Voice Extensible Markup Language

VoiceXML

Version: 0.9
Date: 17 Aug 1999

About the VoiceXML Forum

The VoiceXML Forum is an industry organization founded by AT&T, IBM, Lucent and Motorola. It was established to promote the Voice eXtensible Markup Language (VoiceXML) which is a new standard essential to making 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 is uniquely positioned to make speech-enabled applications on the Internet a near-term reality.

For more information on the VoiceXML Forum please visit the website at http://www.voicexml.org/ which includes information on membership and the latest version of the specification.

Feedback

Comments to the technical working committee can be sent to submission@voicexml.org

A general mailing list for discussion is available. Address entries to discussion@voicexml.org. Information about joining this list can be found at the website at http://www.voicexml.org/.

Disclaimers

This is a draft document. It is subject to change without notice and may be updated, replaced or made obsolete by other documents at any time. The VoiceXML Forum will not allow early implementation to constrain its ability to make changes to this specification prior to final release.

THE VoiceXML Forum DISCLAIMS ANY AND ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING (WITHOUT LIMITATION) ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

The descriptions contained herein do not imply the granting of licenses to make, use, sell, license or otherwise transfer any technology required to implement systems or components conforming to this specification. The VoiceXML Forum, and its member companies, makes no representation on technology described in this specification regarding existing or future patent rights, copyrights, trademarks, trade secrets or other proprietary rights.

By submitting information to the VoiceXML Forum, and its member companies, including but not limited to technical information, you agree that the submitted information does not contain any confidential or proprietary information, and that the VoiceXML Forum may use the submitted information without any restrictions or limitations.

Revision History

Version	Date	Description
0.9	17 Aug 1999	Initial release. Provided as baseline in support of comment period from supporters.

Contents

1. INTRODUCTION

This document introduces VoiceXML, the Voice Extensible Markup Language. Its goal is to elicit feedback from the wider community, and thus is a language description, not a formal specification.

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>
  <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>
  <form>
    <field name="drink">
      <prompt>Would you like coffee, tea, milk, or nothing?</prompt>
      <grammar src="drink.gram"/>
    </field>
    <block>
      <goto next="http://www.drink.example/drink2.asp"
            submit="drink" method="get"/>
    </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

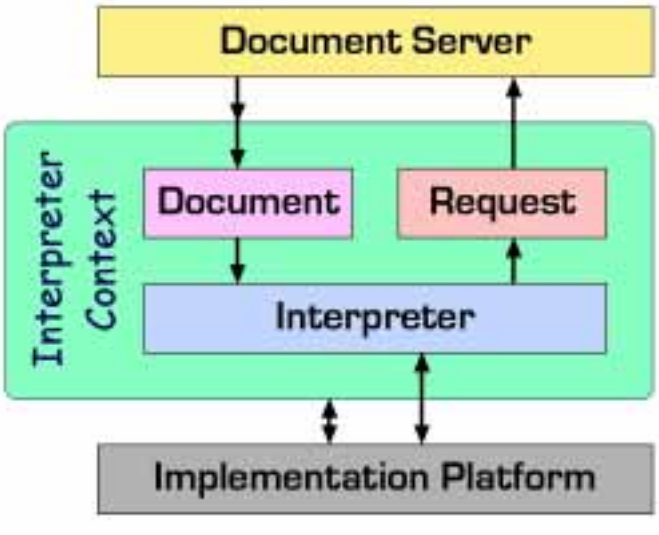
C: *(continues in document drink2.asp)*

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:



A *document server* (e.g., a web server) processes *requests* from a client application (called the *VoiceXML interpreter context*). The server produces *VoiceXML documents* in reply. These documents are read by a *VoiceXML interpreter*, which is inside a *VoiceXML interpreter context*. 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-speechcharacteristics.

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.*
- *Safely handles shared network-based applications. No arbitrary computations are allowed, and platform resources are protected.*
- *Is easy to use for simple interactions, and yet extensible for complex ones.*

While VoiceXML strives to accommodate the requirements of a majority of voice response services, services with stringent performance 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 the ability to:

- *Synthesized speech output (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). When a link is followed, request variables and their values, if present, are submitted to the link's URI.

2.4 Principles of Design

VoiceXML derives its lexical and syntactic conventions from XML. Please refer to the Annotated XML Reference Manual at: http://www.xml.com/axml/testaxml.htm for details.

1. The language promotes portability of service.
2. The language supports ease of authoring for common types of interactions.
3. 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.

4. The language describes voice-based human-machine interaction without depending on *active content*.
5. 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.
6. The language has a control flow mechanism.
7. The language enables a separation of service logic from interaction behavior.
8. It does not include complex computational capabilities, database operations, or legacy system operations. These are assumed to be handled by resources outside the document interpreter, e.g. a web server.
9. General service logic, state management, dialog generation, and dialog sequencing are assumed to reside outside the document interpreter.
10. The language provides ways to link documents using URIs, and also to submit data to server scripts using URIs.
11. 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.
12. 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 will be handled at the platform layer.
13. The language supports the tracking of information presented by the interpreter.

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 intermix the two types of audio freely. Audio files are referred to by a URI. The language does not specify a required set of audio file formats. For TTS, the speech markups expected to be in the emerging W3C TTS standards will be supported. (See http://www.bell-labs.com/project/tts/sable.html).

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 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.* (VoiceXML does not specify a required set of speech or DTMF grammar formats.)

- *It should be able to record audio received from the user. The duration of the recording interval is controllable by two timer values specified by a* VoiceXML document*: one that governs the maximum length of the recording, and one that indicates the maximum length of silence allowed before terminating recording. The platform must be capable of terminating the recording by expiration of either timer, or by character input. The* implementation platform *must be able to make the recording available to a* request *variable explicitly and by URI.* (The language does not specify a required set of audio file formats for recording.)

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

3.1 Dialogs

There are two kinds of dialogs. *Forms* define an interaction that collects values for a set of form variables. 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.

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 root document is also loaded. The root document remains loaded while the user is transitioning between other application documents, 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.

3.4 Grammars

Each dialog has one or more speech and/or DTMF (touch-tone) *grammars* associated with it. In simple *computer 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 computer 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 a surprising amount of flexibility and power to voice applications.

3.5 Links

A *link* is a special construct supporting mixed initiative: it appears at the top level of a document and specifies a bundle of grammars to listen for whenever the user is in that document (or that document's application). If the user says or keys in something matching a link, the link's destination URI is transitioned to. A link is thus a *transition pattern* that applies to all the dialogs in its scope, a much terser and more efficient alternative to specifying the same transition repeatedly.

3.6 Events

Events are thrown by the platform when the user does not speak, doesn't speak intelligibly, asks for help, etc. The interpreter also throws events if it finds a semantic error in a VoiceXML document. Each element in which an event can occur has an event handler. Each event handler is formed from any catch elements (and syntactic shorthands for catch elements) enclosed by that element. An event handler also inherits catch elements ("as if by copy") from each of its higher level elements, as needed. In this way, common event handling behavior can be specified at any level, and it applies to all lower level event handlers.

4. ELEMENTS

The following sections cover the language in topical order. This table is both an overview of the elements and an index to where they are discussed.

Element	Purpose	Page
<assign>	Assign a variable a value.	23
<audio>	Play an audio clip within a prompt.	34
<block>	A container of (non-interactive) executable code.	40
<break>	JSML element to insert a pause in output.	34
<catch>	Catch an event.	30
<clear>	Clear one or more form variables.	45
<choice>	Define a menu item.	22
<disconnect>	Exit a session.	47
	SABLE element to classify a region of text as a particular	34
<dtmf>	Specify a touch-tone key grammar.	27
<enumerate>	Shorthand for enumerating the choices in a menu.	22
<exit>	Exit a document or application.	47
<else>	Used in <if> elements.	45
<elseif>	Used in <if> elements.	45
<emp>	JSML element to change the emphasis of speech output.	34
<form>	A dialog for presenting information and collecting data.	18
<field>	Declares an input field in a form.	37
<filled>	An action executed when fields are filled.	18
<goto>	Go to another dialog in the same or different document.	46
<grammar>	Specify a speech recognition grammar.	27
<help>	Catch a help event.	30
<if>	Simple conditional logic.	45
<initial>	Declares initial logic upon entry into a form.	40
<link>	Specify a transition common to all dialogs in the link's	24
<menu>	A dialog for choosing amongst alternative destinations.	22
<meta>	Define a piece of meta data as a name/value pair.	47
<nomatch>	Catch a "nomatch" event.	30
<noinput>	Catch a "noinput" event.	30
<object>	Interact with a custom extension (e.g., speech object).	40
<prompt>	Queue TTS and audio output to the user.	28
<pros>	JSML element to change the prosody of speech output.	34
<record>	Record an audio sample.	42
<reprompt>	Ask to play a field prompt when a field is re-visited after	46
<sayas>	JSML element to modify how a word or phrase is spoken.	34
<throw>	Throw an event.	30
<transfer>	Transfer the caller to another destination.	42
<value>	Insert the value of a variable in a prompt.	25
<var>	Declare a variable.	25
<vxml>	Top-level element in each VoiceXML document.	12

5. DOCUMENT STRUCTURE

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,*
- *declarations of document-level variables,*
- *document-level catch elements, and*
- *links.*

Execution within one document. Document execution begins at the first dialog by default. As each dialog executes, it determines the next dialog to transition to. 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>
  <meta name="author" content="John Doe"/>
  <meta name="maintainer" content="hello-support@hi.com"/>
  <var name="hi" expr="'Hello World!'"/>
  <form>
    <block>
      <value name="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>
  <meta name="author" content="John Doe"/>
  <meta name="maintainer" content="hello-support@hi.com"/>
  <var name="hi" expr="'Hello World!'"/>
  <form>
    <block><value name="hi"/>Goodbye!</block>
  </form>
</vxml>
```

Attributes of `<vxml>` include:

`application`	The URI of this document's application root document, if any.
`caching`	The default caching policy for this document, either "safe" to ensure that all fetches get the most recent copy, or "fast" to use cached copies if they have not expired. "Fast" is the default.

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 (unless it was told to load the root document itself). The application root document remains loaded until the interpreter is told to load a document that belongs to a different application. So 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:

```
<!–Application root document (doc1.vxml) -->
<?xml version="1.0"?>
<vxml>
  <var name="bye" expr="'Ciao'"/>
  <link next="/operator_xfer.vxml"> Operator </link>
</vxml>

<!-- Second document (doc2.vxml) -->
<?xml version="1.0"?>
<vxml application="doc1.vxml">
  <form id="say_goodbye">
    <field name="answer" type="boolean">
      <prompt>Shall we say <value name="application.bye"/>?</prompt>
      <filled>
        <if cond="answer == true">
          <exit/>
        </if>
        <clear name="answer"/>
      </filled>
    </field>
  </form>
</vxml>
```

In this example, the application is designed so that doc2.vxml must be loaded first. Its "`application`" attribute specifies that doc1.vxml should be imported as the application's root document. So, doc1.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.

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 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.

6. FORMS

Forms mainly consist of *form items*: input fields, control items, and special items. Forms may also contain variable declarations, `<filled>` actions, and event handlers. Forms execute in an implicit loop, where each iteration picks a form item to visit, uses the picked item's information to prompt the user, and fills form variables based on the user's response. As variables are filled, the `<filled>` actions are triggered and executed. As form items have their variables filled, they are no longer visited.

The author of the form typically inserts a `<goto>` to submit the data to the server once the form is filled, or to take the user to a different dialog or document. If a form is filled and no transfer of control takes place, then it is as if the author had placed an `<exit/>` at the end (a reasonable thing to do for forms that only convey information).

The major type of form item is the *field item*, which gathers a value from the user and stores it in a variable defined in the local scope of the form. 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 the resulting events. A field item may also have a `<filled>` element that defines an action to take just after the field item's variable is filled in. Field items are subdivided into:

field	These are plain fields, the ones you will most use. These are used to gather dates, monetary amounts, yes/no answers, phone numbers and so on.
record	A record field item records an audio clip in its variable. It differs from plain fields in that all grammars are turned off while recording. A record field could be used to gather a voice message, for example.
transcribe	A transcribe field also gathers an audio clip, but instead of storing the audio in its variable, it transcribes the audio clip into text and stores the text. This might be used to collect an email or pager message.

Issue: This is experimental. This feature may not be widely implemented yet. Is it nevertheless a useful feature?

Other types of form items are:

initial	Mixed initiative forms may contain one initial item, which controls the form interaction until at least one plain field has been successfully filled in, or until it decides the user must be prompted for each field item individually.
block	A block is a sequence of executable statements used for prompting and computation, but not for gathering input. Normally, blocks are visited and executed once (they act as though they are fields that get filled in automatically the first time).
object	This is a block-like item that executes some platform-specific function. It may control properties of the user's device, gather a string variable using a proprietary DTMF text entry method, or use a platform-specific voice entry module to gather a set of fields (e.g., a date or flight information).
transfer	Another block-like item that transfers the user to another connection.

Each form item has a guard variable that is undefined when the form is entered. As long as the guard variable is `undefined`, that form item is eligible to be visited. The guard variable for a field item (field, record, or transcribe) is simply the item's associated variable, which becomes defined when set by the user utterance.

Non-field items normally have internal guard variables which are defined when the non-field item completes successfully. The initial form item's guard variable gets defined automatically when at least one field item is filled. The block, object, and transfer items always complete successfully, unless an event is thrown that they do not handle.

All forms execute by repeatedly finding and executing the first form item (in document order) whose guard variable is still undefined. Quite often the default behavior is sufficient, but you may encounter a need for more detailed control over the process. This can be done by explicitly assigning values to guard variables, and using `<clear>` to make them undefined again. For instance, you could bypass a field collection by assigning a value to it, or force it to be recollected by clearing its value.

A non-field form item can also be controlled in this manner. Instead of letting its guard variable default to an internal one, specify an explicit guard variable and then set and clear it as needed.

6.1 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="weather.gram#state"/>
    <catch event="help">
      Please speak the state for which you want the weather.
    </catch>
  </field>
  <field name="city">
    <prompt>What city?</prompt>
    <grammar src="weather.gram#city"/>
    <catch event="help">
      Please speak the city for which you want the weather.
    </catch>
  </field>
  <block>
    <goto next="http://www.clouds.example/" submit="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 weather information form's first iteration selects the first block, since its hidden guard variable is initially undefined. This block outputs a main prompt, and its hidden guard variable is set automatically when the block completes. On the form's second iteration, the first block is skipped because its guard variable is defined, and the "state" field is selected because the variable "state" is still undefined. The field prompts the user for the state, and then sets the variable once the user has provided a proper value. The third form iteration prompts and collects the "city" field. The fourth and last 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="type">
    <prompt count="1">What kind of credit card do you have?</prompt>
    <prompt count="2">Type of card?</prompt>
    <!-- The in line grammar syntax is to be determined. -->
    <grammar>
        visa               {visa}
      | master [card]      {mastercard}
      | amex               {amex}
      | american [express] {amex}
    </grammar>
    <help> Please say visa, mastercard, or amex. </help>
  </field>

  <!-- The grammar for type="digits" is built in. -->
  <field name="num" type="digits">
    <prompt count="1">What is your card number?</prompt>
    <prompt count="2">Card number?</prompt>
    <dtmf maxdigits="16"/>
    <catch event="help">
      <if cond="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="type == 'amex' && num.length != 15">
         American Express card numbers must have 15 digits.
         <clear name="num"/>
         <throw event="nomatch"/>
      <elseif cond="type != 'amex' && num.length != 16"/>
         Mastercard and Visa card numbers have 16 digits.
         <clear name="num"/>
         <throw event="nomatch"/>
      </if>
    </filled>
  </field>

  <field name="date" type="digits">
    <prompt count="1">What is your card's expiration date?</prompt>
    <prompt count="2">Expiration date?</prompt>
    <dtmf maxdigits="4"/>
```



```
      <help>
        Say or key in the expiration date, for example 12 01.
      </help>
      <filled>
        <!-- validate the mmyy -->
        <var name="mm"/>
        <var name="i" expr="date.length"/>
        <if cond="i == 3">
            <assign name="mm" expr="date.substring(0,1)"/>
        <elseif cond="i == 4"/>
            <assign name="mm" expr="date.substring(0,2)"/>
        </if>
        <if cond="mm == '' || mm < 1 || mm > 12">
            <clear name="date"/>
            <throw event="nomatch"/>
        </if>
      </help>
    </field>

    <field name="acknowledge" type="boolean">
      <prompt>I have <value name="card_type"/> number <value name="card_num"/>,
          expiring on <value name="expiry_date"/>.  Is this correct? </prompt>
      <filled>
          <if cond="acknowledge == true">
              <goto next="place_order.asp" submit="type num date"/>
          </if>
          <clear name="type num date"/>
      </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 Amex.

H: Uh, American Express. *(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 6

C: What is your card's expiration date?

H: 1 2 0 1

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 variable in the form's local 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.

6.2 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 be given an `<initial>` form item 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.

Form attributes are:

`id`	The name of the form
`cost`	The cost in abstract "credits" of executing the form (default is "`0`"). The interpretation of costs is determined by the application and/or platform.
`scope`	The default scope of the form's grammars. If it is "`local`" – the default – form grammars are active only in the form. If the scope is "`document`", the grammars will be active no matter where the user is in the document, or if the form is in the application root document, its grammars will then be active in any loaded document of the application.

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 an (unnecessary) confirmation of the city and state:

```
<form id="weather_info">
  <grammar src="weather.gram#cityandstate"/>

  <!-- Caller can't barge in on today's advertisement. -->
  <block>
    <prompt bargein="false">
      Welcome to the weather information service,
      <audio src="http://www.adnet.com/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"> <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 tell us the city 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 name="city">, <value name="state">?
    </prompt>
    <filled>
      <if cond="go_ahead == true">
        <prompt bargein="false">
          <audio src="http://www.adnet.com/wis2.wav/"/>
        </prompt>
        <goto next="http://www.clouds.example/" submit="city state"/>
      </if>
      <clear name="city state"/>
    </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: What state?

H: Uh, California.

C: What city?

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 …

Setting the modal attribute to `true` on the "`go_ahead`" field enables only the `boolean` grammar, requiring the confirmation prompt to be replied to before continuing.

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 …

Explicit control over field collection. In addition to altering a form's flow of control by setting and clearing guard variables, you can explicitly specify the next field item to go to with `<goto nextitem>`. After each utterance by the user, all the `<filled>` actions triggered by the new values are executed in document order. If several of them specify a `<goto nextitem>`, the first one wins, and in fact the `<goto>` takes effect immediately. If you force a visit to an item with `<goto nextitem>`, its guard variable is cleared automatically.

Here is an example using the system "`event.cancel`" event:

```
<form id="survey_1999_07_30">
   <catch event="cancel">
      <goto nextfield="confirm_cancel"/>
   </catch>
   <block>
      Hello, you have been called at random to answer questions
      critical to U.S. foreign policy.
   </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 metrics tons per annum?</prompt>
   </field>
   <block>
      <goto next="register" submit="q1 q2 q3"/>
   </block>
   <field name="confirm_cancel" type="boolean">
      <prompt>You have elected to cancel.  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_cancel">
                Okay, but the U.S. State Department is displeased.
                <exit/>
            <else/>
                Good, let's pick up where we left off.
            </if>
        </filled>
    </field>
</form>
```

If the user says "Cancel" in response to any of the survey questions, an "`event.cancel`" event is thrown by the platform and caught by the `<catch>` element which causes "`confirm_cancel`" to be the next visited field. The "`confirm_cancel`" field would not be visited during normal completion of the survey because of the `<block>` element.

6.3 Form Interpretation Algorithm

We've talked about form interpretation at a conceptual level. In this section we describe the model in some detail.

VoiceXML interpretation proceeds by alternating between two phases:

- *The* collect *phase: the next unfilled* form item *(such as a `<field>`) 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 timeout or telephone hangup).*
- *The* process *phase: an input collected by the collect phase is processed by filling form items and executing `<filled>` elements to perform actions such as input validation or default value computation. An event is processed by executing catch elements.*

Collect phase. The purpose of the collect phase is to collect an input or an event. During the collect phase a *form item* (`field`, `initial`, `block`, `object`, `transfer`, or `transcribe`) is selected for visiting as follows:

- *If a `<goto>` from the previous step specified a "`nextitem`" attribute then the specified form item is chosen to be visited.*
- *Otherwise the first form item whose* guard variable *does not yet have a value is chosen to be visited. (The guard variable may be specified by the element's "`name`" attribute or it may be anonymous)*
- *If no such element exists - that is, all form items' guard variables have a value - then interpretation terminates.*

The form item so chosen is then *visited*, which performs actions that depend on the element:

- *A `field`, `initial`, `object`, `record`, `transfer`,* or `transcribe` *element is visited by queuing any specified prompts, enabling appropriate grammars, and then waiting for an input or an event. When the input or event has been collected VoiceXML interpretation proceeds to the process phase to process it.*
- *A `block` element is visited by setting its guard variable to a defined value, evaluating its content, and then remaining in the collect phase, re-examining the form items' guard variables and selecting another form item to be visited as described above.*

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

- *If an input matches a grammar from a link then a `goto` or `throw` for the link is processed.*

- *If an input matches a grammar in a form other than the current form then the current form is abandoned and the other form becomes the current form, and processing continues in that form at the following step.*
- *If an input matches a grammar then:*
    1. *slot values returned by the grammar are assigned to the appropriate field variables,*
    2. *the set of applicable `<filled>` blocks is identified depending on which variables already have values and which have just received values from the input, and*
    3. *the content of each such identified `<filled>` element is executed in document order, each in the context where it is declared. If a `<goto>` is encountered, the remaining `<filled>` elements are not executed.*
- If an event (such as a `timeout` or a `hangup`) occurred then the applicable `catch` element is identified and its content is executed.

After completion of the process phase interpretation continues by returning to the collect 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 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.espn.com/vxml/start.vxml">
     ESPN sports </choice>
  <choice next="http://www.weatherchannel.com/intro.vxml">
     weather </choice>
  <choice next="http://www.caltech.edu/phys/astronews.vxml">
     Caltech astrophysics news </choice>
</menu>
```

This dialog might proceed as follows:

C: Welcome home. Say one of ESPN sports, weather, or Caltech astrophysics news.

H: Astrology.

C: I did not understand what you said.

C: Say one of ESPN sports, weather, or Caltech astrophysics news.

H: ESPN sports.

C: *(proceeds to http://www.espn.com/vxml/start.vxml)*

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

`id`	The name of the menu.
`cost`	The cost in "units" associated with the menu (default is "`0`").
`scope`	The menu's grammar scope. If it is "`local`" – 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.*
- *These grammar fragments are used to form the `<enumerate>` prompt string.*
- *The choice element specifies a URI to go to.*

Choice attributes are:

`dtmf`	The DTMF sequence for this choice.
`next`	The URI of next dialog or document.
`fetchtimeout`	The interval to wait before throwing a "`badnext`" event.
`caching`	Either "`safe`" to ensure that the fetch get the most recent copy, or "`fast`" to use the cached copy if it has not expired. The `<vxml>` element's "`caching`" attribute is used by default.

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

```
<menu>
  <prompt>
    For sports press 1, For weather press 2, For astrophysics press 3
  </prompt>
  <choice dtmf="1" next="http://www.espn.com/vxml/start.vxml"/>
  <choice dtmf="2" next="http://www.weatherchannel.com/intro.vxml"/>
  <choice dtmf="3" next="http://www.caltech.edu/phys/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">
  <prompt>
    For sports press 1, For weather press 2, For astrophysics press 3
  </prompt>
  <choice next="http://www.espn.com/vxml/start.vxml"/>
  <choice next="http://www.weatherchannel.com/intro.vxml"/>
  <choice next="http://www.caltech.edu/phys/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 those choices appear in the menu. If it is used with no content, as in the above example, a default template that lists all the choices is used, based on interpreter context and/or user's locale. 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 name="_prompt"/>, press <value name="_dtmf"/>
    </enumerate>
  </prompt>
  <choice next="http://www.espn.com/vxml/start.vxml">
```



```
    ESPN sports </choice>
  <choice next="http://www.weatherchannel.com/intro.vxml">
    weather </choice>
  <choice next="http://www.caltech.edu/phys/astronews.vxml">
    Caltech astrophysics news </choice>
</menu>
```

then the menu's prompt would be:

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

Issue: <value name="_prompt"/> and <value name="_dtmf"/> It would be useful to come up with shorter, more readable alternatives.

Say what you hear. Any multi-word phrase in the speech grammar fragments actually specifies a set of words and phrases to listen for. This supports "say what you hear" matching. For example, the phrase "Caltech astrophysics news" is transformed to also allow the user to say "Caltech", "astrophysics", "Caltech news", "astrophysics news", and so on.

Issue: We need to specify this transformation. For instance, "stop words" like "a" and "the" should not be used in generating the alternatives. Need to handle disambiguation (e.g. "news" could match two items)

Interpretation model. The menu is conceptually transformed into 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 (because they may be active outside the menu).

When the user says something matching the grammars, the single field's dummy variable is set to indicate the choice selected. The field's implicit `<filled>` block contains conditional logic determining where to transition to.

8. LINKS

A link specifies one or more grammars and a destination to go to when one of those grammars is matched. For instance, this link takes you to a book-selling application when you say "books" or press "2".

```
<link next="http://www.vxmlbooks.com/intro.asp">
  <dtmf> 2 </dtmf>
  <grammar> books | Voice XML books </grammar>
</link>
```

The link's grammars are active throughout its document. If the link is defined in an application root document, its grammars are active in any loaded document belonging to that application. Links occur inside documents, not inside forms or menus.

You can also define a link that, when followed, throws an event instead of going to a new document. This event is thrown in the context of the form item the user input was gathered in, not the context of the document was defined in. For example, this link causes "event.help" to be thrown in the current form item when the user exhibits frustration:

```
<link event="help">
  <grammar> arrgh | alas all is lost | fie ye froward machine </grammar>
</link>
```

Attributes include:

`next`	The document to go to when the user says or keys something matching this link.

fetchtimeout	The interval to wait while fetching this document before throwing "error.badnext".
event	Alternatively, the event to throw when this link is matched.
caching	Either "safe" to ensure that the fetch get the most recent copy, or "fast" to use the cached copy if it has not expired. The <vxml> element's "caching" attribute is used by default.

9. VARIABLES AND EXPRESSIONS

VoiceXML supports a simple notion of variables and expressions. The goal is to support conditional transitioning and validation checks, not to support arbitrary computation.

The expression language is ECMAScript, for familiarity. (This may also enable implementers to reuse ECMAScript interpreters.)

9.1 Variables

Variables are loosely typed, and their values are cast to strings, numbers, or Boolean as needed. If a cast fails, an "error.semantic" event is thrown.

Variable names are an initial letter or underscore, followed by zero or more letters, digits, and underscores. Variables beginning with underscore are reserved for internal use.

Issue: See W3C Namespaces in XML: http://www.w3.org/TR/1999/REC-xml-names-19990114

Declarations

Variables are declared by <var> elements:

```
<var name="home_phone" expr="9786122500"/>
<var name="pi"         expr="3.14159"/>
<var name="city"       expr="'Sacramento'"/>
```

They are also declared by <field> elements:

```
<field name="home_phone" type="phone">
  <prompt>Enter your home phone number.</prompt>
</field>
```

Variables declared without an explicit initial value are undefined, and any reference to them causes "error.semantic" to be thrown.

Scopes

Variables can be defined in various scopes:

session	These variables are global to the entire user session, from call initiation to termination. They are read-only and specific to each individual interpreter context. New session variables cannot be declared.
document	These are declared with <var> elements that are children of the document's <vxml> element. They exist while the document is loaded, and are visible only within that document.
local	These are visible to the current dialog only. They are reinitialized each time the dialog is entered.

The document variables of the application root document are also visible to any other loaded document in that application, as "application" variables.

Conceptually, each form item has its own scope to hold event counters and other control information, but these are not accessible to the VoiceXML author.

Issue: Since session variables are vendor-specific, portable applications will need a way to test a variable to see if it is defined.

Issue: Should we encourage the use of Vcard naming standards for profile information in the session scope for interpreter contexts that are profile-aware? It would be unfortunate if one session had the current user's address in a variable called "HADDR" and another had it in HOME_ADDR".

Referencing Variables

Variable values are set by <assign>:

```
<assign name="i" expr="i+1"/>
```

Variables are referenced by "name" attributes:

```
<field name="flight_num" type="digits"> …
<prompt>Your charge is <value name="charge"/> …
```

They also appear in expression attributes:

```
<assign name="var1" expr="var1 + 1"/>
<if cond="var1 > 12"> …
```

Variable references match the closest enclosing scope. You can prefix a reference with a scope name for clarity or to resolve ambiguity.

For instance, if the session, the application root document, another loaded document in that application, and a form within that other loaded document each define a variable called "x", then the form can reference these as:

```
session.x
application.x
document.x
local.x
```

Note that if you have a variable called "x" in the application root document, it is referred to as "document.x" in the root document, and "application.x" in any non-root documents.

9.2 Expressions

The expression language is a subset of ECMAScript (see http://www.ecma.ch/stand/ECMA-262.htm).

Note that the operators ">", "<", ">=", "<=", and "&&" must be escaped in XML (to ">" and so on). Our examples do not show these escapes; we assume that authors are using XML editors that allow the unescaped forms to be used.

Issue: Should we define VoiceXML-specific entities ∧ (for &&) ≤ (for <=) ≥ (for >=) to augment the XML entities < > ??

There are string, number, and boolean literals, plus the null literal:

```
'This is a string'
100
29.95
true
false
null
undefined
```


Two attributes are used to specify expressions: "cond" for conditional expressions resulting in true or false, and "expr" for expressions that compute a value.

```
<assign name="flavor" expr="'vanilla'"/>
<if cond="airline == 'united'"> …
```

There are arithmetic operators:

```
total_cost + 10
unit_price * quantity
cost = cost / 100
attempts = attempts - 1
```

comparison operators:

```
amount < 250
amount <= 1000
title == 'director'
amount != 0
balance >= charge_amount
x > y
```

logical operators:

```
title == 'manager' && amount < 250
!(amount > 250.00 || tip/amount > .20)
```

and the special ternary operator:

```
(a > b)? a:b
```

String operations include length, index, substring, and concatenation:

```
<if cond="ssn.length == 9"> …
<if cond="part_no.indexOf('#') != -1"> …
<assign name="manufacturer" expr="part_code.substring(0, 3)"/>
<assign name="dtmf_input" expr="dtmf_input.concat(last_digit)"/>
```

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 VoiceXML <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 tag. For inline grammars the type parameter specifies a mime type that governs the interpretation of the content of the <grammar> tag.

Issue: No existing grammar format is XML-based. We will work with the W3C to create such a format.

An external JSGF grammar is specified by an element of the form

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

The mime type may be optional in this case because this information may be obtained via the URI protocol (as in the case of HTTP) or may be inferred from the filename "extension". However, if the type is specified using the type attribute it should override other information about the type.

See Appendix E for notes on using the Java Speech API Grammar Format (JSGF) with VoiceXML.

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>
```

A grammar format can be based on XML, or not. See Appendix F for one possible XML-based DTMF grammar format.

10.3 Scope of Grammars

Field grammars are always scoped to their fields, that is, they are not active unless the user is in that field.

Link grammars are by default given the document scope, and are active throughout their document. If they are defined in the application root document, links are also active in any other loaded application document.

Form grammars are by default given the local 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.

Menu grammars are also by default given local 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.

Sometimes a menu or form may need to have some grammars active throughout the document, and other grammars that should be active only when in the menu or 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 <menu> or <form> element itself:

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

10.4 Activation of Grammars

Speech and DTMF grammars can be statically analyzed to determine which ones are active at any particular wait for user input. When the interpreter waits for input as a result of visiting a field, the following grammars are active:

- *The grammars for that field;*
- *The grammars for its form;*
- *Any grammars for links in its document and its application root document;*
- *Any grammars for menus and other forms in its document which are given document scope; and*
- *Any grammars for menus and forms in its application root document which are given document scope.*

If the form item is a modal field (i.e., its "modal" attribute is set to "true"), all grammars except its own are turned off during the wait. If the field item is a record or transcribe, no grammar is active.

Issue: What happens if several grammars are active, and more than one matches a user utterance? The current view is that which one gets picked is arbitrary, and that authors should be careful never to overlap grammars. But grammars in inner scopes could be given priority over those in outer scopes.

Issue: Currently if a grammar is matched and takes you out of a form, all that form data is lost. Do we want an option where a form's data is sticky from one visit to the next?

11. EVENT HANDLING

Events are thrown by the platform when the user does not speak, doesn't speak intelligibly, asks for help, etc. The interpreter also throws events if it finds a semantic error in a VoiceXML document. They can also be thrown by the <throw> element. Events have a type, which is a string value.

Each element in which an event can occur has an *event handler*. Each event handler is formed from any catch elements (and syntactic shorthands for catch elements) enclosed by that element.

An event handler also inherits catch elements ("as if by copy") from each of its higher level elements, as needed. If a field, for example, has no catch for "nomatch" specified, and its form does, the form's "nomatch" catch is conceptually copied into the field's event handler. In this way, common event handling behavior can be specified at any level, and it applies to all lower level event handlers.

If an event handler *still* does not have catch elements for certain important events ("nomatch", "noinput", "help", and "cancel"), the interpreter copies in its own defaults for them (see below).

When an event is thrown, the element's event handler processes it. If that event handler has a catch element for the event, that element processes it. If the event handler does not have a

catch element for the event, the event is thrown to the interpreter, which reports it and returns control to the interpreter context.

11.1 Throw

"Throw" throws an event. An application can define its own event types as needed:

```
<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 field.

```
<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" count="3">
      <prompt>Security violation!</prompt>
      <goto next="apprehend_felon" submit="user_id"/>
    </catch>
  </field>
  <block>
    <goto next="#get_city"/>
  </block>
</form>
```

It can contain sequential logic, prompts, gotos, etc. It cannot contain fields.

Attributes of <catch> are:

event	The event to catch. Required.
count	The occurrence of the event (default 1). Using the count you can handle different occurrences of the same event differently. Event counters exist for each type of event handled by an event handler: these counters are reset each time the handler's enclosing element is re-entered.
cond	An optional condition to test to see if the event is caught by this handler. Defaults to "true".

11.3 Shorthand Notation

The <help>, <noinput>, and <nomatch> elements are shorthands for very common types of event handlers:

```
<help count="1">No help is available.</help>
<noinput>I didn't hear anything, please try again.</noinput>
<nomatch>I heard something, but it wasn't a known city.</nomatch>
```

These elements take the attributes:

count	The event count.
cond	An optional condition to test to see if the event is caught by this handler. Defaults to "true".

11.4 Default Catch Elements

The event handler for a field, dialog, or document is comprised of all its catch elements, including those elements that are syntactic short-hands for <catch>, plus copies of all higher level catch elements. The interpreter is also required to add in default catch elements for the "noinput", "help", "nomatch", and "cancel" events if the author did not specify them. This is done for conciseness.

For instance, with no default mechanism, the following form would terminate if the user said nothing, said something that was misunderstood, said "help", or said, "cancel":

```
<form id="wrap_up">
   <field name="send_it_to_my_pager" type="boolean">
      <prompt>Do you want it sent to your pager?</prompt>
   </field>
</form>
```

Rather than force the author to insert four catch elements for each field, the interpreter must insert them automatically. The effect on the above example is as if the author had written:

```
<form id="wrap_up">
   <field name="send_it_to_my_pager" type="boolean">
      <prompt>Do you want it sent to your pager?</prompt>
      <catch event="noinput">
         <reprompt/>
      </catch>
      <catch event="help">
         <prompt>Sorry, no help is available.</prompt>
         <reprompt/>
      </catch>
      <catch event="nomatch">
         <prompt>I did not understand what you said.</prompt>
         <reprompt/>
      </catch>
      <catch event="cancel"/>
   </field>
</form>
```

11.5 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:

event.cancel	The user has asked to "cancel".
event.disconnect.hangup	The user has hung up.
event.disconnect.transfer	The user has been transferred unconditionally to another line and will not return.
event.exit	A document has asked the interpreter to exit and return control to the interpreter context. (Higher-level documents in the call stack can intercept this event).
event.goodbye	The user has asked the system to hang up, but is still on the line.

Issue: is this an interpreter context-specific event?

event.help	The user has asked for help.
event.noinput	The user has not said anything within the timeout interval.

`event.nomatch` The user said something, but we did not understand it.

The predefined errors are:

`error.badnext` A "`next`" resulted in a failed fetch. These may be further subdivided into `error.badnext.http.404`, and so on for applications that require fine-grained error handling.

`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.unsupported.mime_type`
The requested resource has a MIME type that is not supported by the platform, e.g., an unsupported grammar format or audio file format.

`error.unsupported.`*`element`*
The platform does not support the given *element*. For instance some platforms may not implement `<transcribe>`. This allows the use of event handling to adapt to different platform capabilities.

`error.unsupported.object`
An `<object>` element refers to an object that is not supported by the platform.

Some application-specific error types could be:

`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.

References to the pre-defined event types may omit the "`error`" or "`event`" prefix. Catches can catch specific events ("`event.cancel`") or all those sharing a prefix ("`event.badnext.*`").

12. CACHING

VoiceXML interpreter contexts, 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 pages as images are to HTML pages.

Fetch performance is even more important for audio interactions than for visual. Users expect quicker feedback in audio interactions, and there are fewer techniques to give them partial feedback while the full information is fetched. In the visual world, the user can be occupied reading low-bandwidth text and looking at gradually forming images. In the audio world, essentially the full audio recording must be fetched before the user hears anything.

Accuracy of cached content is also more important for audio interactions. While the visual user sees more information and is somewhat better able to perceive when it is out of date, the audio user hears less information. The visual user may know she is interacting with a program that caches information and that has (personalized) controls determining when to refresh, but these points are far more subtle for the user in an audio interaction.

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 browsers 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 in the `<vxml>` element can be set to "`safe`" or "`fast`" to determine what default policy to use for that document. If this attribute is not specified, the "`fast`" policy is used.

For finer-grained control, each element that refers to a resource fetched from the web can also be given a "`caching`" attribute of "`safe`" or "`fast`". If it is not specified, the policy in effect for the document is used.

For example:

```
<!-- By default, this document will use caching="fast". -->
<vxml>
   …
   <form id="test">
      <block>
         <!-- Welcome rarely changes, so fast caching is fine. -->
         <audio src="http://www.weather4U.com/vxml/welcome.wav"/>
         <!-- Ads change all the time, so safe caching is needed. -->
         <audio caching="safe" src="http://www.ads4U.com/weather4U/ad17"/>
      </block>
      …
   </form>
   …
</vxml>
```

Also, the "`caching`" attribute of the application root document overrides the caching attribute on all other documents in that application.

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 outputs synthesized speech and audio clips to the user. 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.

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>
```

Issue: Is this reasonable?

13.2 Speech Markup

Prompts can have speech markups to indicate emphasis, breaks, and prosody:

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

VoiceXML will track W3C standards for speech markup as they emerge. Initially we will support the <break>, , <emp>, <pros>, and <sayas> markups. See the DTD in the appendices for details.

See http://www.javasoft.com/products/javamedia/speech/forDevelopers/JSML for the Java API Speech Markup language and http://www.bell-labs.com/project/tts/sable.html for the Sable Consortium's markup language. These are very close, and the W3C standard will likely be based on them.

While the interpreter must tolerate the full set of speech markups, if it's 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.

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.com/thrush.wav"/>
    We have 250 kilogram drums of thistle seed for
   <sayas class="money">299.95</sayas> plus shipping
   and handling this month.
   <audio src="http://www.birdsounds.com/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 name="greeting"/>
   To keep it press star 5.
```



```
    To re-record press star 7.
    To exit to the main menu press star 9.
</prompt>
```

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

```
<prompt>
  <audio src="welcome.wav">Welcome to TelePortal</audio>
</prompt>
```

If the platform supports local audio prompt lookups, a "builtin://" protocol might be used to access audio samples kept locally:

```
<prompt>
  <audio src="builtin://maya/hello">Welcome</audio>
</prompt>
```

The "mode" attribute can be used to specify that an audio clip be constructed by concatenating a given set of digits and characters:

```
<audio mode="builtin://walter/%c.wav"><value name="acct_num"/></audio>
```

Issue: This assumes a "builtin" protocol understood by the platform and that the platform understands that '%c' means character replacement. The extensibility of this scheme to more complex types, such as money or time, is unknown.

Attributes of <audio> include:

src	The URI of the audio prompt.
mode	Specifies that the enclosed text is to be spoken using concatenated audio instead of TTS, if possible. It also specifies the location of the audio fragments.
caching	Either "safe" to ensure that the fetch get the most recent copy, or "fast" to use the cached copy if it has not expired. The <vxml> element's "caching" attribute is used by default.
fetchtimeout	The interval to wait for the fetch to return before throwing a "badnext" event.

Issue: Is it better to have "builtin://prompt/foo" and "builtin://grammar/bar"? The URI ought to be interpretable without reference to its syntactic context.

Issue: Must be careful to keep the <audio> element consistent with W3C markup standards.

13.4 The <value> element

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

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

Sometimes a variable needs to be rendered using an appropriate format. A North American phone number 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 name="home_num" class="phone"/></prompt>
```

The valid formats are the same as those supported in the <sayas> speech markup.

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

Attributes of <value> are:

name	The name of the variable to render.
class	The <sayas> class of the variable, e.g. phone, date, money.
mode	The type of rendering: "tts" (the default), or "recorded".

recsrc	The URI of the audio files to be concatenated when mode="recorded".

Issue: A naming convention is needed to name the individual audio files needed for the supported types.

Issue: How can we best express the automated generation of voice file sequences from variable data? Two alternatives are shown here, one using <audio>, one using <value>.

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".

13.6 Counts

Tapered prompts are those that change with use. Information-requesting prompts may become terser 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 field 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 an odd form with a form level prompt and field level prompts:

```
<form id="tapered">
  <block>
    <prompt bargein="false">Welcome to the ice cream form.</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 form.

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 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.

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 a platform-specific (or perhaps even a user-specific) value, e.g., "`5s`". Timeouts may be specified on prompts or on fields. The reason for allowing them on prompts is to support tapered timeouts: the user may be given five seconds for the first input attempt, and ten seconds on the next, for example.

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 1924 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, if it is explicitly specified. Otherwise, the timeout of the input field itself is used, if it is explicit. Finally, if neither the last prompt nor the input field has an explicit timeout, the platform default timeout is used.

13.8 Summary

Prompts have the following attributes:

`count`	A number that allows you to emit different prompts if the user is doing something repeatedly. If omitted, it defaults to "`1`".
`bargein`	Is bargein allowed? Default is "`true`".
`timeout`	What timeout should be used for the following user input? The default "`noinput`" timeout is roughly five seconds, but the actual value is a platform or even user preference.

14. FIELD

A field specifies an input item to be gathered from the user. It specifies:

- *the declaration of a form-level variable to hold the field's value;*
- *lead-in prompt(s) to use for that field;*
- *speech and DTMF grammar(s) to be used in interpreting the user's response;*
- *event handlers for "`help`", "`nomatch`", "`filled`", and other events.*

Attributes of fields include:

`name`	The variable (in the form scope) that will hold the result.
`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).
expr	The initial value of the field's variable.
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.
timeout	The "noinput" timeout. After waiting for speech for this interval, the "noinput" event is thrown. Defaults to a platform and perhaps user dependent amount.

The "slot" attribute is used to map the slots returned by a grammar into the field variables, in cases where the grammar slot names differ from the field names. In most cases, you needn't bother specifying this attribute. If the grammar returns only one slot, as do the built-in type grammars like "phone", then no matter what the slot's name, the field variable gets the value of that slot. Or, if the grammar returns many slots, but there is one that matches this field's name, then you can also leave off the "slot" attribute.

14.1 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 read in prompts. An example:

```
<field name="lo_fat_meal" type="boolean" timeout="8000ms">
  <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>
</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" 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>
       <throw event="nomatch"/>
     </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 storage convention for the value returned. These are independent of locale and of the implementation.

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

Issue: The following descriptions assume U.S. English. We need to specify them in detail.

The builtin types are:

`boolean`	Inputs include “yes” or “no” phrases. DTMF 1 is yes and 2 is no. DTMF 6 is no and 9 is yes. The output is “true” or “false”.

Issue: maybe no good input standards for this one.

Issue: ECMAScript booleans when converted to strings are "true" or "false", but a bare string used in a conditional expression is true if it is not the empty string. So "false" is interpreted as "true"! We'll need to disallow bare strings in conditional expressions.

`date`	Inputs are a variety of phrases. The output is a date string with format yyyy/mm/dd, e.g., “2000/07/04”.
`digits`	The output is a string of digits from 0 to 9. Inputs are spoken or DTMF digits. You can say “two one two seven”, but not “twenty one hundred and twenty-seven”.
`money`	The output is a string amount with zero or more decimal digits, as is appropriate. Inputs are spoken or DTMF, with the “*” key acting as a decimal point.

Issue: we might have `money/usd`*, etc.*

`number`	The output is a string holding an integer. The inputs are spoken or DTMF digits. You can say “twenty one hundred and twenty-seven”.
`phone`	The output is a string holding a full phone number. Inputs are spoken or DTMF digits. There are locale dependencies here. In North America, the result would be, e.g., “6307482500”.
`time`	The output is a five character string in 24 hour format, e.g., “15:30”. Input can be via DTMF.

14.2 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="text/jsgf"/>
</field>
```

A “`builtin://`” protocol can be used to refer to built-in grammars not in the standard set described above:

```
<field name="product_code" slot="pcode">
  <prompt>What is the product code?</prompt>
  <grammar src="builtin://ibm_prodcode.jsgf"/>
</field>
```

Grammars can be specified inline, in (for example) JSGF:

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


15. BLOCK

This element is a form item. It specifies a block of executable elements that are to be executed if the block's guard variable is unset.

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

Blocks are normally executed just once per form invocation. They have an internal guard variable that is undefined each time the user enters the form, and that is automatically defined once the block is executed.

Sometimes you may need more control over blocks. To do this, you can provide the name of the guard variable. This variable is declared in the "local" scope of the form. You can control the block by setting the variable and clearing it.

```
<block name="set_pi">
   I'm settting pi now.
   <assign name="pi" expr="3.14"/>
</block>
```

Attributes include:

name	The name of the guard variable used to track whether this block has been executed. Defaults to an inaccessible internal variable.

16. INITIAL

In a 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 prompted for. Like field items, it has prompts, catches, and event counters. Unlike field items, an initial has no grammars, no field to assign to, and no <filled> actions. There is at most one initial form item per form. For instance:

```
<form id="get_from_and_to_cities">
   <grammar src="http://www.directions.com/grammars/from_to.jsgf"/>
   <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 cont="1">Please say something like "from Atlanta Georgia
         to Toledo Ohio".</prompt>
      <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" value="true"/>
      </nomatch>
   </initial>
   <field name="from_city">
      <grammar src="http://www.directions.com/grammars/city/jsgf"/>
      <prompt>Which city are you leaving from?</prompt>
      … etc. …
   </field>
   … etc. …
</form>
```

When control is in the initial item, the user is prompted with only the form grammars turned on, and no field grammars (though grammars from other forms, menus, and links may also be active). If an event occurred in response ("noinput", or "nomatch", say) then the initial's event handler processes it and the user is reprompted. The initial item is always selected to prompt

the user until its guard variable is set (as in the above example), or until at least one regular field variable is given a value successfully.

Note that once the initial item's guard variable is set, the form interpretation algorithm will automatically start visiting each individual form item in sequence.

Attributes include:

`name`	The name of the guard variable. Defaults to an inaccessible internal variable.

17. OBJECT

A VoiceXML implementation platform may have its own, platform-specific, functionality. For instance, one may give authors access to vendor-specific information-gathering objects (e.g., one that collects credit card information and completes a purchase).

This is done with the `<object>` element, which is essentially a function call that takes zero or more arguments and returns zero or more results. The "function name" is a URI. The current VoiceXML session suspends until the object returns.

For example, one platform may support an object that gathers credit card information from the user and then commits a purchase:

```
<object src="method://approve_purchase"
     submit="amount description"
     expect="card cardno expiry amount approved"/>
```

On another platform, users might enter text using an advanced keypad-based entry algorithm that uses dictionaries and built-in grammar rules:

```
<form id="gather_pager_message">
  <block>
     Enter your message by pressing your keypad once per letter.  For
     a space, enter star.  To end the message, press the pound sign.
     If a word's key sequence is ambiguous, you will be asked to choose
     the word you want after you press the pound sign.
  </block>
  <object src="text://enter" expect="message"/>
  <block>
    <assign name="document.pager_message" expr="message"/>
    <goto next="#confirm_pager_message"/>
  </block>
</form>
```

The user is prompted for the pager message, then keys it in. The final block puts the message in the variable "`document.message`". The next dialog presumably reads the saved message back to the user and then asks if it should be sent, re-entered, or discarded.

Attributes include:

`src`	The URI of the object to execute.
`submit`	A list of the variables to pass to the object.
`expect`	A list of variables to expect in return from the object. These are automatically declared in the form's "local" scope.

If an object has positional arguments, then the "`submit`" and "`expect`" variable names are plain VoiceXML variables. If an object has named arguments, use the ":" notation to map object variable names to VoiceXML variable names. Here is one such object that makes a flight reservation:

```
<object src="http://flight.com/reserve_flight"
```



```
        submit="rf_date:date rf_seat:seat_type rf_class:class"
        expect="rf_flight:flight_no rf_seat:seat rf_cost:cost"/>
```

The object variable names all start with “rf_”, while the VoiceXML variables don’t.

And if James Bond’s car were “telematically” controlled:

```
<object src="astonmartin://eject_passenger"
        submit="dl:document.delay alt:document.altitude"
        expect="rc:result_code"/>
```

If an <object> element refers to an unknown object, the “event.unsupported.object” event is thrown.

Issue: Should <object> be able to contain argument content, such as prompts and catches?

Issue: The "slot:variable" mapping syntax is not well-liked.

18. RECORD

This element accepts a recording from the user and stores it in a variable. The variable is declared by this element, in the form’s “local” scope. It throws “noinput” if the user has not responded in the correct interval. It contains only <prompt> and <catch> elements.

```
<record name="mailing_address" maxlength="10s"
       finalsilence="2s" dtmfterm="true">
  <prompt>Say your mailing address</prompt>
  <noinput>I didn't hear anything, please try again.</noinput>
</record>
```

The <record> element appears only inside a <form>, where it acts like an audio input field. During a recording, all grammars are turned off.

The attributes are:

beep	Is a tone emitted just prior to recording? Defaults to “true”.
name	The resulting audio variable.
maxlength	The maximum time to record.
finalsilence	The interval of silence to indicate end of speech.
dtmfterm	Does a DTMF key press terminate recording ? Defaults to “true”.
type	The MIME format, defaults to a platform-specific format.

19. TRANSCRIBE

This field item transcribes spoken input from the user and stores the transcribed text in a variable. The variable is declared by this element, in the "dialog" environment. It contains only <prompt>, <dtmf>, <grammar>, and <catch> elements. The <dtmf> and <grammar> elements specify DTMF or spoken input that terminates the transcription. This terminating input is not included in the transcription.

The attributes are:

name	The variable (in the local scope) that will hold the result.
expr	The initial value of the variable.
modal	If this is false, all active grammars are turned on while collecting this field. If this is true (the default), then only this field’s grammars are enabled: the others are temporarily disabled.

timeout	The "noinput" timeout. After waiting for speech for this interval, the "noinput" event is thrown. This timeout defaults to a platform and perhaps user-dependent amount.
maxlength	The maximum time to record.
finalsilence	The interval of silence to indicate end of speech.

Issue: Is this as harmonized with <record> as it could be? These could have identical interfaces, in theory.

20. TRANSFER

Transfer the caller to another phone number, and either wait for the call to return, or continue execution (to tidy up and terminate the session).

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

```
<form name="transfer">
   <block>
      <audio src="prompt://muzak/chopin12>
   </block>
   <transfer
      dest="phone://18005445555"
      timeout="30s"
      bridge="true"
      duration="document.call_duration">
      <catch event="busy" next="#main_menu>
         Sorry, our customer support team is busy serving
         other customers.  please try again later.
      </catch>
      <catch event="noanswer" next="#main_menu">
         Sorry, our customer support team's normal hours
         are 9 am to 7 pm monday through saturday.
      </catch>
   </transfer>
   <block>
      <goto next="#report_call_duration"/>
   </block>
</form>
```

You can provide a <dtmf> sub-element specifying the digits that terminate the transfer. (Conceivably, one could have <grammar> sub-elements, too, but this involves allocating a full ASR resource to the transferred conversation, and these are far more expensive than a DTMF recognition resource.)

Attributes include:

dest	The URI of the destination (phone, IP telephony address).
bridge	This attribute determines what to do once the call is connected. If "bridge" is "true", the VoiceXML session suspends until the transferred call finishes. If it is "false", the session continues after the call is transferred. If the session then tries to speak or listen to the user, the platform will throw "error.semantic".
timeout	The time to wait while trying to connect the call before raising the "noanswer" event. Default is "15s".

`maxlength`	The time that the call is allowed to last, or "`0`" if it can last arbitrarily long. Only valid if "`bridge`" is "true". Default is "`0`".
`duration`	If present, the variable in which the actual call length in milliseconds is stored. Only valid if "`bridge`" is "`true`".

Events thrown inside a `<transfer>` include:

`event.disconnect.hangup`	If the inbound caller hung up.
`event.disconnect.transfer`	If the user has been transferred unconditionally to another line and will not return.
`event.busy`	If the called party is busy.
`event.noanswer`	If the called party did not answer in the time specified.
`error.semantic`	If there is a semantic error , e.g., invalid destination.

21. FILLED

Forms contain various form items which are visited in order to collect user input. Some of these form items – `<field>`, `<record>`, and `<transcribe>` – are called field items because they specify an individual form field. The `<filled>` item differs from form items in that it specifies an action to perform when one or more field items are filled in.

The `<filled>` element may be put in two places. If it is a child of the `<form>` element itself, it specifies an action to perform after some combination of fields are filled. One could have a `<filled>` element that does a cross-check to ensure that a starting city field differs from the ending city field, for instance:

```
<form id="get_starting_and_ending_cities">
   <field name="start_city">
      <grammar src="http://www.grammars.com/voicexml/city.grm"/>
      <prompt>What is the starting city?</prompt>
   </field>
   <field name="end_city">
      <grammar src="http://www.grammars.com/voicexml/city.grm"/>
      <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 namelist="start_city end_city"/>
      </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 the user. 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.fedex.com/grammars/served_cities.grm"/>
      <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 "any" (the default), or "all". 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. Ignored when specified in <filled> inside a field item.
namelist	The fields to trigger on. These default to all of the form's field items. Ignored when specified in a <filled> inside a field item.

Most of the time you needn't specify attributes: it is nearly always possible to bundle all the form filled actions inside a single <filled> that is fired on each change. Only when you want to catch specific combinations of fillings do you need these attributes.

22. VAR

This element declares a variable, in cases where it is not input with a <field>. Declarations must go in the document (for document variables) or dialog (for local variables).

```
<var name="phone" expr="6305487401"/>
<var name="y" expr="document.z+1"/>
```

Attributes include:

name	The name of the variable that will hold the result.
expr	The initial value of the variable.

23. ASSIGN

An assignment of a value to a variable:

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

Attributes include:

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

24. CLEAR

The <clear> element concise way to unset the values of one or more variables. Instead of:

```
<assign name="city"  expr="undefined"/>
<assign name="state" expr="undefined"/>
<assign name="zip"   expr="undefined"/>
```

You can say:

```
<clear name="city"/>
<clear name="state"/>
```



```
<clear name="zip"/>
```

Or simply:

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

This works for variables defined in any scope. As a special case, <clear> with no attributes unsets all the guard variables in a form, including implicitly defined guard variables:

```
<clear/>
```

Attributes include:

name	The name of the variable(s) being cleared. Defaults to all the guard variables in the current form.

25. REPROMPT

This element appears inside <catch> elements or the shorthand equivalents. A <catch> by default transfers control so that the user is not reprompted. In cases where you would like the user to be reprompted, insert this element at the end of the catch.

26. IF

The <if> element is used for conditional logic. It has an optional <else> element, and also a form with <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>
```

Issue: Currently, <else/> and <elseif/> act as separators (empty content model), not containers. Should these be changed to be non-empty elements with the same content model as <if>?

27. GOTO

Transition to a field in the current form, a new dialog, or a different document:

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


When transitioning to a new VoiceXML document, its first dialog becomes the current dialog, unless the URI explicitly mentions another one.

Attributes include:

nextitem	The name of the next field item to visit in the current form.
next	The URI of next dialog or document.
submit	A list of the variables to submit in an HTTP request (only if getting a new document). The default is the set of all field item (field, record, transcribe) variables if the <goto> is in a form, or nothing if the <goto> is anywhere else. You can submit a whole scope at a time by saying "local.*", "document.*", etc.
method	The request's method if getting a new document – "get" (the default) or "post".
enctype	The MIME encoding type of the submitted document.
caching	Either "safe" to ensure that the fetch get the most recent copy, or "fast" to use the cached copy if it has not expired. The <vxml> element's "caching" attribute is used by default.
fetchtimeout	The interval to wait before throwing a "badnext" event.

When going to a new document, the "session" level variable environment is retained. If the current document and the next one both share the same application root document, the application root document's variable environment is also retained. But the current document's variables are lost (to keep them, copy them into the application root document's variable environment).

28. EXIT

Throws an "exit" event. Control returns to the interpreter context, which determines what to do next. This event cannot be caught.

```
<exit/>
```

Attributes include:

expr	A return expression (e.g., "0", or "oops!").

29. DISCONNECT

Causes the platform to disconnect from the user. As a by-product, the platform will throw a "event.disconnected.hangup" event.

```
<disconnect/>
```

30. META

The <meta> element specifies meta-data, as in HTML. For example:

```
<meta name="maintainer" content="support@vxml.org"/>
```

Issue: Probably want to recommend a standard set of meta-data.

31. TIME DESIGNATIONS

Time designations follow cascading style sheet conventions (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

32. MISCELLANEOUS ISSUES

Issue: User profile environment and Vcard.

Issue: n-best results from grammar?

Issue: Probabilistic prompts (to reduce boredom, increase comprehension).

Issue: Forward and backward controls – not a platform issue but a language issue? Notionally, there may be a "session" document with links for browser controls and bookmarks.

Issue: Capability announcements to server. W3C CC/PP.

Issue: Should confirmation be formalized with a separate <ack> element?

APPENDIX A. GLOSSARY OF TERMS

active content Code that permits arbitrary computation, such as ActiveX components, Java applets, etc.

active grammar A grammar, speech or DTMF, that remains active even when the user is in a different dialog.

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 An element that defines a portion of an *event handler* that responds to a given *event*.

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

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.

event handler An execution time construct composed of all the *catches* defined in an element plus certain default catches.

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

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>, <transcribe>, <object>, and <transfer>.

guard variable A variable, either implicitly or explicitly defined, associated with each *form item* in a *form*. If the guard 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.

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*.

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 to become 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*.

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*.

APPENDIX B. VOICEXML DOCUMENT TYPE DEFINITION

```
<!-- A DTD for Voice Extensible Markup Language -->
<!-- Copyright 1999 VoiceXML Forum (AT&T, IBM, Lucent Technologies, Motorola) -->

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

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

<!ENTITY % content.type "CDATA">

<!ENTITY % condition "CDATA" >

<!ENTITY % duration "CDATA" >

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

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

<!ENTITY % field.name "NMTOKEN" >

<!ENTITY % field.names "NMTOKENS" >

<!ENTITY % integer "CDATA" >

<!ENTITY % uri "CDATA" >

<!ENTITY % cache.attrs
       "caching       (safe|fast)    'fast'" >

<!ENTITY % next.attrs
       "next          %uri;          #REQUIRED
       fetchtimeout   %duration;     #IMPLIED " >

<!ENTITY % submit.attrs
       "%next.attrs;
       method         (get|post)     'get'
       enctype        %content.type; 'application/x-www-form-urlencoded'
       submit         NMTOKENS       #IMPLIED" >

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

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

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

<!ELEMENT vxml         (%event.handler; | form | link | menu | meta | var)+ >
<!ATTLIST vxml
       %cache.attrs;
       application    %uri;          #IMPLIED >

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

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

<!ENTITY % input       "dtmf | grammar" >

<!ENTITY % scope       "(document | local)" >

<!ENTITY % dialog.attrs
       "id            ID             #IMPLIED
       timeout        %duration;     #IMPLIED
       bargein        %boolean;      'true'
       cost           %integer;      #IMPLIED " >

<!ELEMENT form
       (grammar | %event.handler; | filled | initial | object |
       record | transcribe | transfer | %variable;)* >
<!ATTLIST form
       %dialog.attrs;
       scope          %scope;        #IMPLIED >

<!ELEMENT menu         (prompt | choice)* >
```



```
<!ATTLIST menu
        %dialog.attrs;
        scope           %scope;         #IMPLIED
        dtmf            %boolean;       'false' >

<!ELEMENT choice        (#PCDATA)* >
<!ATTLIST choice
        %cache.attrs;
        %next.attrs; >

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

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

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

<!ELEMENT reprompt      EMPTY >

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

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

<!ELEMENT field                 (%audio; | %event.handler; | filled | %input; | prompt)* >
<!ATTLIST field
        name            %field.name;    #REQUIRED
        type            %field.type;    "string"
        expr            CDATA           #IMPLIED
        slot            NMTOKEN         #IMPLIED
        modal           %boolean;       'false'
        timeout         %duration;      #IMPLIED >

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

<!ELEMENT initial       (%audio; | %event.handler; | prompt)* >
<!ATTLIST initial
        name            %field.name;    #IMPLIED >

<!ELEMENT block                 (%executable.content;)* >
<!ATTLIST block
        name            %field.name;    #IMPLIED >

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

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

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

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

<!ENTITY % event.name  "NMTOKEN" >

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

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


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



```
        type            NMTOKENS        #REQUIRED >

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

<!ELEMENT link          (dtmf | grammar)* >
<!ATTLIST link
        %submit.attrs;
        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
        fetchtimeout    %duration;      #IMPLIED
        %cache.attrs; >

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

<!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" >

<!ELEMENT dtmf          (#PCDATA)* >
<!ATTLIST dtmf
        src             %uri;           #IMPLIED
        scope           %scope;         #IMPLIED
        inter           %duration;      #IMPLIED
        mindigits       %integer;       #IMPLIED
        maxdigits       %integer;       #IMPLIED
        termkey         %key;           #IMPLIED >

<!ELEMENT grammar       (#PCDATA)* >
<!ATTLIST grammar
        src             %uri;           #IMPLIED
        type            CDATA           #IMPLIED
        scope           %scope;         #IMPLIED >

<!ELEMENT record        (%audio; | catch | error | noinput | prompt)* >
<!ATTLIST record
        name            NMTOKEN         #REQUIRED
        beep            %boolean;       'false'
```



```
        type            CDATA           #IMPLIED
        maxlength       %duration;      #IMPLIED
        finalsilence    %duration;      #IMPLIED
        dtmfterm        %boolean;       'true'  >

<!ELEMENT transcribe    (%audio; | catch | error | noinput | prompt)* >
<!ATTLIST transcribe
        name            NMTOKEN         #REQUIRED
        expr            CDATA           #IMPLIED
        modal           %boolean;       'false'
        timeout         %duration;      #IMPLIED
        maxlength       %duration;      #IMPLIED
        finalsilence    %duration;      #IMPLIED >


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

<!ELEMENT disconnect    EMPTY >

<!ELEMENT transfer      (catch | dtmf | grammar)* >
<!ATTLIST transfer
        dest            %uri;           #REQUIRED
        bridge          %boolean;       'false'
        timeout         %duration;      #IMPLIED
        maxlength       %duration;      #IMPLIED
        duration        %field.name;    #IMPLIED >

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

<!ENTITY % if.attrs
        "cond           CDATA           #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            CDATA           #IMPLIED >

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

<!ELEMENT goto          EMPTY >
<!ATTLIST goto
        %submit.attrs;
        %cache.attrs;
        nextitem        %field.name;    #IMPLIED >

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

<!ELEMENT object        EMPTY >
<!ATTLIST object
        src             %uri;           #REQUIRED
        submit          CDATA           #IMPLIED
        expect          CDATA           #IMPLIED >
```


APPENDIX C. FORM INTERPRETATION ALGORITHM

The algorithm below handles:

- *Entering the form with an utterance spoken to one of its grammars while the user was in a different form.*
- *Leaving the form because the user spoke to grammar belonging to another form, menu, or link.*
- *Processing multiple field fills from one utterance.*
- *Prompt tapering.*
- *Fields, blocks, objects, and filleds.*
- *Grammar activation and deactivation.*

The interpretation of menus is done by the same algorithm. A menu is just a form with a single field. The field's grammar is the set of choices for the menu. The field has a <filled> action that does a <goto> based on the chosen field value.

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

active grammar set
: A set of grammars that is active during a listen operation. This set is statically determinable.

utterance
: A summary of what the user said or keyed in, including the grammar matched, and a dictionary of grammar slot name/value pairs.

execute
: By this, we mean to execute some executable content – either a block, a filled action, or a set of filled actions. If an event is thrown, 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:

```
// Initialize the form.
foreach ( form item in the form )
   Clear its guard variable.
foreach ( field item in the form )
   Set its prompt counter to 1.
if ( there is an initial item )
   Set its prompt counter to 1.
if ( user entered form by speaking to its grammar in a different form )
{
   Fill any fields mentioned in the user utterance.
   Execute the corresponding <filled> actions in document order.
}

// Do forever: choose a form item to visit and execute it.
while ( true )
{
   // Collect phase: choose a form item to visit.
   if ( the last form item visit ended with a <goto nextitem> )
      Choose that next field item.
   else if ( there is a form item whose guard variable is undefined )
      Choose the first such form item in document order.
   else
      Do an <exit/> -- the form specified no successor dialog or document.

   // Execute that form item.
   if ( a field was chosen )
```



```
{
   // Visit the field.
   Play the appropriate prompt.
   Increment the field's prompt counter.
   if ( the field is modal )
      Set the active grammar set to the field grammars only.
   else
      Set the active grammar set to the field's grammars, the
        form's grammars, and any grammars from other forms, menus,
        and links that are active.
   Listen to the user and obtain the utterance.

   // Process the field.
   if ( utterance matched a grammar from outside the form )
      Go to that form or menu; or execute the link's goto or throw.
   Fill any fields mentioned in the user utterance.
   Execute the corresponding <filled> actions in document order.
}

if ( a record was chosen )
{
   // Visit the record.
   Play the appropriate prompt.
   Increment the record's prompt counter.
   Set the active grammar set to the empty set.
   Record an audio clip.

   // Process the record.
   Fill this record field with the audio clip.
   Execute the corresponding <filled> actions in document order.
}

if ( a transcribe was chosen )
{
   // Visit the transcribe.
   Play the appropriate prompt.
   Increment the transcribe's prompt counter.
   Set the active grammar set to the transcription grammar only.
   Listen to the user and obtain the utterance.

   // Process the transcribe.
   Fill this transcripton field to the transcription.
   Execute the corresponding <filled> actions in document order.
}

if ( the initial was chosen )
{
   // Visit the initial.
   Play the appropriate prompt.
   Increment the initial's prompt counter.
   Set the active grammar set to the form's grammars, and any
      grammars from other forms, menus, and links that are active.
   Listen to the user and obtain the utterance.

   // Process the initial.
   if ( utterance matched a grammar from outside the form )
      Go to that form or menu; or execute the link's goto or throw.
   Fill any fields mentioned in the user utterance.
   Execute the corresponding <filled> actions in document order.
   Set the initial's guard variable to a defined value.
}

if ( a block was chosen )
{
   // Visit the block (no interaction with the user).
```



```
         // Process the block.
         Set the block's guard variable to a defined value.
         Execute the block.
      }

      if ( an object was chosen )
      {
         // Visit the object.
         Set the active grammar set to the empty set.
         Execute that object using the given arguments.

         // Process the object.
         Copy the expect variables returned into the form's local scope.
         Set the object's guard variable to a defined value.
      }

      if ( a transfer was chosen )
      {
         // Visit the transfer.
         Effect the transfer, possibly blocking until the transfer returns.
           (If the transfer is permanent, any subsequent attempt to
           communicate with the user throws error.semantic.)

         // Process the transfer.
         Set the transfer's guard variable to a defined value.
      }
   }
```


APPENDIX D. USING EXTENSIBLE STYLE SHEETS

The VoiceXML forum recognizes there may be a need to control resources at a more basic level than those offered by the high-level constructs presented in this document. In the world of visual presentation, HTML has been extended with a precise formatting model as defined in the Cascading Style Sheet specifications. This model defines a "flow object" calculus, specifying how boxes are arranged on a two-dimensional surface. With this model, the meaning of most HTML elements can be explained by style sheet parameters that fix the actual sizes and shapes of boxes, fonts, etc.

Similarly, the VoiceXML notation could be defined in terms of a “core” language, which is a simple, reactive programming model. The language, which would be a part of VoiceXML itself, would simplify the design of new abstractions. For example, in the core language, it would be possible to describe sub-dialogues by syntactic nesting. This syntactic flexibility allows the design of new abstractions to be expressed in the XSLT style sheet notation.

As an example of an abstraction, we may consider introducing a greater variety of error conditions than provided by the high-level language. These error conditions require the insertion of event handlers, and maybe sub-dialogues, in various critical places – something that can be expressed conveniently by XSLT style sheets.

The VoiceXML forum is aware that XSLT style sheets may be too complicated to be used by most programmers, although XSLT may constitute an important tool for defining customized dialog templates and other abstractions.

Unfortunately, the more accessible CSS notation is lacking in one fundamental regard: it cannot be used to insert default content (except strings).

For the reasons mentioned above, the forum has not presently decided on a style sheet language recommendation for VoiceXML.

APPENDIX E. 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 (for more details see JSGF specification).

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
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> =	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.

Issue: if no tag is specified and there is more than one word what is value - words separated by spaces? Does this work for all languages? What if multiple tags are specified?

Issue: The tag mechanism by itself only provides a single-value mechanism, and doesn't allow values returned by referenced rules to be modified and combined. A mechanism built on the JSGF tag mechanism for attribute-value pairs and for value composition will be defined in cooperation with the W3C.

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="text/jsgf">
        [please] help [me] [please] | [please] I (need|want) help [please]
```



```
    </grammar>
</link>
<field name="sandwich">
    <grammar type="text/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.

Issue: For this case in particular an XML-ized version of JSGF would be useful. This will be worked out with the W3C.

Issue: How is this case distinguished from the inline grammar fragment? If an XML-ized version is available, Is this option really 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.*

Issue: How can a grammar use the builtin grammars? They need to have JSGF names; how do these relate to the builtin: names?

APPENDIX F. POSSIBLE DTMF GRAMMAR

This appendix contains a possible DTMF grammar using XML.

Elements

```
<dtmf interdigittimeout="t"/>
```

This element contains the XML tree defining a grammar for DTMF detection. The "interdigittimeout" attribute indicates default interdigit timeout to be used between detected DTMF digits. It defaults to an implementation-defined value.

Grammars are composed of a sequence grammar elements or PCDATA, and recognized grammars can be annotated with returned values, and names using the <token> element.

The grammar is composed of the following elements:

```
<or/>
```

This element represents a disjunction of this left and right sides.

```
<choice count="n" min="m" max="k" interdigittimeout="t"/>
```

Repeat the contained element at least "m" times, to a maximum if "k" times, or repeat exactly "n" times. The default interdigit timeout is "t". The count attribute is mutually exclusive with min or max. If unspecified, min is 0. If unspecified, max is unlimited (except if there is an implementation defined maximum). If unspecified, count does not apply.

```
<any/>
```

This element defines one of any of the 12 DTMF characters (0-9, *, #)

```
<digits/>
```

This element defines one of any of the 10 DTMF numbers (0-9)

```
<token name="n" expr="e"/>
```

Where this element follows a grammar, it will return from a recognition the value denoted by the expression "e", if provided. If the name attribute is provided, then the value of the recognition will be assigned to the named field variable. This would be suitable for using the <dtmf> element as a form level grammar. If the "name" attribute is absent, then the recognized value can be assigned only to the field variable of the <field> element in which the grammar is declared.

Examples

Here are some examples:

1. To recognize the sequence consisting of two star keys pressed quickly after each other, we specify

```
<dtmf>
   **
</dtmf>
```

This would have to be declared in a <field> element.

2. To recognize either 1, corresponding to the value "blue_widgets" or 2, corresponding to the value "green_widgets", write

```
<dtmf interdigittimeout="500ms">
   1 <token expr="'blue_widgets'"> <or/>
   2 <token expr="'green_widgets'">
</dtmf>
```

This would have to be declared in a <field> element.

The interdigittimeout of 500ms is only relevant if this <dtmf> declaration is active simultaneously with another declaration that declares a DTMF sequence with a prefix that is either 1 or 2. For example, if the <dtmf> element

```
<dtmf interdigittimeout="2s">
   11
</dtmf>
```

is also active and 1 is pressed, then the inter-digit timeout is 500ms, the minimum of 500ms and 2s.

3. To recognize either 1 or *, corresponding to the value “blue_widgets” or 2, corresponding to the value “green_widgets”, write

```
<dtmf>
       <choice> 1 <or/> * </choice> <token expr="'blue_widgets'">
   <or/>
       2 <token expr="'green_widgets'">
</dtmf>
```

4. To recognize a sequence of digits with at least two digits and starting with 1, write

```
<dtmf>
   1
   <choice min="1">
      <digit/>
   </choice>
</dtmf>
```

5. To recognize any sequence of DTMF, write

```
<dtmf>
   <choice min="0">
      <any/>
   </choice>
</dtmf>
```

6. To recognize two arbitrary DTMF keys, write

```
<dtmf
   <choice count="2">
      <any/>
   </choice>
</dtmf>
```

7. To recognize a toll-free 10 digit North American Dialing Plan phone number, with optional leading 0 or 1, and optional trailing “#”.

```
<dtmf>
   <choice min=0 max=1>
      0 <or/> 1
   </choice>
   800 <or/> 888 <or/> 877
   <choice count=7>
      <digits/>
   </choice>
   <choice min=0 max=1>
      #
   </choice>
</dtmf>
```

8. To recognize a set of grammars in a form-level dtmf grammar which would define a set of voice mail box commands: *3 for delete, *6 to restart, *9 to end, **4 for help, *8 for transfer, and 0 for operator.

```
<dtmf>
*3  <token name="delete"> <or/>
*6  <token name="restart"> <or/>
*9  <token name="end"> <or/>
**4 <token name="help"> <or/>
*8  <token name="transfer"> <or/>
```



```
    0   <token name="operator">
    </dtmf>
```

DTD

```
<!ELEMENT dtmf         %dtmf.choices;>
<!ATTLIST dtmf
        name           NMTOKEN        #IMPLIED
        %dtmf.interdigittimeout;>

<!ELEMENT choice %dtmf.choices;>
<!ATTLIST choice
        %dtmf.count;
        %dtmf.interdigittimeout;>

<!ENTITY dtmf.interdigittimeout
        timeout        %duration;     #IMPLIED>

<!ENTITY dtmf.count
        count          %integer;      #IMPLIED
        min            %integer;      #IMPLIED
        max            %integer;      #IMPLIED>

<!ENTITY dtmf.string   #PCDATA>

<!ENTITY dtmf.choices (%dtmf.string; | or | choice | digits | any | token)*>

<!ELEMENT or EMPTY>

<!ELEMENT digits EMPTY>

<!ELEMENT any EMPTY>

<!ELEMENT token EMPTY>
<!ATTLIST token
        name           NMTOKEN        #IMPLIED
        expr           NMTOKEN        #REQUIRED>
```

