While the interpreter must tolerate the full set of speech markup, if its implementation platform uses a text-to-speech engine that doesn't have this level of speech markup functionality, the platform will have to map the VoiceXML markups as best it can. Specifically, all platforms must allow all speech markup elements, and if an element with contained text is not supported, the contained text must still be spoken.

13.3 Audio Prompting

Prompts can have audio clips intermingled with synthesized speech:

```
<prompt>
   Welcome to the Bird Seed Emporium.
   <audio src="http://www.birdsounds.example/thrush.wav"/>
   We have 250 kilogram drums of thistle seed for
   <sayas class="currency">$299.95</sayas>
   plus shipping and handling this month.
   <audio src="http://www.birdsounds.example/mourningdove.wav"/>
</prompt>
```

Audio can be played in any prompt. Typically it is specified via a URI, but it can also be in an *audio* variable previously recorded:

```
<prompt>
   Your recorded greeting is
   <value expr="greeting"/>
   To rerecord, press 1.
   To keep it, press pound.
   To return to the main menu press star M.
   To exit press star, star X.
</prompt>
```

The audio tag can have alternate text (with markups) in case the audio sample is not available:

```
<prompt>
   <audio src="welcome.wav"><emp>Welcome</emp> to Voice Portal.</audio>
</prompt>
```

If the audio file cannot be played (e.g. unsupported format, invalid URI, etc.), the content of the audio element is played instead. The content may include text, speech markup, or another audio element. If the audio file cannot be played (e.g. unsupported format, invalid URI, etc.) and the content of the audio element is empty, an appropriate error event will be thrown.

Attributes of <audio> include:

src	The URI of the audio prompt. See Appendix E for suggested audio file formats.
caching	See Section 12.1.
fetchtimeout	See Section 12.1.
fetchhint	See Section 12.1.

13.4 The <value> Element

Prompts can contain embedded variable references using the <value> element:

```
<prompt>You are calling <value expr="home_num"/></prompt>
```

Attributes of <value> are:

expr	The expression to render.

class	The <sayas> class of the variable, e.g. phone, date, currency. The valid formats are the same as those supported in the <sayas> speech markup.
mode	The type of rendering: tts (the default), or recorded.
recsrc	The URI of the audio files to be concatenated when mode is recorded.

13.5 Barge-in

If an implementation platform supports barge-in, the service author can specify whether a user can interrupt, or "barge-in" on, a prompt. This speeds up conversations, but is not always desired. If the user must hear all of a warning, legal notice, or advertisement, barge-in should be disabled. This is done with the bargein attribute:

```
<prompt bargein="false"><audio src="legalese.wav"/></prompt>
```

Users can interrupt a prompt whose bargein attribute is true, but must wait for completion of a prompt whose bargein attribute is false. In the case where several prompts are queued, the bargein attribute of each prompt is honored during the period of time in which that prompt is playing. If bargein occurs during any prompt in a sequence, all subsequent prompts are not played. If bargein is not specified, then the value of the bargein property is used.

13.6 Prompt Selection

Tapered prompts are those that may change with each attempt. Information-requesting prompts may become more terse under the assumption that the user is becoming more familiar with the task. Help messages become more detailed perhaps, under the assumption that the user needs more help. Or, prompts can change just to make the interaction more interesting.

Each form item and each menu has an internal prompt counter that is reset to one each time the form or menu is entered. Whenever the system uses a prompt, its associated prompt counter is incremented. This is the mechanism supporting tapered prompts.

For instance, here is a form with a form level prompt and field level prompts:

```
<form id="tapered">
  <block>
    <prompt bargein="false">Welcome to the ice cream survey.</prompt>
  </block>
  <field name="flavor">
    <grammar>vanilla|chocolate|strawberry</grammar>
    <prompt count="1">What is your favorite flavor?</prompt>
    <prompt count="3">Say chocolate, vanilla, or strawberry.</prompt>
    <help>Sorry, no help is available.</help>
  </field>
</form>
```

A conversation using this form follows:

C: Welcome to the ice cream survey.	
C: What is your favorite flavor?	*(the "flavor" field's prompt counter is 1)*
H: Pecan praline.	
C: I do not understand.	

C: What is your favorite flavor? *(the prompt counter is now 2)*

H: Pecan praline.

C: I do not understand.

C: Say chocolate, vanilla, or strawberry. *(prompt counter is 3)*

H: What if I hate those?

C: I do not understand.

C: Say chocolate, vanilla, or strawberry. *(prompt counter is 4)*

H: ...

When it is time to select a prompt, the prompt counter is examined. The child prompt with the highest `count` attribute less than or equal to the prompt counter is used. If a prompt has no `count` attribute, a `count` of "1" is assumed.

A *conditional prompt* is one that is spoken only if its condition is satisfied. In this example, a prompt is varied on each visit to the enclosing form.

```
<form id="another_joke">
   <var name="r" expr="Math.random()"/>
   <field name="another" type="boolean">
      <prompt cond="r < .50">
         Would you like to hear another elephant joke?
      </prompt>
      <prompt cond="r >= .50">
         For another joke say yes.  To exit say no.
      </prompt>
      <filled>
         <if cond="another">
            <goto next="#pick_joke"/>
         </if>
      </filled>
   </field>
</form>
```

When a prompt must be chosen, a set of prompts to be queued is chosen according to the following algorithm:

1. Form an ordered list of prompts consisting of all prompts in the enclosing element in document order.
2. Remove from this list all prompts whose `cond` evaluates to `false`.
3. Find the "correct count": the highest count among the prompt elements still on the list less than or equal to the current count value.
4. Remove from the list all the elements that don't have the "correct count".

All elements that remain on the list will be queued for play.

13.7 Timeout

The `timeout` attribute specifies the interval of silence allowed while waiting for user input after the end of the last prompt. If this interval is exceeded, the platform will throw a `noinput` event. This attribute defaults to the value specified by the `timeout` property (see Section 17).

The reason for allowing timeouts to be specified as prompt attributes is to support tapered timeouts. For example, the user may be given five seconds for the first input attempt, and ten seconds on the next.

The prompt `timeout` attribute determines the `noinput` timeout for the following input:

```
<prompt count="1">Pick a color for your new Model T.</prompt>
<prompt count="2" timeout="120s">
   Please choose color of your new nineteen twenty four Ford Model T.
   Possible colors are black, black, or black.  Please take your time.
</prompt>
```

If several prompts are queued before a field input, the timeout of the last prompt is used.

14 FORM ITEMS

A *form item* is an element of a `<form>` that can be visited during form interpretation. They include `<field>`, `<block>`, `<initial>`, `<subdialog>`, `<object>`, `<record>`, and `<transfer>`.

All form items have the following characteristics:

- They have a result variable, specified by the `name` attribute. This variable may be given an initial value with the `expr` attribute.
- They have a guard condition specified with the `cond` attribute.
- Form items are subdivided into *field items*, those that define the form's field item variables, and *control items*, those that help control the gathering of the form's fields. Field items (`<field>`, `<subdialog>`, `<object>`, `<record>`, and `<transfer>`) generally may contain the following elements:
- `<filled>` elements containing some action to execute at the moment the result field is filled in.
- `<property>` elements to specify properties that are in effect for this field item.
- `<prompt>` elements to specify prompts to be played when this field is visited.
- `<grammar>` and `<dtmf>` elements to specify allowable spoken and character input for this field item.
- `<catch>` elements and catch shorthands that are in effect for this field item.

Each field item may have an associated set of *shadow variables*. Shadow variables are used to return results from the execution of a field item, other than the value stored under the `name` attribute. For example, it may be useful to know the confidence level that was obtained as a result of a recognized grammar in a `<field>` element. A shadow variable is referenced as *name*`$.`*shadowvar* where *name* is the value of the field item's `name` attribute, and *shadowvar* is the name of a specific shadow variable. For example, the `<field>` element returns a shadow variable `confidence`. The code fragment below illustrates how this shadow variable is accessed.

```
<field name="state">
  <prompt> Please say the name of a state. </prompt>
  <grammar src="http://mygrammars.example/states.gram"/>
  <filled>
    <if cond="state$.confidence < 0.4">
      <throw  event="nomatch"/>
      </if>
    </filled>
  </field>
```

In the example, the confidence of the result is examined, and the result is rejected if the confidence is too low.

14.1 FIELD

A field specifies an input item to be gathered from the user. Attributes of fields include:

`name`	The field item variable in the dialog scope that will hold the result.
`expr`	The initial value of the form item variable; default is ECMAScript `undefined`. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
`cond`	A boolean condition that must also evaluate to `true` in order for the form item to be visited.
`type`	The type of field, i.e., the name of an internal grammar. This name must be from a standard set supported by all conformant platforms. If not present, `<grammar>` and/or `<dtmf>` elements can be specified instead.
`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 field item variable names. If the grammar returns only one slot, as do the built-in type grammars like `boolean`, then no matter what the slot's name, the field item variable gets the value of that slot.
`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.

The shadow variables of a `<field>` element whose name is *name* are:

name`$.confidence`	The confidence level in the recognized result from 0.0-1.0. A value of 0.0 indicates minimum confidence, and a value of 1.0 indicates maximum confidence. More specific interpretation of a confidence value is platform-dependent.
name`$.utterance`	The raw string of words that were recognized. The exact tokenization and spelling is platform-specific (e.g. "five hundred thirty" or "5 hundred 30" or even "530").
name`$.inputmode`	The mode in which user input was provided (`dtmf` or `voice`).

14.1.1 Fields Using Built-in Grammars

The `<field>` `type` attribute is used to specify a built-in grammar for one of the fundamental types, and also specifies how its value is to be spoken if subsequently used in a value attribute in a prompt. An example:

```
<field name="lo_fat_meal" type="boolean">
  <prompt>Do you want a low fat meal on this flight?</prompt>
  <help>Low fat means less than 10 grams of fat, and under
          250 calories.</help>
  <filled>
    <prompt> I heard <emp><value expr="lo_fat_meal"/></emp>.</prompt>
  </filled>
</field>
```

In this example, the `boolean` type indicates that inputs are various forms of `true` and `false`. The value actually put into the field is either `true` or `false`. The field would be read "yes" or "no" in prompts.

In the next example, `digits` indicates that input will be spoken or keyed digits. The result is stored as a string, and rendered as digits, i.e., "one-two-three", not "one hundred twenty-three". The `<filled>` action tests the field to see if it has 12 digits. If not, the user hears the error message, and `nomatch` event is thrown to cause a `reprompt`.

```
<field name="ticket_num" type="digits">
  <prompt>Read the 12 digit number from your ticket.</prompt>
  <help>The 12 digit number is to the lower left.</help>
  <filled>
      <if cond="ticket_num.length != 12">
        <prompt>Sorry, I didn't hear exactly 12 digits.</prompt>
        <assign name="ticket_num" expr="undefined"/>
      </if>
  </filled>
</field>
```

It is important that there be input conventions for each built-in type, so that, for instance, generic prompt and help messages can be written that apply to all implementations of VoiceXML. These are locale-dependent, and a certain amount of variability is allowed. For example, the `boolean` type's grammar should minimally allow "yes" and "no" responses, but each implementation is free to add other choices, such as "yeah" and "nope". In cases where an application requires a different behavior, it should use explicit field grammars.

In addition, each built-in type has a convention for the format of the value returned. These are independent of locale and of the implementation. The return type for built-in fields is `string` except for the `boolean` field type. To access the actual recognition result, the author can reference the shadow variable *name*`$.utterance`.

All built-in types must support both voice and DTMF entry.

The builtin types are:

`boolean`	Inputs include affirmative and negative phrases appropriate to the current locale. DTMF 1 is yes and 2 is no. The result is ECMAScript `true` for "yes" or `false` for "no". The value will be submitted as the string "true" or the string "false". If the field value is subsequently used in a prompt, it will be spoken as an affirmative or negative phrase appropriate to the current locale.

date	Valid spoken inputs include phrases that specify a date, including a month day and year. DTMF inputs are: four digits for the year, followed by two digits for the month, and two digits for the day. The result is a fixed-length date string with format yyyymmdd, e.g. “20000704”. If the year is not specified, yyyy is returned as “????”; if the month is not specified mm is returned as “??”; and if the day is not specified dd is returned as “??”.
digits	Valid spoken or DTMF inputs include one or more digits, 0 through 9. The result is a string of digits. If the field value is subsequently used in a prompt, it will be spoken as a sequence of digits. A user can say for example “two one two seven”, but not “twenty one hundred and twenty-seven”.
currency	Valid spoken inputs include phrases that specify a currency amount. For DTMF input, the “*” key will act as the decimal point. The result is a string with the format UUUmm.nn, where UUU is the three character currency indicator according to ISO standard 4217:1995 or null if not spoken by the user. If the field value is subsequently used in a prompt, it will be spoken as a currency amount appropriate to the current locale.
number	Valid spoken inputs include phrases that specify numbers, such as “one hundred twenty-three”, or “five point three”. Valid DTMF input includes positive numbers entered using digits and “*” to represent a decimal point. The result is a string of digits from 0 to 9 and may optionally include a decimal point (“.”) and/or a plus or minus sign.
phone	Valid spoken inputs include phrases that specify a phone number. DTMF asterisk “*” represents “x”. The result is a string containing a telephone number consisting of a string of digits and optionally containing the character “x” to indicate a phone number with an extension. For North America, a result could be “8005551234x789”.
time	Valid spoken inputs include phrases that specify a time, including hours and minutes. The result is a five character string in the format hhmmx, where x is one of “a” for AM, “p” for PM, “h” to indicate a time specified using 24 hour clock, or “?” to indicate an ambiguous time. Input can be via DTMF. Because there is no DTMF convention for specifying AM/PM, in the case of DTMF input, the result will always end with “h” or “?”. If the field value is subsequently used in a prompt, the value will be spoken as a time appropriate to the current locale.

14.1.2 Fields Using Explicit Grammars

Explicit grammars can be specified via a URI, which can be absolute or relative:

```
<field name="flavor">
  <prompt>What is your favorite ice cream?</prompt>
  <grammar src="../grammars/ice_cream.gram" type="application/x-jsgf"/>
```

```
</field>
```

Grammars can be specified inline, for example using JSGF:

```
<field name="flavor">
  <prompt>What is your favorite flavor?</prompt>
  <help>Say one of vanilla, chocolate, or strawberry.</help>
  <grammar type="application/x-jsgf">
      vanilla {van} | chocolate {choc} | strawberry {straw}
  </grammar>
  <dtmf type="application/x-jsgf"> 1 {van} | 2 {choc} | 3 {straw} </dtmf>
</field>
```

14.1.3 Fields Using Option Lists

When a simple set of alternatives is all that is needed to specify the legal input values for a field, it may be more convenient to use an option list than a grammar. An option list is represented by a set of <option> elements contained in a <field> element. Each <option> element contains PCDATA that is used to generate a grammar for the spoken input it accepts using the same method described for <choice>. It also has attributes specifying the DTMF key for selecting the option and the value to assign to the field when the option is chosen.

The following field offers the user three choices and assigns the value of the value attribute of the selected option to the maincourse variable:

```
<form>
  <field name="maincourse">
    <prompt>Please select an entree. Today, we're featuring <enumerate/></prompt>

    <option dtmf="1" value="fish"> swordfish </option>
    <option dtmf="2" value="beef"> roast beef </option>
    <option dtmf="3" value="chicken"> frog legs </option>

    <filled>
      <submit next="/cgi-bin/maincourse.cgi" method="post" namelist="maincourse"/>
    </filled>
  </field>
</form>
```

This conversation might sound like:

C: Please select an entree. Today, we're featuring swordfish; roast beef; frog legs.

H: frog legs

C: *(assigns "chicken" to "maincourse", then submits "maincourse=chicken" to /maincourse.cgi)*

The <enumerate> element is discussed in Section 7.

The attributes of <option> are:

dtmf	The DTMF sequence for this option.
value	The string to assign to the field item variable when a user selects this option, whether by speech or DTMF. The default value for this attribute is the CDATA content of the <option> element with leading and trailing white space removed.

14.1.4 Built-in Grammars

Some built-in field types can be parameterized. This may be done by explicitly referring to built-in grammars using a special-purpose "builtin:" URI scheme and a URI-style query syntax of the form *type*?*param*=*value* in the src attribute of a <grammar> or <dtmf> element, or in the type attribute of a field, for example:

```
<grammar src="builtin:grammar/boolean"/>
<dtmf src="builtin:dtmf/boolean?y=7"/>
<field type="digits?minlength=3;maxlength=5">…</field>
```

By definition the following:

```
<field type="X">…</field>
```

is equivalent to:

```
<field>
  <grammar src="builtin:grammar/X"/>
  <dtmf src="builtin:dtmf/X"/>
  …
</field>
```

where *X* is one of the built-in field types (boolean, date, etc.). The digits and boolean grammars may be parameterized as follows:

digits?minlength=n	A string of at least *n* digits.
digits?maxlength=n	A string of at most *n* digits.
digits?length=n	A string of exactly *n* digits.
boolean?y=d	A DTMF grammar that treats the keypress *d* as an affirmative answer.
boolean?n=d	A DTMF grammar that treats the keypress *d* as a negative answer.

Note that more than one parameter may be specified separated by ";" as illustrated above. In <grammar> or <dtmf> elements, the src attribute URI must start with builtin:grammar/ or builtin:dtmf/ as shown above. When a <grammar> element is specified in a <field>, it overrides the default speech grammar implied by the type attribute of the field. Likewise, when a <dtmf> element is specified in a <field>, it overrides the default DTMF grammar.

14.2 BLOCK

This element is a form item. It contains executable content that is executed if the block's form item variable is undefined and the block's cond attribute, if any, evaluates to true.

```
<block>
   Welcome to Flamingo.example, your source for lawn ornaments.
</block>
```

The form item variable is automatically set to true just before the block is entered. Therefore, blocks are typically executed just once per form invocation.

Sometimes you may need more control over blocks. To do this, you can name the form item variable, and set or clear it to control execution of the <block>. This variable is declared in the dialog scope of the form.

Attributes of <block> include:

name	The name of the form item variable used to track whether this block is eligible to be executed; defaults to an inaccessible internal variable.
expr	The initial value of the form item variable; default is ECMAScript undefined. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
cond	A boolean condition that must also evaluate to true in order for the form item to be visited.

14.3 INITIAL

In a typical mixed initiative form, the <initial> element is visited when the user is initially being prompted for form-wide information, and has not yet entered into the directed mode where each field is solicited individually. Like field items, it has prompts, catches, and event counters. Unlike field items, <initial> has no grammars, and no <filled> action. For instance:

```
<form id="get_from_and_to_cities">
   <grammar src="http://www.directions.example/grammars/from_to.gram"/>
   <block>
      Welcome to the Driving Directions By Phone.
   </block>
   <initial name="bypass_init">
      <prompt>Where do you want to drive from and to?</prompt>
      <nomatch count="1">
         Please say something like "from Atlanta Georgia to Toledo Ohio".
      </nomatch>
      <nomatch count="2">
         I’m sorry, I still don’t understand.
         I’ll ask you for information one piece at a time.
         <assign name="bypass_init" expr="true"/>
         <reprompt/>
      </nomatch>
   </initial>
   <field name="from_city">
      <grammar src="http://www.directions.example/grammars/city.gram"/>
      <prompt>From which city are you leaving?</prompt>
      … etc. …
   </field>
   … etc. …
</form>
```

While visiting an <initial> element, no field grammar is active. If an event occurs while visiting an <initial>, then one of its event handlers executes. As with other form items, <initial> continues to be eligible to be visited while its form item variable is undefined and while its cond attribute is true. If one or more of the field item variables is set by user input, then all <initial> form item variables are set to true, before any <filled> actions are executed.

An <initial> form item variable can be manipulated explicitly to disable, or re-enable the <initial>'s eligibility to the FIA. For example, in the program above, the <initial>'s form item variable is set on the second nomatch event. This causes the FIA to no longer consider the <initial> and to choose the next form item, which is a <field> to prompt explicitly for the origination city. Similarly, an `<initial>`'s form item variable could be cleared, so that `<initial>` gets selected again by the FIA.

Note: explicit assignment of values to field item variables does not affect the value of an `<initial>`'s form item variable.

Attributes of `<initial>` include:

`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`	The initial value of the form item variable; default is ECMAScript `undefined`. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
`cond`	A boolean condition that must also evaluate to `true` in order for the form item to be visited.

14.4 SUBDIALOG

A `<subdialog>` element invokes a "called" dialog (known as the *subdialog*) identified by its `src` attribute. The subdialog executes in a new execution context. The subdialog proceeds until the execution of a `<return>` element which causes the subdialog to return. When the subdialog returns, its execution context is deleted, and execution resumes in the calling dialog with any appropriate `<filled>` elements. An execution context includes all declarations and state information for the dialog, the dialog's document, and the application root (if present). Subdialogs can permit the reuse of a common dialog such as this example of prompting a user for credit card information, or build libraries of reusable applications.

The attributes are:

`name`	The result returned from the subdialog, an ECMAScript object whose properties are the ones defined in the `namelist` attribute of the `<return>` element.
`expr`	The initial value of the form item variable; default is ECMAScript `undefined`. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
`cond`	A boolean condition that must also evaluate to `true` in order for the form item to be visited.
`modal`	Controls which grammars are active during the subdialog. If `true` (the default) all grammars active in the calling dialog are disabled. If `false`, they remain active.
`namelist`	Same as `namelist` in `<submit>`, except that the default is to submit nothing. Only valid when fetching another document.
`src`	The URI of the `<subdialog>`.
`method`	See Section 19.8.
`enctype`	See Section 19.8.
`caching`	See Section 12.1.

fetchaudio	See Section 12.1.
fetchtimeout	See Section 12.1.
fetchhint	See Section 12.1.

The <subdialog> element may contain elements common to all form items, and may also contain <param> elements. The <param> elements of a <subdialog> specify the parameters to pass to the subdialog. These parameters must be declared in the subdialog using <var> elements; it is a semantic error to attempt to set a form item variable or an undeclared variable using <param>. When a subdialog initializes, its variables are initialized in document order to the corresponding <param> value, if they don't have an expr attribute. Thus <param> elements can only initialize <var> elements without expr attributes.

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

```
<!-- form dialog that calls a subdialog -->
<form>
  <subdialog name="result" src="#getdriverslicense">
    <param name="birthday" expr="'2000-02-10'"/>
    <filled>
      <submit next="http://myservice.example/cgi-bin/process"/>
    </filled>
  </subdialog>
</form>

<!-- subdialog to get drivers license -->
<form id="getdriverslicense">
  <var name="birthday"/>

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

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

This example also illustrates the convenience of using <param> as a means for forwarding data to the subdialog as a means of instantiating values in the subdialog without using server side scripting. An alternate solution that uses scripting, is shown below.

Document with form that calls a subdialog

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

  <form>
    <field name="birthday" type="date">
```

```
        What is your birthday?
      </field>
      <subdialog name="result"
                 src="/cgi-bin/getlib#getdriverslicense"

                 namelist="birthday">
        <filled>
          <submit next="http://myservice.example/cgi-bin/process"/>
        </filled>
      </subdialog>
    </form>

  </vxml>
```

<u>Document containing the subdialog</u> (generated by `/cgi-bin/getlib`)

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

    <form id="getdriverlicense">
      <var name="birthday" expr="'1980-02-10'"/> <!-- Generated by server script -->

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

  </vxml>
```

In the above example, a server side script had to generate the document and embed the birthday value.

When a subdialog is interpreted, the only active grammars are those in dialog-scope of the subdialog and the default grammars defined by the interpreter context (e.g. help, cancel). The set of active grammars remains limited for all subsequent dialogs until a <`return`> is executed. For example, if subdialog A transitions to dialog B, then the interpretation of B considers only active grammars in its dialog scope and the default grammars.

One last example is shown below that illustrates a subdialog to capture general credit card information. First the subdialog is defined in a separate document; it is intended to be reusable across different applications. It returns a status, the credit card number, and the expiry date; if a result cannot be obtained, the status is returned with value "`no_result`".

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

      <!-- Example of subdialog to collect credit card information. -->
      <!-- file is at http://www.somedomain.example/ccn.vxml -->

      <form id="getcredit">
          <var name="status" expr="'no_result'"/>
          <var name="username"/>
      <field name="creditcardnum">
         <prompt> What is your credit card number? </prompt>
         <help>
            I am trying to collect your credit card information.
            <reprompt/>
         </help>
         <nomatch> <return namelist="status"/> </nomatch>
         <grammar .../>
      </field>
      <field name="expirydate" type="date">
         <prompt>
            What is the expiry date of this card?
         </prompt>
         <help>
            I am trying to collect the expiry date of the credit
            card number you provided.
            <reprompt/>
         </help>
         <nomatch>
            <return namelist="status"/>
         </nomatch>
      </field>
      <block>
         <assign name="status" expr="'result'"/>
         <return namelist="status creditcardnum expirydate"/>
      </block>
   </form>

</vxml>
```

An application that includes a calling dialog is shown below. It obtains the name of a software product and operating system using a mixed initiative dialog, and then solicits credit card information using the subdialog.

```
<?xml version="1.0"?>
<vxml version="1.0">
<!-- Example main program -->
<!-- http://www.somedomain.example/main.vxml -->
<!-- calls subdialog ccn.vxml -->

<var name="username"/> <!-- assume this gets defined by some dialog -->

<form id="buysoftware">
   <var name="ccn"/>
   <var name="exp"/>
   <grammar ..../>
   <initial name="start">
      <prompt>
         Please tell us the software product you wish to buy and
         the operating system on which it must run.
      </prompt>
      <noinput>
         <assign name="start" expr="true"/>
      </noinput>
   </initial>
   <field name="product">
       <prompt> Which software product would you like to buy? </prompt>
   </field>
   <field name="operatingsystem">
       <prompt> Which operating system does this software need to run on?
       </prompt>
   </field>
   <subdialog name="cc_results" src="http://somedomain.example/ccn.vxml">
      <filled>
         <if cond="cc_results.status=='no_result'">
            Sorry, your credit card information could not be
            Obtained. This order is cancelled.
            <exit/>
         <else/>
            <assign name="ccn" expr="cc_results.creditcardnum"/>
            <asssign name="exp" expr="cc_results.expirydate"/>
         </if>
      </filled>
   </subdialog>
   <block>
       We will now process your order. Please hold.
       <submit namelist="username product operatingsystem ccn exp"/>
   </block>

</vxml>
```

14.5 OBJECT

A VoiceXML implementation platform may have platform-specific functionality that an application wants to use, such as speaker verification, native components, additional telephony functionality, and so on. Such platform-specific objects are accessed using the <object> element, which is analogous to the HTML <OBJECT> element. For example, a native credit card collection object could be accessed like this:

```
<object name="debit"
        classid="method://credit_card/gather_and_debit"
        data="http://www.recordings.example/prompts/credit/jesse.jar"/>
   <param name="amount" expr="document.amt"/>
   <param name="vendor" expr="vendor_num"/>
</object>
```

In this example, the <param> element (Section 18) is used to pass parameters to the object when it is invoked. When this <object> is executed, it returns an ECMAScript object as the value of its form item variable. This <block> presents the values returned from the credit card object:

```
<block>
   <prompt>The card type is <value expr="debit.card"/>. </prompt>
   <prompt>The card number is <value expr="debit.card_no"/>. </prompt>
   <prompt>The expiration date is <value expr="debit.expiry_date"/>. </prompt>
   <prompt>The approval code is <value expr="debit.approval_code"/>. </prompt>
   <prompt>The confirmation number is <value expr="debit.conf_no"/>. </prompt>
</block>
```

As another example, suppose that a platform has a feature that allows the user to enter arbitrary text messages using a telephone keypad.

```
<form id="gather_pager_message">
  <object name="message" classid="builtin://keypad_text_input">
    <prompt>
      Enter your message by pressing your keypad once per letter.  For
      a space, enter star.  To end the message, press the pound sign.
    </prompt>
  </object>
  <block>
    <assign name="document.pager_message" expr="message.text"/>
    <goto next="#confirm_pager_message"/>
  </block>
</form>
```

The user is first prompted for the pager message, then keys it in. The `<block>` copies the message to the variable `document.message`.

Attributes of `<object>` include:

`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 ECMAScript `undefined`. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
`cond`	A boolean condition that must also evaluate to `true` 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`. When absent it defaults to 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. URIs which are relative are interpreted relative to the `codebase` attribute.
`caching`	See Section 12.1.
`fetchaudio`	See Section 12.1.
`fetchhint`	See Section 12.1. This defaults to the `objectfetchhint` property.
`fetchtimeout`	See Section 12.1.

If an `<object>` element refers to an unknown object, the `error.unsupported.object` event is thrown. There is no requirement for implementations to provide platform-specific objects, although support for the `<object>` element is required.

14.6 RECORD

The `<record>` element is a field item that collects a recording from the user. The recording is stored in the field item variable, which can be played back or submitted to a server, as shown in this example:

```
<?xml version="1.0"?>
<vxml version="1.0">
   <form>
      <record  name="greeting" beep="true" maxtime="10s"
               finalsilence="4000ms" dtmfterm="true" type="audio/wav">
```

```
            <prompt> At the tone, please say your greeting.</prompt>
            <noinput>I didn't hear anything, please try again.</noinput>
         </record>

         <field name="confirm" type="boolean">
            <prompt>Your greeting is <value expr="greeting"/>.</prompt>
            <prompt>To keep it, say yes.  To discard it, say no.</prompt>
            <filled>
               <if cond="confirm">
                  <submit next="save_greeting.pl"
                     method="post" namelist="greeting"/>
               </if>
               <clear/>
            </filled>
         </field>
      </form>
   </vxml>
```

The user is prompted for a greeting and then records it. The greeting is played back, and if the user approves it, is sent on to the server for storage using the HTTP POST method. Notice that like other field items, <record> has prompts and catch elements. It may also have <filled> actions. If the platform supports simultaneous recognition and recording, form and document scoped grammars can be active while the recording is in progress.

The attributes of <record> are:

name	The field item variable that will hold the recording.
expr	The initial value of the form item variable; default is ECMAScript undefined. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
cond	A boolean condition that must also evaluate to true in order for the form item to be visited.
modal	If this is true (the default) all higher level speech and DTMF grammars are turned off while making the recording. If this is false, speech and DTMF grammars scoped to the form, document, application, and calling documents are listened for. Most implementations will not support simultaneous recognition and recording.
beep	If true, a tone is emitted just prior to recording. Defaults to false.
maxtime	The maximum duration to record.
finalsilence	The interval of silence that indicates end of speech.
dtmfterm	If true, a DTMF keypress terminates recording. Defaults to true. The DTMF tone is not part of the recording.
type	The MIME format of the resulting recording. Defaults to a platform-specific format.

The <record> shadow variable *name*$ has the following ECMAScript properties after the recording has been made:

name$.duration	The duration of the recording in milliseconds.

name`$.size`	The size of the recording in bytes.
name`$.termchar`	If the `dtmfterm` attribute is `true`, and the user terminates the recording by pressing a DTMF key, then this shadow variable is the key pressed (e.g. "#"). Otherwise it is `null`.

14.7 TRANSFER

Occasionally, it is appropriate to suspend the session between the user and the interpreter and initiate a session with another entity. The most common use for this capability in current practice is to connect a user in a telephone conversation with a interpreter to a third party through the telephone network. The `<transfer>` element directs the interpreter to make such a third party connection. Two scenarios are supported:

bridging	the original caller resumes his session with the interpreter.
blind transfer	no resumption is possible; as soon as the call connects, the platform throws a `telephone.disconnect.transfer`.

The form item variable is used to store the outcome of the transfer attempt. Here are the possible values:

`busy`	The endpoint refused the call.
`noanswer`	There was no answer within the specified time.
`network_busy`	Some intermediate network refused the call.
`near_end_disconnect`	The call completed and was terminated by the caller.
`far_end_disconnect`	The call completed and was terminated by the callee.
`network_disconnect`	The call completed and was terminated by the network.

This example attempts to transfer the user to a customer support operator and then wait for that conversation to terminate.

```
<form name="transfer">
   <var name="mydur" expr="0"/>
   <block>
      <audio src="chopin12.wav">
   </block>
   <transfer name="mycall" dest="phone://18005551234"
     connecttimeout="30s" bridge="true">
      <filled>
        <assign name="mydur" expr="mycall$.duration"/>
        <if cond="mycall == 'busy'">
          <prompt>Sorry, our customer support team is busy serving
          other customers.  Please try again later.</prompt>
        <elseif cond="mycall == 'noanswer'"/>
          <prompt>Sorry, our customer support team's normal hours
          are 9 am to 7 pm Monday through Saturday.</prompt>
        </if>
      </filled>
   </transfer>
   <block>
      <submit namelist="mycall mydur" next="/cgi-bin/report"/>
   </block>
</form>
```

During a bridge transfer, the platform can listen for DTMF input from the caller. In particular, if a DTMF grammar appears inside the <transfer> element, DTMF input matching that grammar will terminate the transfer and return control to the interpreter. A bridge transfer may be terminated by recognition of an utterance matching an enclosed <grammar> element; support of this feature is not required. The <transfer> element is modal in that no grammar defined outside its scope is active.

Attributes include:

name	The outcome of the transfer attempt.
expr	The initial value of the form item variable; default is ECMAScript undefined. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
cond	A boolean condition that must also evaluate to true in order for the form item to be visited.
dest	The URI of the destination (phone, IP telephony address).
destexpr	An ECMAScript expression yielding the URI of the destination.
bridge	This attribute determines what to do once the call is connected. If bridge is true, document interpretation suspends until the transferred call terminates. If it is false, as soon as the call connects, the platform throws a telephone.disconnect.transfer.
connecttimeout	The time to wait while trying to connect the call before returning the noanswer condition. Default is platform specific.
maxtime	The time that the call is allowed to last, or 0 if it can last arbitrarily long. Only applies if bridge is true. Default is 0.

The <transfer> shadow variable (*name*$) has the following ECMAScript properties after a transfer completes:

name$.duration	The duration of a successful call in seconds (floating-point).

Events thrown inside a <transfer> include:

telephone.disconnect.hangup	If the caller hung up.
telephone.disconnect.transfer	If the caller has been transferred unconditionally to another line and will not return.

15 FILLED

The <filled> element specifies an action to perform when some combination of fields are filled by user input. It may occur in two places: as a child of the <form> element, or as a child of a field item.

As a child of a `<form>` element, the `<filled>` element can be used to perform actions that occur when a combination of one or more fields is filled. For example, the following `<filled>` element does a cross-check to ensure that a starting city field differs from the ending city field:

```
<form id="get_starting_and_ending_cities">
   <field name="start_city">
      <grammar src="http://www.grammars.example/voicexml/city.gram"/>
      <prompt>What is the starting city?</prompt>
   </field>
   <field name="end_city">
      <grammar src="http://www.grammars.example/voicexml/city.gram"/>
      <prompt>What is the ending city?</prompt>
   </field>
   <filled mode="any" namelist="start_city end_city">
      <if cond="start_city == end_city">
         <prompt>You can't fly from and to the same city.</prompt>
         <clear/>
      </if>
   </filled>
</form>
```

If the `<filled>` element appears inside a field item, it specifies an action to perform after that field is filled in by user input. This is a notational convenience for a form-level `<filled>` element that triggers on a single field item:

```
<form id="get_city">
   <field name="city">
      <grammar src="http://www.ship-it.example/grammars/served_cities.gram"/>
      <prompt>What is the city?</prompt>
      <filled>
         <if cond="city == 'Novosibirsk'">
            <prompt>Note, Novosibirsk service ends next year.</prompt>
         </if>
      </filled>
   </field>
</form>
```

After each gathering of the user's input, all the fields mentioned in the input are set, and then the interpreter looks at each `<filled>` element in document order (no preference is given to ones in fields vs. ones in the form). Those whose conditions are matched by the utterance are then executed in order, until there are no more, or until one transfers control or throws an event.

Attributes include:

`mode`	Either `all` (the default), or `any`. If `any`, this action is executed when any of the specified fields is filled by the last user input. If `all`, this action is executed when all of the mentioned fields are filled, and at least one has been filled by the last user input. A `<filled>` element in a field item cannot specify a mode.
`namelist`	The fields to trigger on. For a `<filled>` in a form, `namelist` defaults to the names (explicit and implicit) of the form's field items. A `<filled>` element in a field item cannot specify a `namelist`; the `namelist` in this case is the field item name.

16 META

The <meta> element specifies meta-data, as in HTML, which is data about the document rather than the document's content. There are two types of <meta>. The first type specifies a meta-data property of the document as a whole. For example to specify the maintainer of a VoiceXML document:

```
<?xml version="1.0"?>
<vxml version="1.0">
   <meta name="maintainer" content="jpdoe@anycompany.example"/>
   …
</vxml>
```

The interpreter could use this information, for example, to compose and email an error report to the maintainer.

VoiceXML does not specify required meta-data properties, but the following are recommended:

author	Information describing the author.
copyright	A copyright notice.
description	A description of the document for search engines.
keywords	Keywords describing the document.
maintainer	The document maintainer's email address.
robots	Directives to search engine web robots.

The second type of <meta> specifies HTTP response headers. In the following example, the first <meta> element sets an expiration date that prevents caching of the document; the second <meta> element sets the Date header.

```
<?xml version="1.0"?>
<vxml version="1.0">
   <meta http-equiv="Expires" content="0"/>
   <meta http-equiv="Date" content="Thu, 12 Dec 1999 23:27:21 GMT"/>
   …
</vxml>
```

Attributes of <meta> are:

name	The name of the meta-data property.
content	The value of the meta-data property.
http-equiv	The name of an HTTP response header. Either name or http-equiv must be specified, not both.

17 PROPERTY

The <property> element sets a property value. Properties are used to set values that affect platform behavior, such as the recognition process, timeouts, caching policy, etc.

Properties may be defined for the whole application, for the whole document at the <vxml> level, for a particular dialog at the <form> or <menu> level, or for a particular form item. Properties apply to their parent element and all the descendants of the parent. A property at a lower level overrides a property at a higher level. Properties specified in the application root document provide default values for properties in every document in the application;

properties specified in an individual document override property values specified in the application root document.

In some cases, `<property>` elements specify default values for element attributes, such as `timeout` or `bargein`. For example, to turn off `bargein` for all the prompts in a particular form:

```
<form id="no_bargein_form">
   <property name="bargein" value="false"/>
   <block>
      <prompt>This introductory prompt cannot be barged into.</prompt>
      <prompt>And neither can this prompt.</prompt>
      <prompt bargein="true">But this one <emp>can</emp> be barged into.</prompt>
   </block>
   …
</form>
```

Properties are also used to specify platform-specific data and settings. For example, to set a platform-specific property to prepend one second of silence before each recording made by a particular document:

```
<?xml version="1.0"?>
<vxml version="1.0">
   <property name="example.acme.endpointing.record_init_silence" value="1s"/>
   … dialogs that make recordings go here …
</vxml>
```

The generic speech recognizer properties are taken from the Java™ Speech API (see http://www.javasoft.com/products/java-media/speech/index.html):

`confidencelevel`	The speech recognition confidence level, a float value in the range of 0.0 to 1.0. Results are rejected (a `nomatch` event is thrown) when the engine's confidence in its interpretation is below this threshold. A value of 0.0 means minimum confidence is needed for a recognition, and a value of 1.0 requires maximum confidence. The default value is 0.5.
`sensitivity`	Set the sensitivity level. A value of 1.0 means that it is highly sensitive to quiet input. A value of 0.0 means it is least sensitive to noise. The default value is 0.5.
`speedvsaccuracy`	A hint specifying the desired balance between speed vs. accuracy. A value of 0.0 means fastest recognition. A value of 1.0 means best accuracy. The default is value 0.5.
`completetimeout`	The speech timeout value to use when an active grammar is matched. The default is platform-dependent. See Appendix F.
`incompletetimeout`	The speech timeout to use when no active grammar has been matched. The default is platform-dependent. See Appendix F.

Several generic properties pertain to DTMF grammar recognition:

`interdigittimeout`	The inter-digit timeout value to use when recognizing DTMF input. The default is platform-dependent. See Appendix F.
`termtimeout`	The terminating timeout to use when recognizing DTMF input. The default value is `"0s"`. See Appendix F.
`termchar`	The terminating DTMF character for DTMF input recognition. The default value is `"#"`. See Appendix F.

These properties apply to the fundamental platform prompt and collect cycle:

`bargein`	The bargein attribute to use for prompts. Setting this to `true` allows barge-in by default. Setting it to `false` disallows barge-in. The default value is `"true"`.
`timeout`	The time after which a `noinput` event is thrown by the platform. The default value is platform-dependent. See Appendix F.

These properties pertain to the fetching of new documents and resources:

`caching`	Either `safe` to never trust the cache when fetching, or `fast` to always trust the cache. The default value is `fast`.
`audiofetchhint`	This tells the platform whether or not it can attempt to optimize dialog interpretation by pre-fetching audio. The value is either `safe` to say that audio is only fetched when it is needed, never before; `prefetch` to permit, but not require the platform to pre-fetch the audio; or `stream` to allow it to stream the audio fetches. The default value is `prefetch`.
`documentfetchhint`	Tells the platform whether or not documents may be pre-fetched. The value is either `safe` (the default), or `prefetch`.
`grammarfetchhint`	Tells the platform whether or not grammars may be pre-fetched. The value is either `prefetch` (the default), or `safe`.
`objectfetchhint`	Tells the platform whether the URI contents for `<object>` may be pre-fetched or not. The values are `prefetch` (the default), or `safe`.
`scriptfetchhint`	Tells whether scripts may be pre-fetched or not. The values are `prefetch` (the default), or `safe`.
`fetchaudio`	The URI of the audio to play while waiting for a document to be fetched. The default is not to play any audio. There are no `fetchaudio` properties for audio, grammars, objects, and scripts.
`fetchtimeout`	The timeout for fetches. The default value is platform-dependent.

This property determines which input modality to use:

`inputmodes`	The input modes to enable: `dtmf` and `voice`. On platforms that support both modes, `inputmodes` defaults to "`dtmf voice`". To disable speech recognition, set inputmodes to "`dtmf`". To disable DTMF, set it to "`voice`". One use for this would be to turn off speech recognition in noisy environments. Another would be to conserve speech recognition resources by turning them off where the input is always expected to be DTMF.

Our last example shows several of these properties used at multiple levels.

```
<?xml version="1.0"?>
<vxml version="1.0">
    <!-- set default characteristics for page -->
    <property name="caching" value="safe"/>
    <property name="audiofetchhint" value="safe"/>
    <property name="confidence" value="0.75"/>

  <form>
```

```
      <!-- override defaults for this form only -->
      <property name="confidence" value="0.5"/>
      <property name="bargein" value="false"/>
      <grammar src="address_book.gram" type="application/x-jsgf"/>

      <block>
        <prompt> Welcome to the Voice Address Book </prompt>
      </block>

      <initial name="start">
        <!-- override default timeout value -->
        <property name="timeout" value="5s"/>
        <prompt> Who would you like to call? </prompt>
      </initial>

      <field name="person">
        <prompt> Say the name of the person you would like to call. </prompt>
      </field>

      <field name="location">
        <prompt> Say the location of the person you would like to call. </prompt>
      </field>

      <field name="confirm" type="boolean">
        <!-- Use actual utterances to playback recognized words,
             rather than returned slot values -->
        <prompt>
          You said to call <value expr="person$.utterance"/>
          at <value expr="location$.utterance"/>.
          Is this correct?
        </prompt>
        <filled>
          <if cond="confirm">
            <submit next="http://www.messagecentral.example/voice/make_call"
                    namelist="person location" />
          </if>
          <clear/>
        </filled>
      </field>
  </form>
</vxml>
```

18 PARAM

The <param> element is used to specify values that are passed to subdialogs or objects. It is modeled on the HTML <PARAM> element. Its attributes are:

name	The name to be associated with this parameter when the object or subdialog is invoked.
expr	An expression that computes the value associated with name.
value	Associates a literal string value with name.
valuetype	One of data or ref, by default data; used to indicate to an object if the value associated with name is data or a URI (ref). This is not used for <subdialog>.
type	The MIME type of the result provided by a URI if the valuetype is ref; only relevant for uses of <param> in <object>.

Exactly one of expr or value must be present. The use of valuetype and type is optional in general, although they may be required by specific objects. When <param> is contained in a <subdialog> element, the values specified by it are used to initialize dialog <var> elements in the subdialog that is invoked. When <param> is contained in an <object>, the use of the parameter data is specific to the object that is being invoked, and is outside the scope of the VoiceXML specification.

Below is an example of <param> used as part of an <object>. In this case, the first two <param> elements have expressions (implicitly of valuetype="data"), the third <param> has an explicit value, and the fourth is a URI that returns a MIME type of text/plain. The meaning of this data is specific to the object.

```
<object  name="debit"
       classid="method://credit_card/gather_and_debit"
       data="http://www.recordings.example/prompts/credit/jesse.jar"/>
    <param name="amount" expr="document.amt"/>
    <param name="vendor" expr="vendor_num"/>
    <param name="application_id" value="ADC5678-QWOO"/>
    <param name="authentication_server" value="http://auth_svr.example"
      valuetype="ref" value="text/plain"/>
</object>
```

The next example illustrates <param> used with <subdialog>. In this case, two expressions are used to initialize variables in the scope of the subdialog form.

<u>Form with calling dialog</u>

```
<form>
  <subdialog name="result" src="http://another.example/#getssn">
    <param name="firstname" expr="document.first"/>
    <param name="lastname" expr="document.last"/>
    <filled>
      <submit namelist="result.ssn"
        next="http://myservice.example/cgi-bin/process"/>
    </filled>
  </subdialog>
</form>
```

<u>Subdialog in http://another.example</u>

```
<form id="getssn">
  <var name="firstname"/>
  <var name="lastname"/>
  <field name="ssn">
    <grammar src="http://grammarlib/ssn.gram" type="application/x-jsgf"/>
    <prompt> Please say social security number. </prompt>
    <filled>
      <if cond="validssn(firstname,lastname,ssn)">
        <assign name="status" expr="true"/>
        <return namelist="status ssn"/>
      <else/>
        <assign name="status" expr="false"/>
        <return namelist="status"/>
      </if>
    </filled>
  </field>
</form>
```

Using <param> in a <subdialog> is a convenient way of passing data to a subdialog without requiring the use of server side scripting.

19 EXECUTABLE CONTENT

Executable content refers to a block of procedural logic. Such logic appears in:

- The `<block>` form item.
- The `<filled>` actions in forms and form items.
- Event handlers (`<catch>`, `<help>`, et cetera).

This section covers the elements that can occur in executable content.

19.1 VAR

This element declares a variable. It can occur in executable content or as a child of `<form>` or `<vxml>`. Examples:

```
<var name="phone" expr="6305551212"/>
<var name="y"     expr="document.z+1"/>
```

If it occurs in executable content, it declares a variable in the anonymous scope associated with the enclosing `<block>`, `<filled>`, or catch element. This declaration is made only when the `<var>` element is executed. If the variable is already declared in this scope, subsequent declarations act as assignments, as in ECMAScript.

If a `<var>` is a child of a `<form>` element, it declares a variable in the `dialog` scope of the `<form>`. This declaration is made during the form's initialization phase as described in Section 6.6.1. The `<var>` element is not a form item, and so is not visited by the Form Interpretation Algorithm's main loop.

If a `<var>` is a child of a `<vxml>` element, it declares a variable in the `document` scope. This declaration is made when the document is initialized; initializations happen in document order.

Attributes of `<var>` include:

`name`	The name of the variable that will hold the result.
`expr`	The initial value of the variable (optional). If there is no `expr` attribute, the variable retains its current value, if any. Variables start out with the ECMAScript value `undefined` if they are not given initial values.

19.2 ASSIGN

The `<assign>` element assigns a value to a variable:

```
<assign name="flavor" expr="'chocolate'"/>
<assign name="document.mycost" expr="document.mycost+14"/>
```

Attributes include:

`name`	The name of the variable being assigned to.
`expr`	The new value of the variable.

19.3 CLEAR

The <clear> element resets one or more form items. Resetting includes:

- Setting the form item variable to ECMAScript undefined.
- Reinitializing the prompt counter and the event counters for the form item.

For example:

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

The attribute is:

namelist	The names of the form items to be reset. When not specified, all form items in the current form are cleared.

19.4 IF, ELSEIF, and ELSE

The <if> element is used for conditional logic. It has optional <else> and <elseif> elements.

```
<if cond="total > 1000">
   <prompt>This is way too much to spend.</prompt>
   <throw "com.xyzcorp.acct.toomuchspent"/>
</if>

<if cond="amount < 29.95">
   <assign name="x" expr="amount"/>
<else/>
   <assign name="x" expr="29.95"/>
</if>

<if cond="flavor == 'vanilla'">
   <assign name="flavor_code" expr="'v'"/>
<elseif cond="flavor == 'chocolate'"/>
   <assign name="flavor_code" expr="'h'"/>
<elseif cond="flavor == 'strawberry'"/>
   <assign name="flavor_code" expr="'b'"/>
<else/>
   <assign name="flavor_code" expr="'?'"/>
</if>
```

19.5 PROMPT

Prompts can appear in executable content, in their full generality, except that the <prompt> count attribute is meaningless. In particular, the cond attribute can be used in executable content. Prompts may be wrapped with <prompt> and </prompt>, or represented using PCDATA. Wherever <prompt> is allowed, the PCDATA *xyz* is interpreted exactly as if it had appeared as <prompt>*xyz*</prompt>.

```
<nomatch count="1">
   To open the pod bay door, say your code phrase clearly.
</nomatch>
<nomatch count="2">
   <prompt> This is your <emp>last</emp> chance. </prompt>
</nomatch>
<nomatch count="3">
   Entrance denied.
   <exit/>
</nomatch>
```

19.6 REPROMPT

The FIA assumes that when a catch element is executed, it has queued appropriate prompts. Therefore the FIA normally suppresses playing of prompts on the iteration of the FIA following the execution of a catch element. However, if a `<reprompt>` is executed in the catch, this tells the FIA that when it selects the next form item to visit, it should do the normal prompt processing (which includes selection of a prompt and incrementing the prompt counter).

For example, this `noinput` catch expects the next form item prompt to be selected and played:

```
<field name="want_ice_cream" type="boolean">
   <prompt>Do you want ice cream for dessert?</prompt>
   <prompt count="2">
      If you want ice cream, say yes.
      If you don't want ice cream, say no.
   </prompt>
   <noinput>
      I could not hear you.
      <reprompt/>   <!-- Cause the next prompt to be selected and played. -->
   </noinput>
</field>
```

A quiet user would hear:

C: Do you want ice cream for dessert?

H: *(silence)*

C: I could not hear you.

C: If you want ice cream, say yes. If you don't want ice cream, say no.

H: *(silence)*

C: I could not hear you.

C: If you want ice cream, say yes. If you don't want ice cream, say no.

H: No

If there were no `<reprompt>`, the user would instead hear:

C: Do you want ice cream for dessert?

H: *(silence)*

C: I could not hear you.

C: I could not hear you.

H: No

Note that if no `<reprompt>` is executed in a catch, then the FIA skips the prompt selection and queuing phase of the selected form item. The form item's prompt counter is therefore not incremented.

If a `<reprompt>` is executed, then the FIA executes the form item's prompt selection queuing phase. This does increment the form item's prompt counter. A `<reprompt>` does not cause the prior prompt to be played, in general, but will cause prompt(s) to be played based on the current value of the prompt counter and the current values of the prompt conditions.

19.7 GOTO

The `<goto>` element is used to;

- transition to another form item in the current form,
- transition to another dialog in the current document, or
- transition to another document.

To transition to another form item, use the nextitem attribute, or the expritem attribute if the form item name is computed using an ECMAScript expression:

```
<goto nextitem="ssn_confirm"/>
<goto expritem="(type==12)? 'ssn_confirm' : 'reject'"/>
```

To go to another dialog in the same document, use next (or expr) with only a URI fragment:

```
<goto next="#another_dialog"/>
<goto expr="'#' + 'another_dialog'"/>
```

To transition to another document, use next (or expr) with a URI:

```
<goto next="http://flight.example/reserve_seat"/>
<goto next="./special_lunch/#wants_vegan"/>
```

The URI may be absolute or relative to the current document. You may specify the starting dialog in the next document using a fragment that corresponds to the value of the id attribute of a dialog. If no fragment is specified, the first dialog in that document is chosen.

Note that transitioning to another dialog in the current document causes the old dialog's variables to be lost, even in the case where a dialog is transitioning to itself. Transitioning to another document will likewise drop the old document level variables, even if the new document is the same one that is making the transition. If you want data to persist across multiple documents, store data in the application scope.

Attributes of <goto> are:

next	The URI to which to transition.
expr	An ECMAScript expression that yields the URI.
nextitem	The name of the next form item to visit in the current form.
expritem	An ECMAScript expression that yields the name of the next form item to visit.
caching	See Section 12.1.
fetchaudio	See Section 12.1.
fetchhint	See Section 12.1. This defaults to the documentfetchhint property.
fetchtimeout	See Section 12.1.

Exactly one of next, expr, nextitem, or expritem must be specified.

19.8 SUBMIT

The <submit> element is similar to <goto> in that it results in a new document being obtained. Unlike <goto>, it lets you submit a list of variables to the document server via an HTTP GET or POST request. For example, to submit a set of form items to the server you might have:

```
<submit next="log_request" method="post" namelist="name rank serial_number"
        fetchtimeout="100s" fetchaudio="audio/brahms2.wav"/>
```

Attributes of `<submit>` include:

`next`	The URI to which the query is submitted.
`expr`	Like `next`, except that the URI is dynamically determined by evaluating the given ECMAScript expression. One of `next` or `expr` is required.
`namelist`	The list of variables to submit. By default, all the named field item variables are submitted. If a `namelist` is supplied, it may contain individual variable references which are submitted with the same qualification used in the namelist.
`method`	The request method: `get` (the default) or `post`.
`enctype`	The MIME encoding type of the submitted document. The default is `application/x-www-form-urlencoded`. Interpreters may support additional encoding types.
`caching`	See Section 12.1.
`fetchaudio`	See Section 12.1.
`fetchhint`	See Section 12.1. This defaults to the `documentfetchhint` property.
`fetchtimeout`	See Section 12.1.

If an ECMAScript object o is the target of a submit then all its (ECMAScript) fields f1, f2, ... are submitted using the names o.f1, o.f2, etc.

19.9 EXIT

Returns control to the interpreter context which determines what to do next.

```
<exit/>
```

This element differs from `<return>` in that it terminates all loaded documents, while `<return>` returns from a `<subdialog>` invocation. If the `<subdialog>` caused a new document (or application) to be invoked, then `<return>` will cause that document to be terminated, but execution will resume after the `<subdialog>`.

Note that once `<exit>` returns control to the interpreter context, the interpreter context is free to do as it wishes. It may play a top level menu for the user, drop the call, or transfer the user to an operator, for example.

Attributes include:

`expr`	A return expression (e.g. "`0`", or "`oops!`").
`namelist`	Variable names to be returned to interpreter context. The default is to return no variables; this means the interpreter context will receive an empty ECMAScript object.

19.10 RETURN

Return ends execution of a subdialog and returns control and data to a calling dialog. The attributes are:

event	Return, then throw this event.
namelist	Variable names to be returned to calling dialog. The default is to return no variables; this means the caller will receive an empty ECMAScript object.

In returning from a subdialog, an event can be thrown at the invocation point, or data is returned as an ECMAScript object. A return element that is encountered when not executing as a subdialog throws a semantic error. The example below shows an event propagated from a subdialog to its calling dialog when the subdialog fails to obtain a recognizable result. It also shows data returned under normal conditions.

Form with calling dialog

```
<form>
  <subdialog name="result" src="#getssn">
    <nomatch>
      <!-- a no match event that is returned by the subdialog indicates
          that a valid social security number could not be matched. -->
      <goto next="http://myservice.example/ssn-problems.vxml"/>
    </nomatch>
    <filled>
      <submit namelist="result.ssn"
          next="http://myservice.example/cgi-bin/process"/>
    </filled>
  </subdialog>
</form>
```

Subdialog to get social security number

```
<form id="getssn">
  <field name="ssn">
    <grammar src="http://grammarlib/ssn.gram"  type="application/x-jsgf"/>
    <prompt> Please say social security number. </prompt>
    <nomatch count=3>
      <return event="nomatch"/>
    </nomatch>
    <filled>
      <return namelist="ssn"/>
    </filled>
  </field>
</form>
```

The subdialog event handler for <nomatch> is triggered on the third failure to match; when triggered, it returns from the subdialog, and includes the nomatch event to be thrown in the context of the calling dialog. In this case, the calling dialog will execute its <nomatch> handler, rather than the <filled> element, where the resulting action is to execute a <goto> element. Under normal conditions, the <filled> element of the subdialog is executed after a recognized social security number is obtained, and then this value is returned to the calling dialog, and is accessible as result.ssn.

19.11 DISCONNECT

Causes the interpreter context to disconnect from the user. As a result, the interpreter context will throw a telephone.disconnected.hangup event, which may be caught to do cleanup processing, e.g.

```
<disconnect/>
```

A <disconnect> differs from an <exit> in that it forces the interpreter context to drop the call.

19.12 SCRIPT

The <script> element allows the specification of a block of client-side scripting language code, and is analogous to the HTML <SCRIPT> element. For example, this document has a script that computes a factorial.

```
<?xml version="1.0"?>
<vxml version="1.0"?>
    <script> <![CDATA[
        function factorial(n) {  return (n <= 1)? 1 : n * factorial(n-1); }
    ]]> </script>
    <form id="form">
        <field name="fact" type="number">
            <prompt>Tell me a number and I'll tell you its factorial.</prompt>
            <filled>
                <prompt>
                    <value expr="fact"/> factorial is
                    <value expr="factorial(fact)"/>
                </prompt>
            </filled>
        </field>
    </form>
</vxml>
```

A <script> element may occur in the <vxml> element, or in executable content (in <filled>, <if>, <block>, <catch>, or the short forms of <catch>). Scripts in the <vxml> element are evaluated just after the document is loaded, along with the <var> elements, in document order. A <script> element in executable content is executed, like other executable elements, as it is encountered.

The <script> element has the following attributes:

src	The URI specifying the location of the script, if it is external.
charset	The character encoding of the script designated by src.
caching	See Section 12.1.
fetchhint	See Section 12.1. This defaults to the scriptfetchhint property.
fetchtimeout	See Section 12.1.

Each <script> element is executed in the scope of its containing element; i.e., it does not have its own scope.

Here is a time-telling service with a block containing a script that initializes time variables in the dialog scope of a form:

```
<?xml version="1.0"?>
<vxml version="1.0">
    <form>
        <var name="hours"/>
        <var name="minutes"/>
        <var name="seconds"/>
        <block>
            <script>
                var d = new Date();
                hours = d.getHours();
                minutes = d.getMinutes();
                seconds = d.getSeconds();
            </script>
        </block>
        <field name="hear_another" type="boolean">
            <prompt>
                The time is <value expr="hours"/> hours,
                <value expr="minutes"/> minutes, and
                <value expr="seconds"/> seconds.
            </prompt>
            <prompt>Do you want to hear another time?</prompt>
            <filled>
                <if cond="hear_another">
                    <clear/>
                </if>
             </filled>
        </field>
    </form>
</vxml>
```

The ECMAScript scope chain (see section 10.1.4 in http://www.ecma.ch/stand/ECMA-262.htm) is set up so that variables declared with <var> are put into the scope associated with the element in which the <var> element occurs. All variables must be declared before being assigned or referenced by ECMAScript scripts, or by VoiceXML elements.

20 TIME DESIGNATIONS

Time designations follow those used in W3C's Cascading Style Sheet recommendation (http://www.w3.org/TR/REC-CSS2/syndata.html#q20). They consist of an unsigned integer followed by an optional time unit identifier. The time unit identifiers are:

- **ms**: milliseconds (the default)
- **s**: seconds

APPENDIX A. GLOSSARY OF TERMS

active grammar A speech or DTMF grammar that is currently active. This is based on the currently executing element, and the scope elements of the currently defined grammars.

application A collection of *VoiceXML documents* that are tagged with the same application name attribute.

ASR Automatic speech recognition.

author The creator of a *VoiceXML document*.

catch element A <catch> block or one of its abbreviated forms. Certain default catch elements are defined by the *VoiceXML interpreter*.

CSS W3C Cascading Style Sheet specification. See http://www.w3.org/TR/REC-CSS2

dialog An interaction with the user specified in a *VoiceXML document*. Types of dialogs include *forms* and *menus*.

ECMAScript A standard version of JavaScript backed by the European Computer Manufacturer's Association. See http://www.ecma.ch/stand/ECMA-262.htm

event A notification "thrown" by the *implementation platform*, *VoiceXML interpreter context*, *VoiceXML interpreter*, or VoiceXML code. Events include exceptional conditions (semantic errors), normal errors (user did not say something recognizable), normal events (user wants to exit), and user defined events.

executable content Procedural logic that occurs in <block>, <filled>, and *event handlers*.

field item A *form item* whose purpose is to input a field item variable. Field items include <field>, <record>, <object>, <subdialog>, and <transfer>.

form A *dialog* that interacts with the *user* in a highly flexible fashion with the computer and the *user* sharing the initiative.

form item An element of <form> that can be visited during form execution: <initial>, <block>, <field>, <record>, <object>, and <transfer>.

form item variable A variable, either implicitly or explicitly defined, associated with each *form item* in a *form*. If the form item variable is undefined, the form interpretation algorithm will visit the form item and use it to interact with the user.

implementation platform A computer with the requisite software and/or hardware to support the types of interaction defined by VoiceXML.

link A set of grammars that when matched by something the *user* says or keys in, either transitions to a new dialog or document or throws an event in the current form item.

menu A *dialog* presenting the *user* with a set of choices and takes action on the selected one.

mixed initiative A computer-human interaction in which either the computer or the human can take initiative and decide what to do next.

JSGF Java™ API Speech Grammar Format. A proposed standard for representing speech grammars. See http://www.javasoft.com/products/java-media/speech/forDevelopers/JSGF

JSML Java™ API Speech Markup Language. A proposed standard for speech markups. See http://www.javasoft.com/products/java-media/speech/forDevelopers/JSML

object A platform-specific capability with an interface available via VoiceXML.

request A collection of data including: a URI specifying a document server for the data, a set of name-value pairs of data to be processed (optional), and a method of submission for processing (optional).

SABLE A consortium seeking to develop standards for speech markup. See http://www.bell-labs.com/project/tts/sable.html

script A fragment of logic written in a client-side scripting language, especially *ECMAScript*, which is a scripting language that must be supported by any *VoiceXML interpreter*.

session A connection between a *user* and an *implementation platform*, e.g. a telephone call to a voice response system. One session may involve the interpretation of more than one *VoiceXML document*.

subdialog A VoiceXML dialog (or document) invoked from the current *dialog* in a manner analogous to function calls.

tapered prompts A set of prompts used to vary a message given to the human. Prompts may be tapered to be more terse with use (field prompting), or more explicit (help prompts).

throw An element that fires an *event*.

TTS Text-To-Speech; speech synthesis.

user A person whose interaction with an *implementation platform* is controlled by a *VoiceXML interpreter*.

URI Uniform Resource Indicator.

URL Uniform Resource Locator.

VoiceXML document An XML document conforming to the VoiceXML specification.

VoiceXML interpreter A computer program that interprets a *VoiceXML document* to control an *implementation platform* for the purpose of conducting an interaction with a *user*.

VoiceXML interpreter context A computer program that uses a *VoiceXML interpreter* to interpret a *VoiceXML Document* and that may also interact with the *implementation platform* independently of the *VoiceXML interpreter*.

W3C World Wide Web Consortium http://www.w3.org/

APPENDIX B. VOICEXML DOCUMENT TYPE DEFINITION

```
<!-- A DTD for Voice Extensible Markup Language -->
<!-- Copyright (c) 2000 VoiceXML Forum (AT&T, IBM, Lucent Technologies, Motorola) -->

<!ENTITY % audio
       "#PCDATA | audio | enumerate | value" >

<!ENTITY % boolean   "(true|false)" >

<!ENTITY % content.type "CDATA">

<!ENTITY % duration "CDATA" >

<!ENTITY % event.handler "catch | help | noinput | nomatch | error" >

<!ENTITY % event.name "NMTOKEN" >

<!ENTITY % event.names "NMTOKENS" >

<!ENTITY % executable.content
       "%audio; | assign | clear | disconnect | exit | goto | if | prompt |
       reprompt | return | script | submit | throw | var " >

<!ENTITY % expression "CDATA" >

<!ENTITY % field.name "NMTOKEN" >

<!ENTITY % field.names "NMTOKENS" >

<!ENTITY % integer "CDATA" >

<!ENTITY % item.attrs
       "name         %field.name;  #IMPLIED
       cond          %expression;  #IMPLIED
       expr          %expression;  #IMPLIED " >

<!ENTITY % uri "CDATA" >

<!ENTITY % cache.attrs
       "caching      (safe|fast)   'fast'
       fetchhint     (prefetch|safe|stream)'safe'
       fetchtimeout  %duration;    #IMPLIED " >

<!ENTITY % next.attrs
       "next         %uri;         #IMPLIED
       expr          %expression;  #IMPLIED " >

<!ENTITY % submit.attrs
       "method       (get|post)    'get'
       enctype       %content.type;'application/x-www-formurlencoded'
       namelist      %field.names; #IMPLIED" >

<!ENTITY % tts       "break | div | emp | pros | sayas" >

<!ENTITY % variable  "block | field | var" >

<!--================================== Root ==================================-->

<!ELEMENT vxml
       (%event.handler; | form | link | menu | meta |
       property | script | var)+ >
<!ATTLIST vxml
       application   %uri;         #IMPLIED
       base          %uri;         #IMPLIED
       lang          CDATA         #IMPLIED
       version       CDATA         #REQUIRED >

<!ELEMENT meta       EMPTY >
<!ATTLIST meta
       name          NMTOKEN       #IMPLIED
       content       CDATA         #REQUIRED
       http-equiv    NMTOKEN       #IMPLIED >

<!--================================== Dialogs ==============================-->

<!ENTITY % input     "dtmf | grammar" >
```

```
<!ENTITY % scope     "(document | dialog)" >

<!ELEMENT form
       (%input; | %event.handler; | filled | initial | object | link | property |
       record | subdialog | transfer | %variable;)* >
<!ATTLIST form
       id            ID            #IMPLIED
       scope         %scope;       'dialog' >

<!ELEMENT menu
       (%audio; | choice | %event.handler; | prompt | property)* >
<!ATTLIST menu
       id            ID            #IMPLIED
       scope         %scope;       'dialog'
       dtmf          %boolean;     'false' >

<!ELEMENT choice     (%audio; | grammar | %tts;)* >
<!ATTLIST choice
       %cache.attrs;
       dtmf          CDATA         #IMPLIED
       event         %event.name;  #IMPLIED
       fetchaudio    %uri;         #IMPLIED
       %next.attrs; >

<!--=============================== Prompts ==============================-->

<!ELEMENT prompt     (%audio; | %tts;)* >
<!ATTLIST prompt
       bargein       %boolean;     'true'
       cond          %expression;  #IMPLIED
       count         %integer;     #IMPLIED
       timeout       %duration;    #IMPLIED >

<!ELEMENT enumerate  (%audio; | %tts;)*>

<!ELEMENT reprompt   EMPTY >

<!--=============================== Fields ===============================-->

<!ENTITY % field.type
       "(boolean | date | digits | currency | number | phone | time)" >

<!ELEMENT field
       (%audio; | %event.handler; | filled | %input; | link | option | prompt | property)* >
<!ATTLIST field
       %item.attrs;
       type          %field.type;  #IMPLIED
       slot          NMTOKEN       #IMPLIED
       modal         %boolean;     'false' >

<!ELEMENT option     (#PCDATA)* >
<!ATTLIST option
       dtmf          CDATA         #IMPLIED
       value         CDATA         #IMPLIED >

<!ELEMENT var        EMPTY >
<!ATTLIST var
       name          %field.name;  #REQUIRED
       expr          %expression;  #IMPLIED >

<!ELEMENT initial    (%audio; | %event.handler; | link | prompt | property)* >
<!ATTLIST initial
       %item.attrs; >

<!ELEMENT block      (%executable.content;)* >
<!ATTLIST block
       %item.attrs; >

<!ELEMENT assign     EMPTY >
<!ATTLIST assign
       name          %field.name;  #REQUIRED
       expr          %expression;  #REQUIRED >

<!ELEMENT clear      EMPTY >
<!ATTLIST clear
       namelist      %field.names; #IMPLIED >

<!ELEMENT value      EMPTY >
<!ATTLIST value
       class         CDATA         #IMPLIED
       expr          %expression;  #REQUIRED
       mode          (tts|recorded)"tts"
       recsrc        %uri;         #IMPLIED >

<!--================================== Events ==============================-->

<!ENTITY % event.handler.attrs
       "count        %integer;     #IMPLIED
       cond          %expression;  #IMPLIED" >

<!ELEMENT catch      (%executable.content;)* >
<!ATTLIST catch
       event         %event.names; #REQUIRED
       %event.handler.attrs; >

<!ELEMENT error      (%executable.content;)* >
<!ATTLIST error
       %event.handler.attrs; >

<!ELEMENT help       (%executable.content;)* >
<!ATTLIST help
       %event.handler.attrs; >

<!ELEMENT link       (dtmf | grammar)* >
<!ATTLIST link
       %cache.attrs;
       %next.attrs;
       fetchaudio    %uri;         #IMPLIED
       event         %event.name;  #IMPLIED >

<!ELEMENT noinput    (%executable.content;)* >
<!ATTLIST noinput
       %event.handler.attrs; >

<!ELEMENT nomatch    (%executable.content;)* >
<!ATTLIST nomatch
       %event.handler.attrs; >

<!ELEMENT throw      EMPTY >
<!ATTLIST throw
       event         %event.name;  #REQUIRED >


<!--============================== Audio Output ============================-->

<!ELEMENT audio      (%audio; | %tts;)* >
<!ATTLIST audio
       src           %uri;         #IMPLIED
       %cache.attrs; >

<!ELEMENT break      EMPTY >
<!ATTLIST break
       msecs         %integer;     #IMPLIED
       size          (none|small|medium|large) #IMPLIED >

<!ELEMENT div        (%audio; | %tts;)* >
<!ATTLIST div
       type          CDATA  #IMPLIED>

<!ELEMENT emp        (%audio; | %tts;)* >
<!ATTLIST emp
       level         (strong | moderate | none | reduced) "moderate" >

<!ELEMENT pros       (%audio; | %tts;)* >
<!ATTLIST pros
       rate          CDATA         #IMPLIED
       vol           CDATA         #IMPLIED
       pitch         CDATA         #IMPLIED
       range         CDATA         #IMPLIED >

<!ELEMENT sayas      (#PCDATA)* >
<!ATTLIST sayas
       sub           CDATA         #IMPLIED
       class         CDATA         #IMPLIED
       phon          CDATA         #IMPLIED >


<!--============================= Audio Input ==============================-->

<!ENTITY % key       "CDATA" >
```

```
<!ENTITY % grammar.attrs
       "%cache.attrs;
       scope         %scope;       #IMPLIED
       src           %uri;         #IMPLIED
       type          CDATA         #IMPLIED " >

<!ELEMENT dtmf       (#PCDATA)* >
<!ATTLIST dtmf
       %grammar.attrs; >

<!ELEMENT grammar    (#PCDATA)* >
<!ATTLIST grammar
       %grammar.attrs; >

<!ELEMENT record
       (%audio; | %event.handler; | filled | grammar | prompt | property)* >
<!ATTLIST record
       %item.attrs;
       type          CDATA         #IMPLIED
       beep          %boolean;     'false'
       maxtime       %duration;    #IMPLIED
       modal         %boolean;     'true'
       finalsilence  %duration;    #IMPLIED
       dtmfterm      %boolean;     'true'  >

<!--============================ Call Control =============================-->

<!ELEMENT disconnect EMPTY >

<!ELEMENT transfer
       (%audio; | %event.handler; | dtmf | filled | grammar | prompt | property)* >
<!ATTLIST transfer
       %item.attrs;
       dest          %uri;         #IMPLIED
       destexpr      %expression;  #IMPLIED
       bridge        %boolean;     'false'
       connecttimeout%duration;    #IMPLIED
       maxtime       %duration;    #IMPLIED >

<!--============================ Control Flow =============================-->

<!ENTITY % if.attrs
       "cond         %expression;  #REQUIRED" >

<!ELEMENT if         (%executable.content; | elseif | else)* >
<!ATTLIST if
       %if.attrs; >

<!ELEMENT elseif     EMPTY >
<!ATTLIST elseif
       %if.attrs; >

<!ELEMENT else       EMPTY >

<!ELEMENT exit       EMPTY >
<!ATTLIST exit
       expr          %expression;  #IMPLIED
       namelist      %field.names; #IMPLIED >

<!ELEMENT filled     (%executable.content;)* >
<!ATTLIST filled
       mode          (any|all)     "all"
       namelist      %field.names; #IMPLIED >

<!ELEMENT goto       EMPTY >
<!ATTLIST goto
       %cache.attrs;
       %next.attrs;
       fetchaudio    %uri;         #IMPLIED
       expritem      %expression;  #IMPLIED
       nextitem      %field.name;  #IMPLIED >

<!ELEMENT param      EMPTY >
<!ATTLIST param
       name          NMTOKEN       #REQUIRED
       expr          %expression;  #IMPLIED
       value         CDATA         #IMPLIED
       valuetype     (data|ref)    'data'
       type          CDATA         #IMPLIED >
```

```
<!ELEMENT return     EMPTY >
<!ATTLIST return
       namelist      %field.names; #IMPLIED
       event         %event.name;  #IMPLIED >

<!ELEMENT subdialog
       (%audio; | %event.handler; | filled | param | prompt | property)* >
<!ATTLIST subdialog
       %item.attrs;
       src           %uri;         #REQUIRED
       %cache.attrs;
       fetchaudio    %uri;         #IMPLIED
       %submit.attrs; >

<!ELEMENT submit     EMPTY >
<!ATTLIST submit
       %cache.attrs;
       %next.attrs;
       fetchaudio    %uri;         #IMPLIED
       %submit.attrs; >

<!--========================== Miscellaneous ==============================-->

<!ELEMENT object
       (%audio; | %event.handler; | filled | param | prompt | property)* >
<!ATTLIST object
       %item.attrs;
       %cache.attrs;
       classid       %uri;         #IMPLIED
       codebase      %uri;         #IMPLIED
       data          %uri;         #IMPLIED
       type          CDATA         #IMPLIED
       codetype      CDATA         #IMPLIED
       archive       %uri;         #IMPLIED >

<!ELEMENT property   EMPTY >
<!ATTLIST property
       name          NMTOKEN       #REQUIRED
       value         CDATA         #REQUIRED >

<!ELEMENT script     (#PCDATA)* >
<!ATTLIST script
       src           %uri;         #IMPLIED
       charset       CDATA         #IMPLIED
       %cache.attrs; >
```

APPENDIX C. FORM INTERPRETATION ALGORITHM

The form interpretation algorithm (FIA) drives the interaction between the user and a VoiceXML form or menu. A menu can be viewed as a form containing a single field whose grammar and whose <filled> action are constructed from the <choice> elements.

The FIA must handle:

- Form initialization.
- Prompting, including the management of the prompt counters needed for prompt tapering.
- Grammar activation and deactivation at the form and form item levels.
- Entering the form with an utterance that matched one of the form's document-scoped grammars while the user was visiting a different form or menu.
- Leaving the form because the user matched another form, menu, or link's document-scoped grammar.
- Processing multiple field fills from one utterance, including the execution of the relevant <filled> actions.
- Selecting the next form item to visit, and then processing that form item.
- Choosing the correct catch element to handle any events thrown while processing a form item.

First we define some terms and data structures used in the form interpretation algorithm:

active grammar set	The set of grammars active during a VoiceXML interpreter context's input collection operation.
utterance	A summary of what the user said or keyed in, including the specific grammar matched, and a dictionary of slot name/slot value pairs. An example utterance might be: "grammar 123 was matched, and the slots are from_city = 'chicago', to_city = 'new orleans', and flight_num = 2233".
execute	To execute executable content – either a block, a filled action, or a set of filled actions. If an event is thrown during execution, the execution of the executable content is aborted. The appropriate event handler is then executed, and this may cause control to resume in a form item, in the next iteration of the form's main loop, or outside of the form. If a <goto> is executed, the transfer takes place immediately, and the remaining executable content is not executed.

Here is the conceptual form interpretation algorithm. The FIA can start with no initial utterance, or with an initial utterance passed in from another dialog:

```
//
// Initialization Phase
//

foreach ( <var> and form item variable, in document order )
    Declare the variable, initializing it to the value of the "expr" attribute, if
      any, or else to undefined.
foreach ( field item )
```

```
      Declare a prompt counter and set it to 1.
   if ( there is an initial item )
      Declare a prompt counter and set it to 1.
   if ( user entered form by speaking to its grammar while in a different form )
   {
      Enter the main loop below, but start in the process phase, not the select
        phase: we already have a collection to process.
   }


   //
   // Main Loop: select next form item and execute it.
   //

   while ( true )
   {
      //
      // Select Phase: choose a form item to visit.
      //

      if ( the last main loop iteration ended with a <goto nextitem> )
         Select that next form item.
      else if (there is a form item with an unsatisfied guard condition )
         Select the first such form item in document order.
      else
         Do an <exit/> -- the form is full and specified no transition.


      //
      // Collect Phase: execute the selected form item.
      //

      // Queue up prompts for the form item.
      unless ( the last loop iteration ended with a catch that had no <reprompt> )
      {
         Select the appropriate prompts for the form item.
         Queue the selected prompts for play prior to the next collect operation.
         Increment the form item’s prompt counter.
      }

      // Activate grammars for the form item.
      if ( the form item is modal )
         Set the active grammar set to the form item grammars, if any. (Note that
           some form items, e.g. <block>, cannot have any grammars).
      else
         Set the active grammar set to the form item grammars and any grammars
           scoped to the form, the current document, the application root
           document, and then elements up the <subdialog> call chain.

      // Execute the form item.
      if ( a <field> was selected )
         Collect an utterance or an event from the user.
      else if ( a <record> was chosen )
         Collect an utterance (with a name/value pair for the recorded bytes) or
           event from the user.
      else if ( an <object> was chosen )
         Execute the object, setting the <object>’s form item variable to the
           returned ECMAScript value.
      else if ( a <subdialog> was chosen )
         Execute the subdialog, setting the <subdialog>’s form item variable to the
           returned ECMAScript value.
      else if ( a <transfer> was chosen )
         Do the transfer, and (if wait is true) set the <transfer> form item variable
           to the returned result status indicator.
      else if ( the <initial> was chosen )
         Collect an utterance or an event from the user.
      else if ( a <block> was chosen )
      {
         Set the block’s form item variable to a defined value.
         Execute the block’s executable context.
      }

      //
      // Process Phase: process the resulting utterance or event.
      //

      // Process an event.
      if ( the form item execution resulted in an event )
      {
         Find the appropriate catch for the event.
         Execute the catch (this may leave the FIA).
         continue
      }

      // Must have an utterance: process ones from outside grammars.
      if ( the utterance matched a grammar from outside the form )
      {
         if ( the grammar belongs to a <link> element )
             Execute that link’s goto or throw, leaving the FIA.
         if ( the grammar belongs to a menu’s <choice> element )
             Execute the choice’s goto or throw, leaving the FIA.
         // The grammar belongs to another form (or menu).
         Transition to that form (or menu), carrying the utterance to
           the other form (or menu)’s FIA.
      }

      // Process an utterance spoken to a grammar from this form.
      // First copy utterance slot values into corresponding form item variables.
      Clear all “just_filled” flags.
      foreach ( slot in the user’s utterance )
      {
         if ( the slot corresponds to a field item )
         {
             Copy the slot value into the field item’s form item variable.
             Set this field item’s “just_filled” flag.
         }
      }

      // Set <initial> form item variable if any field items are filled.
      if ( any field item variable is set as a result of the user utterance )
         Set the <initial> form item variable.

      // Next execute any <filled> actions triggered by this utterance.
      foreach ( <filled> action in document order )
      {
         // Determine the form item variables the <filled> applies to.
         N = the <filled>’s “namelist” attribute.
         if ( N equals “” )
         {
             if ( the <filled> is a child of a form item )
                 N = the form item’s form item variable name.
             else if ( the <filled> is a child of a form )
                 N = the form item variable names of all the form items in that form.
         }

         // Is the <filled> triggered?
```

```
          if ( any form item variable in the set N was “just_filled”
               AND  (   the <filled> mode is “all” AND all variables in N are filled
                     OR the <filled> mode is “any” AND any variables in N are filled))
               Execute the <filled> action.
      }
   }
```

APPENDIX D. JSGF AS A VOICEXML GRAMMAR FORMAT

In this section we will describe how the Java™ Speech Grammar Format (JSGF) can be used with VoiceXML <grammar> element.

As stated in the section on grammars, a VoiceXML grammar must:

- specify a set of utterances that a user may speak to perform an action or supply information, and
- provide a corresponding string value (in the case of a field grammar) or set of attribute-value pairs (in the case of a form grammar) to describe the information or action.

JSGF supports the first requirement above by providing a language for describing *context-free grammars*. The following table is a summary of the features of JSGF.

Feature	Purpose
word or "word"	words (terminals, tokens) need not be quoted
<rule>	rule names (non-terminals) are enclosed in <>
[x]	optionally x
(...)	Grouping
x {tag text}	arbitrary "tag" text may be associated with any of the above
x*	0 or more occurrences of x
x+	1 or more occurrences of x
x y z ...	a sequence of x then y then z then ...
x | y | z | ...	a set of alternatives of x or y or z or ...
<rule> = x; public <rule> = x;	a private and a public rule definition

The JSGF tag facility provides a means for meeting the second requirement of providing values for forms to describe the action requested. In the case of field grammars, where only a single string value is needed, a tag may be used to supply the value. If no tag is specified, the text of the utterance itself is used as the value.

As described in the section on grammars, a grammar element be either *inline* or *external.* Furthermore, in the case of JSGF, an inline grammar may be either a *grammar fragment* or *complete grammar*. These three cases are described below.

Inline grammar fragment. The content of the <grammar> element is the right-hand-side of a JSGF rule. (In JSGF terminology this is called a "rule expansion"). In the most common case, where no reference to non-terminals is made, no use is made of the XML reserved special characters, and so the rule expansion may be specified inline without need for quoting or use of a PCDATA element. This form is thus particularly convenient for expressing simple lists of alternative ways of saying the same thing, for example:

```
<link event="help">
    <grammar type="application/x-jsgf">
        [please] help [me] [please] | [please] I (need|want) help [please]
    </grammar>
</link>
<field name="sandwich">
    <grammar type="application/x-jsgf">
```

```
            hamburger | burger {hamburger} | (chicken [sandwich]) {chicken}
        </grammar>
    </field>
```

In the first example, any of the ways of saying "help" result in a help event being thrown. In the second example, the user may say "hamburger" or "burger" and the "sandwich" field will be given the value "hamburger", or the user may say "chicken" or "chicken sandwich" and the "sandwich" field will be given the value "chicken".

Inline complete grammar. The content of the <grammar> element is a complete JSGF grammar, consisting of one or more rule definitions, with possible reference to external grammars. In this case all public rules in the supplied grammar are used. Since this form requires the use of XML reserved special characters generally a PCDATA element will be needed.

External grammar. A complete JSGF grammar is found at the URI specified by the src attribute of the grammar element; the <grammar> element content must be empty. The specified URI may take the form of

- a URI naming a whole document, in which case all public rules in the grammar contained in the document at the specified URI are used, or
- a URI naming a document fragment, that is, a URI ending with #*fragment*, in which case the *fragment* name is taken to be the name of a public rule from the grammar contained in the document at the specified URI; only the rule so named is used.

APPENDIX E. SUGGESTED AUDIO FILE FORMATS

VoiceXML recommends that a platform support the playing and recording audio formats specified below. Note: a platform need not support both A-law and µ-law simultaneously.

Audio Format	MIME Type
Raw (headerless) 8kHz 8-bit mu-law [PCM] single channel.	audio/basic (from http://ietf.org/rfc/rfc1521.txt)
Raw (headerless) 8kHz 8 bit A-law [PCM] single channel.	audio/x-alaw-basic
WAV (RIFF header) 8kHz 8-bit mu-law [PCM] single channel.	audio/wav
WAV (RIFF header) 8kHz 8-bit A-law [PCM] single channel.	audio/wav

APPENDIX F. TIMING PROPERTIES

The various timing properties for speech and DTMF recognition work together to define the user experience. The ways in which these different timing parameters function are outlined in the timing diagrams below. In these diagrams, the start for wait of DTMF input, or user speech both occur at the time that the last prompt has finished playing.

A.1. DTMF Grammars

DTMF grammars use `timeout`, `interdigittimeout`, `termtimeout` and `termchar` to tailor the user experience. The effects of these are shown in the following timing diagrams.

`timeout`, **No Input Provided**

The timeout parameter determines when the <`noinput`> event is thrown because the user has failed to enter any DTMF (Figure 5).

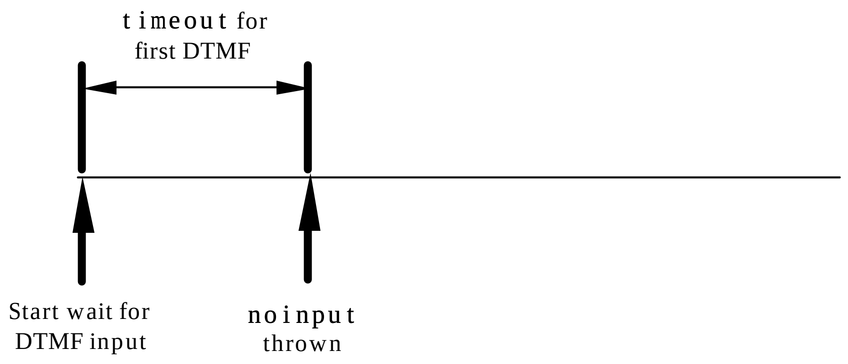

Figure 5 Timing diagram for `timeout` when no input provided.

`interdigittimeout`, **Grammar is Not Ready to Terminate**

In Figure 6, the `interdigittimeout` determines when the `nomatch` event is thrown because a DTMF grammar is not yet recognized, and the user has failed to enter additional DTMF.

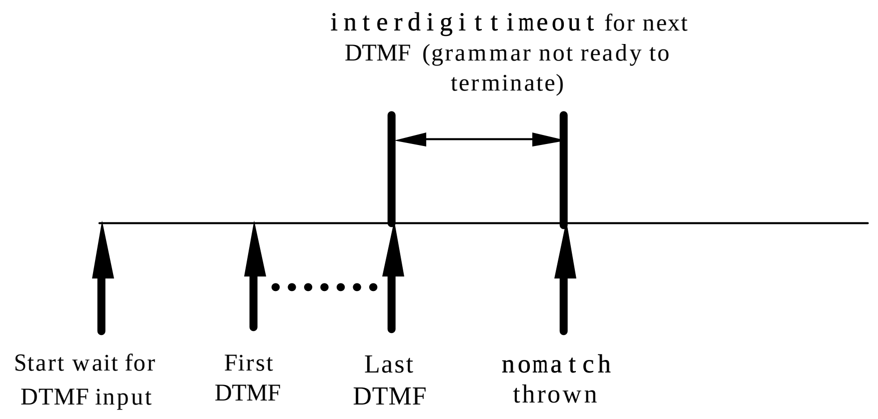
interdigittimeout for next DTMF (grammar not ready to terminate)
Start wait for DTMF input
First DTMF
Last DTMF
nomatch thrown

Figure 6 Timing diagram for `interdigittimeout`, grammar is not ready to terminate.

`interdigittimeout`, Grammar is Ready to Terminate

The example below shows the situation when a DTMF grammar could terminate, or extend by the addition of more DTMF input, and the user has elected not to provide any further input.

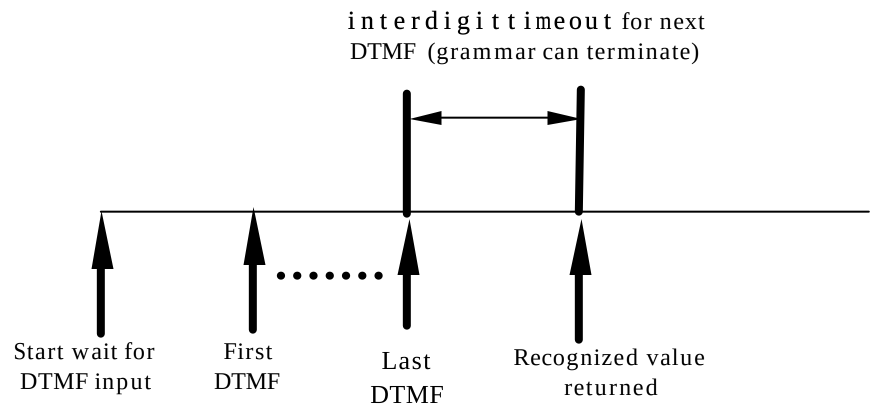
interdigittimeout for next DTMF (grammar can terminate)
Start wait for DTMF input
First DTMF
Last DTMF
Recognized value returned

Figure 7 Timing diagram for `interdigittimeout`, grammar is ready to terminate.

`termchar` and `interdigittimeout`, Grammar Can Terminate

In the example below, a `termchar` is non-empty, and is entered by the user before an `interdigittimeout` expires, to signify that the users DTMF input is complete; the `termchar` is not included as part of the recognized value.

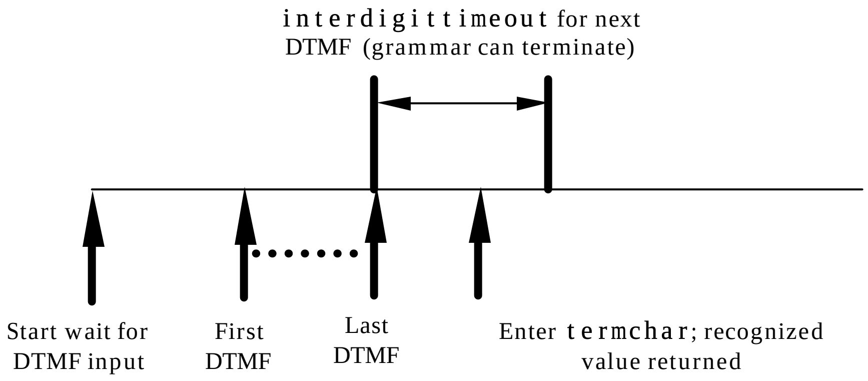
interdigittimeout for next DTMF (grammar can terminate)
Start wait for DTMF input
First DTMF
Last DTMF
Enter termchar; recognized value returned

Figure 8 Timing diagram for termchar and interdigittimeout, grammar can terminate.

termchar **Empty When Grammar Must Terminate**

In the example below, the entry of the last DTMF has brought the grammar to a termination point at which no additional DTMF is expected. Since termchar is empty, there is no optional terminating character permitted, thus the recognition ends and the recognized value is returned.

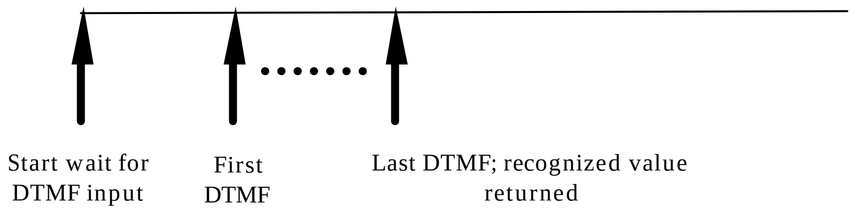
Start wait for DTMF input
First DTMF
Last DTMF; recognized value returned

Figure 9 Timing diagram for termchar empty when grammar must terminate.

termchar **Non-Empty and** termtimeout **When Grammar Must Terminate**

In the example below, the entry of the last DTMF has brought the grammar to a termination point at which no additional DTMF is allowed by the grammar. If the termchar is non-empty, then the user can enter an optional termchar DTMF. If the user fails to enter this optional DTMF within termtimeout, the recognition ends and the recognized value is returned. If the termtimeout is 0s (the default), then the recognized value is returned immediately after the last DTMF allowed by the grammar, without waiting for the optional termchar.

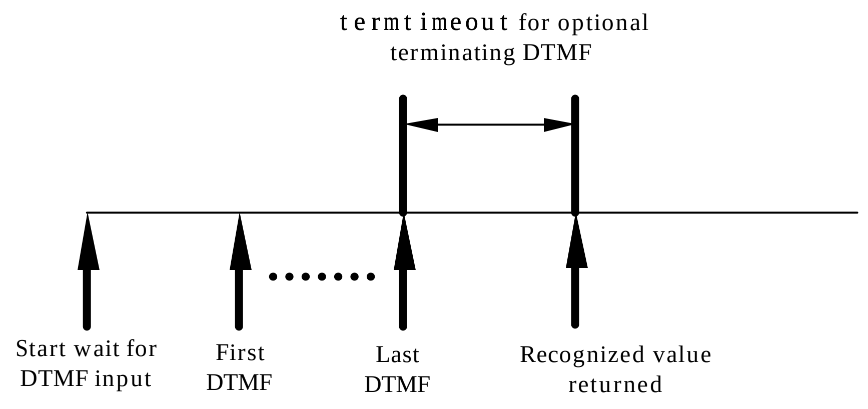
termtimeout for optional terminating DTMF
Start wait for DTMF input
First DTMF
Last DTMF
Recognized value returned

Figure 10 Timing diagram for termchar non-empty and termtimeout when grammar must terminate.

termchar Non-Empty and termtimeout When Grammar Must Terminate

In this last DTMF example, the entry of the last DTMF has brought the grammar to a termination point at which no additional DTMF is allowed by the grammar. Since the termchar is non-empty, the user enters the optional termchar within termtimeout causing the recognized value to be returned (excluding the termchar).

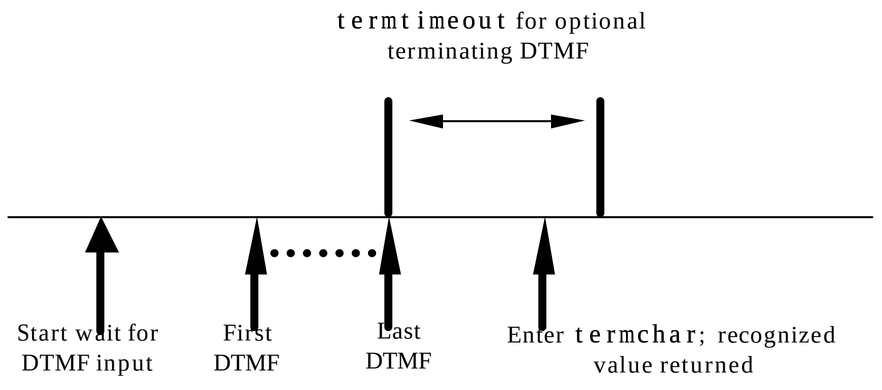
termtimeout for optional terminating DTMF
Start wait for DTMF input
First DTMF
Last DTMF
Enter termchar; recognized value returned

Figure 11 Timing diagram for termchar non-empty when grammar must terminate.

A.2. Speech Grammars.

Speech grammars use timeout, completetimeout, and incompletetimeout to tailor the user experience. The effects of these are shown in the following timing diagrams.

timeout When No Speech Provided

In the example below, the timeout parameter determines when the noinput event is thrown because the user has failed to speak.

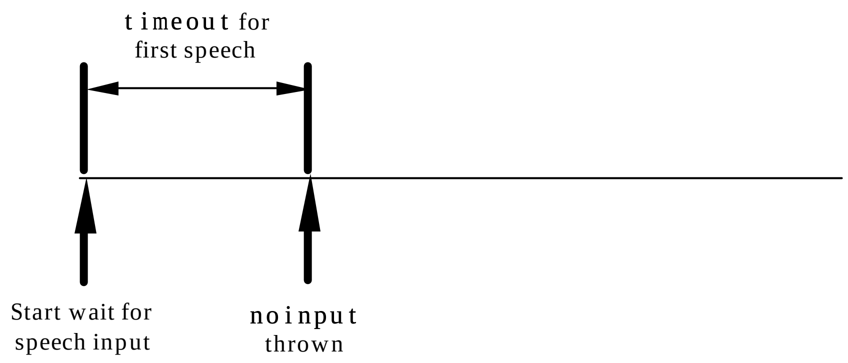

Figure 12 Timing diagram for timeout when no speech provided.

completetimeout With Speech Grammar Recognized

In the example above, the user provided a utterance that was recognized by the speech grammar. After a silence period of completetimeout has elapsed, the recognized value is returned.

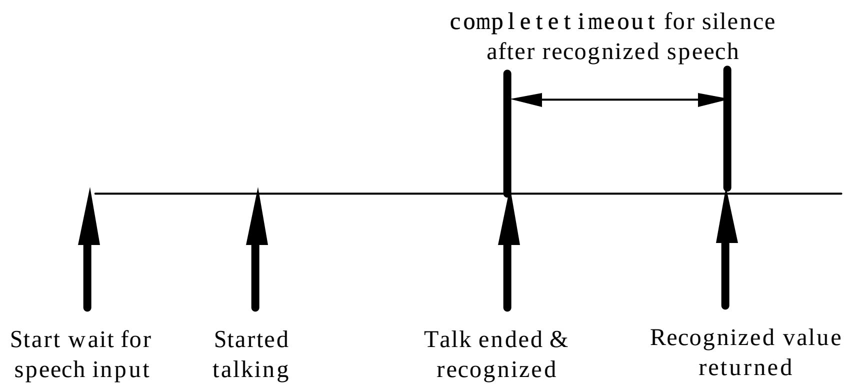
completetimeout for silence after recognized speech
Start wait for speech input
Started talking
Talk ended & recognized
Recognized value returned

Figure 13 Timing diagram for completetimeout with speech grammar recognized.

incompletetimeout with Speech Grammar Unrecognized

In the example above, the user provided a utterance that is not as yet recognized by the speech grammar but is the prefix of a legal utterance. After a silence period of incompletetimeout has elapsed, a nomatch event is thrown.

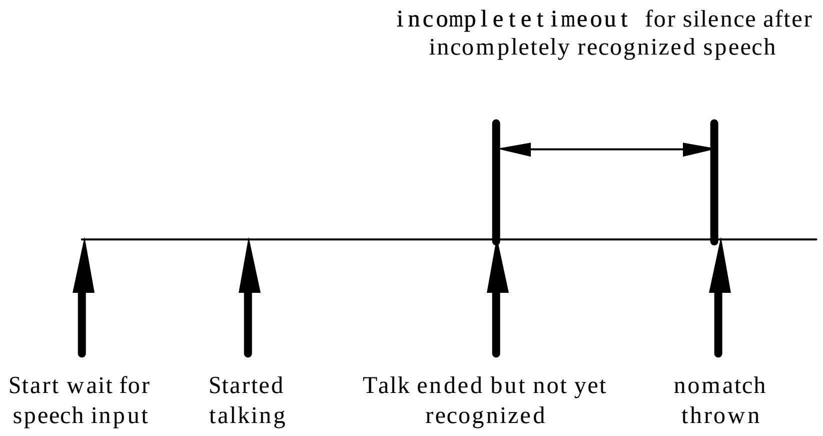
incompletetimeout for silence after incompletely recognized speech
Start wait for speech input
Started talking
Talk ended but not yet recognized
nomatch thrown

Figure 14 Timing diagram for `incompletetimeout` with speech grammar unrecognized.

APPENDIX G. PROPOSED EXTENSION: TRANSCRIBE

This is an example of extending VoiceXML with a new element.

The <transcribe> element is a field item that collects a transcription of the user's utterance. The transcription string is stored in the field item variable.

```
<?xml version="1.0"?>
<vxml version="1.0">
   <form>
      <transcribe name="message" beep="true" maxtime="30s">
         <prompt>What is the message you want to leave?</prompt>
         <nomatch count="2">
            Try to make your message simpler.
         </nomatch>
         <nomatch count="3">
            Transferring to operator.
            <goto next="queue_caller.pl"/>
         </nomatch>
      </transcribe>

      <field name="confirm" type="boolean">
         <prompt>Your message is <value expr="message"/>.</prompt>
         <prompt>To send it, say yes.  To discard it, say no.</prompt>
         <filled>
            <if cond="confirm">
               <submit next="send_page.pl" namelist="message"/>
            </if>
            <clear/>
         </filled>
      </field>
   </form>
</vxml>
```

Not all interpreter contexts will support <transcribe>.

The attributes of <transcribe> are:

name	The field item variable that will hold the transcription.
expr	The initial value of the form item variable; default is ECMAScript undefined. If initialized to a value, then the form item will not be visited unless the form item variable is cleared.
cond	A boolean condition that must also evaluate to true in order for the form item to be visited.
modal	If this is true (the default) all higher level speech and DTMF grammars are turned off while making the transcription. If false, speech and DTMF grammars scoped to the form, document, application, and calling documents are also listened for (if the implementation supports that).
beep	If true, a tone is emitted just prior to transcription. Defaults to false.
maxtime	The maximum duration to transcribe.
finalsilence	The interval of silence that indicates end of speech.
dtmfterm	If true, a DTMF keypress terminates transcription. Defaults to true. The DTMF tone is not part of the transcription.

The `<transcribe>` shadow variable (*name*$) has the following ECMAScript properties after the transcription has been made:

name`$.confidence`	The confidence level in the transcription from 0.0-1.0. A value of 0.0 indicates minimum confidence, and a value of 1.0 indicates maximum confidence. More specific interpretation of a confidence value is platform-dependent.
name`$.termchar`	If the `dtmfterm` attribute is `true`, and the user terminates the transcription by pressing a DTMF key, then this shadow variable is the key pressed (e.g. “#”). Otherwise it is `null`.
name`$.utterance`	The raw string of words that were recognized. The exact tokenization and spelling is platform-specific (e.g. “five hundred thirty” or “5 hundred 30” or even “530”). For example, the raw utterance might be “I need a hundred and twenty five dollars by tonight” for a final transcription of “I need $125 by tonight”.

The following are the required changes to the DTD:

```
<!--================================= Dialogs ==============================-->
...
<!ELEMENT form
       (%input; | %event.handler; | filled | initial | object | link | property |
       record | subdialog | transcribe | transfer | %variable;)* >
<!ATTLIST form
       id             ID             #IMPLIED
       scope          %scope;        'dialog' >

…
!--============================= Audio Input ==============================-->
…

<!ELEMENT transcribe
       (%audio; | %event.handler; | filled | grammar | prompt | property)* >
<!ATTLIST transcribe
       %item.attrs;
       beep           %boolean;      'false'
       maxtime        %duration;     #IMPLIED
       modal          %boolean;      'true'
       finalsilence   %duration;     #IMPLIED
       dtmfterm       %boolean;      'true'  >
…
```

The Form Interpretation Algorithm would be modified as follows:

```
   // Execute the form item.
      if ( a <field> was selected )
         Collect an utterance or an event from the user.
      else if ( a <record> was chosen )
         Collect an utterance (with a name/value pair for the recorded bytes) or
           event from the user.
      else if ( a <transcribe> was chosen )
         Collect an utterance (with a name/value pair for the transcription) or
           event from the user.
      else if ( an <object> was chosen )
         Execute the object, setting the <object>’s form item variable to the
           returned ECMAScript value.
      …
```