Media Objects Markup Language (MOML)
draft-melanchuk-sipping-moml-00

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

Abstract

The Media Objects Markup Language (MOML) is used to define media processing objects which execute on media servers. It defines a set of primitive media objects (called primitives) and provides tools to group primitives together and specify how they interact with each other. Clients use MOML to create precisely tailored media processing objects which may be used as parts of application interactions with users or conferences or to transform media flowing internal to a media server. IVR is an example of an application interaction with a user.

Table of Contents

1. Introduction......................................................3
2. Overview........................................................3
   2.1 Primitives..................................................4
2.2 Groups....................................................5
2.3 Events....................................................8
3. Structural Elements..........................................9
  3.1 <moml>....................................................9
  3.2 <group>....................................................9
  3.3 <groupexit>.............................................10
  3.4 <send>....................................................10
  3.5 <exit>....................................................11
4. Elements for Primitive Objects..........................11
  4.1 <play>....................................................11
    4.1.1 Child Elements...................................13
      4.1.1.1 <audio>.......................................13
      4.1.1.2 <tts>...........................................13
      4.1.1.3 <var>..........................................13
      4.1.1.4 <dtmfgen>....................................14
      4.1.1.5 <playexit>....................................15
  4.2 <record>.................................................15
    4.2.1 Child Elements..................................16
      4.2.1.1 <recordexit>..................................16
  4.3 <dtmf>....................................................17
    4.3.1 Child Elements..................................18
      4.3.1.1 <pattern>....................................18
      4.3.1.2 <detect>.......................................19
      4.3.1.3 <noinput>.....................................19
      4.3.1.4 <nomatch>....................................19
      4.3.1.5 <dtmfexit>....................................19
  4.4 <speech>..................................................20
    4.4.1 Child Elements..................................21
      4.4.1.1 <grammar>....................................21
      4.4.1.2 <match>.......................................22
      4.4.1.3 <noinput>.....................................22
      4.4.1.4 <nomatch>....................................22
      4.4.1.5 <speechexit>..................................22
  4.5 <vad>....................................................23
    4.5.1 Child Elements..................................23
      4.5.1.1 <voice>, <silence>, <tvoice>, <tsilence>.....23
  4.6 <gain>....................................................24
  4.7 <agc>....................................................24
  4.8 <clamp>..................................................24
  4.9 <relay>..................................................25
5. Examples.....................................................25
  5.1 Announcement...........................................25
  5.2 Voice Mail Retrieval....................................25
  5.3 Play/Record..............................................26
  5.4 Speech Recognition.....................................27
  5.5 Play/Collect.............................................28
  5.6 User Gain Control Widget................................29
6. XML Schema...................................................30
Security Considerations......................................39
1. Introduction

This document describes a markup language to configure and define media resource objects within a media server. The language allows the definition of sophisticated and complex media processing objects which may be used for application interactions with users, i.e. as part of a user dialog, or as media transformation operations. Media Objects Markup Language (MOML) itself does not specify a language suitable for constructing complete user interfaces as does VoiceXML. Rather, it defines a language from which individual pieces of a dialog may be specified.

MOML is not a standalone language but will generally be used in conjunction with other languages such as the Media Sessions Markup Language (MSML) [8]. MSML is used to invoke and control many different services on a media server and to manipulate the flow of media streams within a media server. Future work will define how MOML may be used directly with SIP using mechanisms such as "Basic Network Media Services with SIP" [9], App-info [10], and SIP events [11].

MOML has both a framework, which describes the composition of media resource objects, and the definition of an initial set of primitive media resource objects. The following sections describe the structure and usage of MOML followed by sections defining all of the MOML XML elements.

This work has been influenced by concepts from VoiceXML [7], "MRCP: Media Resource Control Protocol", [2] and "Media Policy Manipulation in the Conference Policy Control Protocol" [3].

Simple media resources and their composition into more complex operations is a central concept of this specification. This concept is used to precisely define the required behaviors. It is not meant to imply that media servers must be implemented from the same building blocks used to describe the behavior.

2. Overview

MOML is an XML [4] language for composing complex media objects from a vocabulary of simple media resource objects called primitives. It is primarily a descriptive or declarative language to describe media
processing objects. MOML does not directly define how objects get instantiated and used. Instead, it defines a minimal coordination mechanism between itself and its invocation environment through which that environment may cause objects to be instantiated and through which events can be sent or received.

MOML may be used to simply expose primitive media resource objects but will be used more often to describe dialog operations and media transformation objects which can be controlled via user interaction.

MOML does not contain any computation or flow control constructs. There are no results automatically generated when media operations complete. Users of MOML which require results must explicitly specify those results with a <send> or <exit> elements as part of the definition of the MOML object.

2.1 Primitives

Primitives perform a single function on a media stream such as generating audio, recognizing speech or DTMF, or adjusting the gain. They may be composed so that primitives execute concurrently. Primitives not composed for concurrent execution simply execute sequentially in the order they occur in a MOML document. All concurrently executing primitives in the same MOML object (defined in one MOML document) can interact with each other through events.

Currently all primitives use audio media but primitives for text and video will be defined in a future version of this specification. Primitives can roughly be considered to fall into one of three descriptive categories:

- recognizers have a media input but no output. They allow different things within a media stream to be recognized or detected and for events to be generated based upon received media.

- transformers have one media input and output and may send and receive events;

- sources and sinks generate or consume media. They have either a media input or a media output but not both. They may receive and generate events.

Primitives may define different media processing behavior (states) based upon the events which they receive. Primitives which support different processing states must define their default starting state and should support the "initial" attribute to allow that state to be specified when the primitive is instantiated. All primitives must support the "stop" event class.
The following types of primitives are defined within this specification:

<table>
<thead>
<tr>
<th>Recognizers</th>
<th>Transformers</th>
<th>Source/Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>dtmf</td>
<td>agc</td>
<td>play</td>
</tr>
<tr>
<td>speech</td>
<td>clamp</td>
<td>record</td>
</tr>
<tr>
<td>vad</td>
<td>gain</td>
<td></td>
</tr>
<tr>
<td>relay</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Primitives have shadow variables, similar to those within VoiceXML, which are automatically assigned values when the primitives are used. Upon initialization of a MOML context, all shadow variables have the string value "undefined". Each primitive has its own instance of shadow variables which are global in scope to the entire MOML context.

Names may be assigned to individual primitives when more than one primitive of the same type is used within one MOML document. Shadow variables are overwritten if the primitive has not been named and is instantiated a second time.

Shadow variables cannot be modified under user control. They may be returned from the MOML context using the <send> element.

2.2 Groups

Primitives are composed for concurrent execution by placing them within a <group> element. Groups define how media flows between multiple concurrently executing primitives. They have one or more inputs and one or more outputs. A <group> represents the declaration of a complex media processing operation. The event interaction between primitives (see the following sub-section) is defined within the context of one or more groups. However groups themselves do not scope events, they simply define that primitives are concurrently executing and a primitive must be executing in order to receive an event.

Groups may be used to describe dialog commands, such as a play/collect or play/record. They may also be used to describe media objects which transform a media stream while optionally allowing application or user control of the transformation. For example a gain control could be defined which responds to user speech or DTMF input. In this case a recognition primitive would send events to a gain control primitive.

Groups have one attribute which defines the media flow within them. They also have a dimension which defines how many media inputs and outputs they have. Currently dimensions of 1 and 2 are supported.
based upon the group topology. These correspond to a group with one
input and one output and a group with two inputs and two outputs.

Media flow to and from the primitives within the group is based upon
a topology attribute of the <group> element. This attribute defines a
topology schema and implies the group dimension. There are several
common ways in which primitives are often connected together. A
schema provides a convenient template which can be applied to
multiple primitives without having to define all of the individual
media relationships. The following two schemas are initially defined
for 1-dimensional groups:

- **star**: specifies that media sent to the group is sent to every
  primitive which has an input. The group bridges the output from
every primitive which has an output into a single common group
  output;

- **pipe**: specifies that the first primitive listed in the group
  receives the media sent to the group. Its output is to be
  connected to the input of the next primitive defined within the
  group and so on until the last primitive within the group which
  becomes the group output.

Groups with these topologies are shown in the two diagrams below. The
group on the left has a star topology and that on the right has a
pipe topology.

```
   /-> P1 --\   \             
   /          \            /   
G(in) +----> P2 ----> G(out) G(in) --> P1 --> P2 --> P3 --> G(out)  
\          /\           \    /     
 \-> P3 --/ 
```

More complex media flows may be created by nesting groups of pipe and
star topologies within each other. For example, the diagram below has
a group with a pipe topology nested within a star.

```
  /-----> P1 ------------------------\ 
  /                                       
Gs(in) +-> Gp(in) --> P2 --> P3 --> Gp(out) --> Gs(out)  
```

This combination could be used to create record operation where DTMF
was to be clamped from the recording itself, but a DTMF key press is
still used to stop the recording. In this case, P1 would be a DTMF
recognizer, P2 would be a clamp primitive, and P3 a recorder as shown
by the following example. This example omits child elements and
attributes not concerned with the core concept. The following section
discusses sending events and the details of each of the primitives is
defined in section 4.
A single schema, "fullduplex" is defined for a two dimensional group. A full-duplex two dimensional group has exactly two immediate children. Those children may be primitives or other one dimensional groups. A "fullduplex" group must only be used as the top most group and must not be nested. Each primitive (P1) and group (G2) becomes half of the full-duplex group as shown in the diagram below.

\[
\text{G-A(in1)} \rightarrow G2 \rightarrow \text{G-B(out1)}
\]

\[
\text{G-A(out2)} \leftarrow \text{P1} \leftarrow \text{G-B(in2)}
\]

Full duplex groups are symmetrical when both halves are the same. They are asymmetrical when they differ. Asymmetric groups need to have a name associated with each side. The left hand side is defined as the input of the first child of the full-duplex group combined with the output of the second child. The right hand side is reverse. These sides were named A and B respectively in the preceding diagram.

An example of a full-duplex group is the user operated gain control mentioned at the beginning of this sub-section. The gain should operate on the audio which a user hears, but the gain is controlled by recognizing things such as DTMF or spoken commands in media which the user originates. The following shows the XML tag grouping which would accomplish this and corresponds to the media flow shown in the diagram above. If the user’s audio is not required for anything other than control of the gain, then the <relay> is not required and the internal group could be omitted. A complete XML description for this is included in the examples section.

\[
\text{<group topology="fullduplex" lhs="A" rhs= "B"} \rangle
\]

\[
\text{<group topology="star">}
\text{<dtmf/>}
\text{<relay/>}
\text{<gain/>}
\text{</group>}
\]

It is expected that additional topology schemas together with methods to allow media flow to be explicitly defined will be developed in a future version of this specification.
Primitives within a group begin concurrently but may finish asynchronously based upon events which they receive or their task completes. A group terminates when all of the primitives within it have completed. If the group contains a `<groupexit>` element, then the contents of that element are executed as part of group termination.

A group itself may receive a stop event requesting termination. A stop event sent to the group causes a stop event to be sent to each of its currently active primitives. The `<groupexit>` element is not executed until all primitives have processed their respective stop events.

### 2.3 Events

Events provide the mechanism for primitives to interact with each other and for a MOML context to interact with its external environment. The external environment is defined by the way in which a MOML context has been invoked. This will generally be through MSML but other languages and protocols may also be used.

Every primitive and group conceptually implements their own event queue. Events sent to them get placed into their associated queue. Events are removed from their queues and processed in order. Primitives within a group conceptually have their own thread of execution. Due to the asynchronous nature of servicing events from multiple queues, it cannot be assumed that several events sent in sequence to different queues, will be processed in the order in which they were sent. For example, if recognition of something led to sending events to both a `<play>` and a `<record>` in that order, it is possible that the `<record>` may process its event before the `<play>`

Primitives each define the set of events which they support and the behavior associated with their handling of each event. This allows many types of behaviors to be defined. For example, VCR type controls can be constructed by defining primitives which support events corresponding to each control. Media recognition/detection can be used to cause those events to be generated.

Alternatively, events can be originated elsewhere, such as from an application server, and simply received by the primitive implementing the control. Examples of the use of events include adjusting volume (gain) and pause and resume of both announcement playout and record creation.

Primitives act on events based upon the longest match of an event name. Event names are a period '.' delimited sequence of tokens. The first token, or the root of the name, can be considered an event class. Matching allows a standard meaning to be defined and then extended based upon what triggers an event’s generation. For example,
a record primitive has different behavior depending upon whether it completed because a user stopped speaking or because it was cancelled. The recording is retained in the first case but not the second.

Longest match allows new recognizers to be created and used without changing how existing primitives are defined. For example, a face recognition capability could be created which generates a stop.frowning event when a user looks puzzled. Although no primitive directly defines this event, it will still effect a generic stop action. Primitives which require specialized behavior based upon frowning may be extended to support this. As well, the event can still be exported from the MOML context without requiring that primitives receiving the event understand facial expressions.

### 3. Structural Elements

Framework elements provide the structure for MOML.

#### 3.1 <moml>

The root element for MOML. The contents of this element describe a complete execution context for a media resource object.

Attributes:

- version: "1.0" Mandatory.
- id: an identifier unique to this object. Events returned from MOML (the "target" attribute of a <send> is equal to "source") will be correlated with this identifier. Mandatory.

Events:

- terminate: terminates the MOML context. A terminate event gets sent to the currently executing <group> or primitive.

#### 3.2 <group>

The <group> element allows the contained primitives to be executed concurrently.

Attributes:

- topology: specifies a schema which defines the flow of media within the group. Three schemas are initially defined. "fullduplex" is specified for use with two dimensional groups. "star" and "pipe" are for use with one dimensional groups. The
definition of these topologies is defined in section 2. Mandatory.

id: identifies name of the group. Mandatory when groups are nested.

lhs: the name of the left hand side of a full-duplex group. It consists of the input of the first child of the group combined with the output of the second child. Mandatory for a full-duplex group, forbidden otherwise.

rhs: the name of the right hand side of a full-duplex group. It consists of the output of the first child of the group combined with the input of the second child. Mandatory for a full-duplex group, forbidden otherwise.

Events:

terminate: causes a terminate event to be sent to each element contained within the group.

3.3 <groupexit>

The <groupexit> element allows events to be sent when group processing completes. Group processing completes when all contained primitives terminate.

Attributes:

none

Events:

none

3.4 <send>

Sends an event and optional namelist to the recipient identified by the target attribute. Event names are defined by the recipient. In the case where the recipient is a MOML group or primitive, the events are defined within this document. Other recipients may use names that are suitable for their environment.

Attributes:

event: the name of an event.

target: a type of primitive element or "group" or "source". When <send> is used within a group containing multiple
instances of the same type of primitive, then the specific primitive must be identified by appending its name to the type separated by a period ".". The token "group" identifies the enclosing group and the token "source" identifies the context which invoked the MOML object.

**namelist**: a list of zero or more shadow variables which are included with the event.

### 3.5 `<exit>`

Exit causes execution of the MOML object to terminate.

**Attributes:**

- **namelist**: a list of one or more shadow variables which may optionally be sent to the context which invoked the MOML object.

### 4. Elements for Primitive Objects

The following information is described for each primitive:

- the function which it performs
- the attributes which may be used to tailor its behavior
- the events which it is capable of understanding
- the shadow variables which provide access to information determined as a result of the primitive’s operation.

Subsections of a primitive define child elements of that primitive and are not themselves considered primitives. They do not receive events or populate shadow variables.

#### 4.1 `<play>`

Play is used to generate an audio stream. It plays in sequence the media created by the child media elements `<audio>`, `<tts>`, `<var>`, and `<dtmfgen>`. When the play stops, either because the terminate event is received or all media generation has completed, the `<playexit>` element, if present, is executed. At least one media generation element must be present.

Play supports two states; generate and suspend. Media generation occurs in the generate state and is suspended in the suspend state. Once in the suspend state, media generation continues upon receiving the generate event. The default initial state is generate.
Attributes:

- **id**: specifies an identifier for the audio stream sequence. The identifier, if specified, may be used to target event. Optional.

- **interval**: specifies the delay between stopping one iteration and beginning another. The attribute has no effect if iterations is not also specified. Default is no interval.

- **iterations**: specifies the number of times the media specified by the child media elements should be played. Defaults to once.

- **initial**: defines the initial state for the play element. Default is "generate".

- **maxtime**: defines the maximum allowed time for the <play> to complete.

- **offset**: defines an offset, measured in units of time, where the <play> is to begin media generation. Offset is only valid when all child media elements are <audio>.

- **skip**: an amount, expressed in time, which will be used to skip through the media when "forward" and "backward" events are received. Default is 3s (three seconds).

Events:

- **pause**: causes the play to enter the suspend state.

- **resume**: causes play to enter the generate state.

- **forward**: skips forward through the media. Only has effect when all child media elements are <audio>.

- **backward**: skips backward through the media. Only has effect when all child media elements are <audio>.

- **restart**: skips to the beginning of the media. Only has effect when all child media elements are <audio>.

- **toggle-state**: causes the suspend / generate state to toggle.

- **terminate**: terminates the play and assigns values to the shadow variables.

Shadow Variables:
play.amt: identifies the length of time for which media was generated before the play was stopped. This does not include time which may have elapsed while the play was in the suspend state.

play.end: contains the event which caused the play to stop. When the play stops because all media generation has completed, end is assigned the value "play.complete".

4.1.1 Child Elements

4.1.1.1 <audio>

Identifies a single file containing recorded audio. The URI attribute identifies the location of the audio file, which may be located internally within the Media Server or externally on an HTTP server.

Attributes:

uri: Identifies the location of the audio file. The file and http schemes are supported.

iterations: specifies the number of times the audio file is to be played. Defaults to once.

4.1.1.2 <tts>

Contents of the <tts> element are rendered using Text To Speech services and must be compliant to the SSML specification. Element content may be plain text, contain the SSML <speak> element, or the uri attribute should identify the location of text to be rendered.

Attributes:

uri: Identifies the location of the text to be rendered. The file and http schemes are supported.

iterations: specifies the number of times the text to speech block is to be rendered. Defaults to once.

4.1.1.3 <var>

Specifies the generation of audio from a variable using prerecorded audio segments. A variable represents a semantic concept (such as date or number) and dynamically produces the appropriate speech.

Prerecorded audio allows an application vendor or service provider to choose the exact voice for their audio and therefore completely control the "sound and feel" of the service provided to end users. It
provides very high audio quality and allows the variables to blend seamlessly into the surrounding audio segments.

Text to speech (TTS) using SSML may also be used to render variables, but may not provide as good quality, or allow as complete control of the "sound and feel" or user experience. TTS is normally used for reading text such as emails and for very large vocabularies such as stock names. TTS results in a very clear difference between the variables and the surrounding audio segments.

Attributes:

- **type**: specifies the type of variable. Mandatory. Variable type must be one of "date", "digits", "duration", "month", "money", "number", "silence", "time", or "weekday".

- **subtype**: specifies an optional clarification of type. Specific values depend upon the type.

- **value**: text which should be rendered appropriate to the type and subtype attributes.

4.1.1.4 <dtmfgem>

DTMF generator originates one or more DTMF digits in sequence.

Attributes:

- **digits**: A string of characters from the alphabet "0-9a-d##*" which correspond to a sequence of DTMF tones. Mandatory.

- **level**: used to define the power level for which the tones will be generated. Expressed in dBm0 in a range of 0 to -96 dBm0. Larger negative values express lower power levels. Note that values lower than -55 dBm0 will be rejected by most receivers (TR-TSY-000181, ITU-T Q.24A). Default is -6 dBm0.

- **dur**: the duration in milliseconds for which each tone should be generated. Implementations may round the value if they only support discrete durations. Default 100 ms.

- **interval**: the duration in milliseconds of a silence interval following each generated tone. Implementations may round the value if they only support discrete durations. Default 100 ms.
4.1.1.5 <playexit>

The <playexit> element is invoked when generation of all content of the <play> has come to completion. The contents of this element may be used to send events.

Attributes:

none

4.2 <record>

Record creates a recording. Similar to play, <record> supports two states; create and suspend. Received media becomes part of the recording when <record> is in the create state and is discarded when it is in the suspend state.

Recording terminates when a stop event is received or when a nospeech event is received and no audio has yet been recorded. <record> differentiates different types of stop events.

Attributes:

id: an optional identifier which may be referenced elsewhere for sending events to the record primitive.

append: a boolean which defines whether the recording is allowed to be appended to an existing file if dest already exists. Default is "false". The attribute is ignored if the scheme is http.

dest: the destination for the recording. Recording may be either local or external based upon the attribute value. Currently the file and http schemes are supported.

format: defines the encoding and file type of the recording.

initial: defines the initial state for the record element. Default is "create".

maxtime: defines the maximum length of the recording in units of time.

Events:

pause: causes the record to enter the suspend state. Received media is discarded.
resume: causes record to resume if it was suspended. It has no effect otherwise.

toggle-state: causes the suspend / create state to toggle.

stop: terminates the recording and assigns values to the shadow variables.

stop.cancelled: terminates the recording and assigns values to the shadow variables. If the dest attribute used the file scheme, the local recording is deleted. Applications are responsible for removing external files created using the http scheme.

stop.finalsilence: terminates the recording and assigns values to the shadow variables. If the dest attribute used the file scheme, the final silence is removed from the recording.

nospeech: terminates the recording and assigns values to the shadow variables if it is received and no recording has yet been created. The "nospeech" event is ignored if audio has already been recorded.

Shadow Variables:

record.len: the actual length of the recording measured in units of time. This does not include time which may have elapsed while the record was in the suspend state.

record.end: contains the event which caused the record to stop. When the record stops because maxtime is exceeded, end is assigned the value "record.timeexceeded".

4.2.1 Child Elements

4.2.1.1 <recordexit>

The <recordexit> element is invoked when the record operation completes or when the recording is terminated as a result of receiving the stop event. The <recordexit> element may be used to send events when the recording has completed.

Attributes:

none
4.3 <dtmf>

DTMF input fulfills several roles within MOML. It is used to trigger events which will affect the media processing operation of other primitives. It is also used to collect DTMF digits from a media stream which are to be reported back to the user of MOML. Often DTMF detection is used for both purposes. Barge is the most common example, where a prompt is stopped based upon DTMF input but more digits may remain to be collected.

DTMF detection supports multiple simultaneous recognition patterns. Different patterns can be used to trigger sending different events in order to implement DTMF controls. Alternatively one pattern may be used to represent a collection and another pattern, a substring of the first, used as a barge indication.

Note that all patterns share the same digit collection buffer, inter-digit timing, a single <nomatch> element, and a single <noinput> element. As such, multiple patterns may not be suitable to support simultaneous collections for different purposes. When this is required, separate <dtmf> elements should be used instead.

<dtmf> terminates if any of the <pattern>, <noinput>, or <nomatch> elements are matched the maximum number of times that they are allowed. The number of times they may match may be specified as an attribute of <dtmf> or of the individual child elements.

Attributes:

cleardb: a boolean indication of whether the buffer for digit collection should be cleared of any collected digits when the element is instantiated. If set to false, any digits currently in the buffer are immediately compared against the pattern elements.

fdt: defines the first-digit timer value. The first-digit timer is started when DTMF detection is initially invoked. If no DTMF digits are detected during this initial interval, the <noinput> element is invoked.

idt: defines the inter-digit timer to be used when digits are being collected. When specified, the timers is started when the first digit is detected and restarted on each subsequent digit. Timer expiration is applied to all patterns. After that, if any patterns remain active and a nomatch element is specified, the nomatch is executed and DTMF input terminates. The idt attribute should only be used when digit collection is being performed. No default.
starttimer: boolean value which defines whether the first digit timer (fdt) is started initially. When set to false, the starttimer event must be received for it to start. Default is false.

max: specifies the maximum number of times the <pattern>, <noinput>, and <nomatch> elements may be executed unless that element specifies differently. The value "0" may be used to indicate that there is no maximum. Default is once '1'.

Events:

starttimer: starts the first digit timer (fdt) if it has not already been started. Has no effect otherwise.

terminate: terminates the DTMF input and assigns values to the shadow variables.

Shadow Variables:

   dtmf.digits: the string of DTMF digits which have been received (the contents of the digit buffer).

   dtmf.len: the number of digits in the digit buffer.

   dtmf.last: the last digit in the digit buffer.

   dtmf.end: contains the event which caused the <dtmf> to terminate or is assigned one of "dtmf.match", "dtmf.noinput", or "dtmf.nomatch" depending upon which of the corresponding elements reached its maximum.

4.3.1 Child Elements

4.3.1.1 <pattern>

The pattern element describes one or more DTMF digits that are to be recognized. When the pattern is matched, the child elements are executed.

Attributes:

digits: The digit pattern which should be matched.

format: an enumerated value which defines the format used to express the digit pattern. The format may be "mgcp" or "megaco" for patterns expressed as digit map from those specifications, or as one of the simple built-in formats defined within this
specification. Specific formats are TBD, but will include a
generic "digits" which will be the default.

max: specifies the maximum number of times the <pattern> may be
matched. The value zero ‘0’ may be used to indicate that there
is no maximum. This value overrides any specified in <dtmf>.
Default is once.

4.3.1.2 <detect>

The contents of the <detect> element are executed whenever any DTMF
is first detected. It may be matched at most once.

Attributes:

none

4.3.1.3 <noinput>

The <noinput> element is used when DTMF is being collected. Children
of the <noinput> element are executed when DTMF has not been detected
and the first digit timeout occurs.

Attributes:

max: specifies the maximum number of times the <noinput> may be
matched. The value zero ‘0’ may be used to indicate that there
is no maximum. This value overrides any specified in <dtmf>.
Default is once.

4.3.1.4 <nomatch>

The <nomatch> element is used when DTMF is being collected. Children
of the <nomatch> element are executed when it is determined that none
of the individual patterns can be matched.

Attributes:

max: specifies the maximum number of times the <nomatch> may be
matched. The value zero ‘0’ may be used to indicate that there
is no maximum. This value overrides any specified in <dtmf>.
Default is once.

4.3.1.5 <dtmfexit>

The <dtmfexit> element is invoked when the dtmf input completes
because one of <pattern>, <noinput>, or <nomatch> occurred its
maximum number of times.
Attributes:

none

4.4 <speech>

Activates grammars or user input rules associated with speech recognition. If multiple grammars are specified, all are activated. All active grammars share the same timers, recognition attributes, and <noinput> and <nomatch> elements. Each grammar may have its own <match> element.

<speech> terminates if any of the <grammar>, <noinput>, or <nomatch> elements are matched the maximum number of times that they are allowed. The number of times they may match may be specified as an attribute of <speech> or of the individual child elements.

Attributes:

noint: specifies a time period during which speech input must be started, otherwise the associated <noinput> element is invoked.

norect: specifies a maximum time period during which speech must begin to be matched, otherwise the associated <nomatch> element is invoked.

spcmlt: specifies the length of silence necessary after speech before a result will be finalized in the case where there is a complete match of an active grammar. Following the silence, the appropriate <match> element will be triggered if the result is above the confidence level. Otherwise a <nomatch> element will be triggered.

spincmplt: specifies the length of silence necessary after speech before a result will be finalized in the case where there is an incomplete match of all active grammars. Following the silence, the <nomatch> element will be triggered.

confidence: the minimum confidence level which the recognizer must have to consider a recognition result as matching a grammar. Expressed as an integer between 1-100.

sens: specifies the sensitivity of the recognizer to determine whether speech is present. Lower sensitivity may be required for the recognizer to work well in the presence of high background noise or line echo.
starttimer: boolean value which defines whether the no input (noint) and no recognition (norect) are started initially. When set to false, the starttimer event must be received in order to start them. Default false.

max: specifies the maximum number of times the <grammar>, <noinput>, and <nomatch> elements may be matched unless the element itself specifies differently. The value zero '0' may be used to indicate that there is no maximum. Default is once.

Events:

sens: sets the sensitivity of the recognizer as described above.

starttimer: starts the no input (noint) and no recognition (norect) timers if they have not already been started. Has no effect otherwise.

terminate: terminates the speech input and assigns values to the shadow variables.

Shadow Variables:

speech.end: contains the event which caused the <speech> to terminate or is assigned one of "speech.match", "speech.noinput", or "speech.nomatch" depending upon which of the corresponding elements reached its maximum.

speech.results: contains the results of a matched grammar. The results are formatted using the Natural Language Semantics Markup Language (NLSML) [6]. When this variable is referenced to return results, the results are returned as a separate MIME entity.

4.4.1 Child Elements

4.4.1.1 <grammar>

Specifies and activates a speech grammar based on Speech Recognition Grammar Specification (SRGS) [5] XML notation. Grammars may be referenced by a URI or defined inline. Child elements of <match> are executed when the specified speech grammar is matched.

Attributes:

uri: specifies the location of an SRGS grammar when the grammar is not defined inline.
max: specifies the maximum number of times the <grammar> may be matched. The value zero ‘0’ may be used to indicate that there is no maximum. This value overrides any specified in <speech>. Default is once.

4.4.1.2 <match>

<match> is a child of <grammar> and specifies the actions to take when the corresponding grammar is matched.

4.4.1.3 <noinput>

The <noinput> element is used when speech is being recognized. Children of the <noinput> element are executed when speech has not been detected and the no input timeout (noint) occurs.

Attributes:

max: specifies the maximum number of times the <noinput> may be matched. The value zero ‘0’ may be used to indicate that there is no maximum. This value overrides any specified in <speech>. Default is once.

4.4.1.4 <nomatch>

The <nomatch> element is used when speech is being recognized. Children of the <nomatch> element are executed when it is determined that none of the active grammars will match.

Attributes:

max: specifies the maximum number of times the <nomatch> may be matched. The value zero ‘0’ may be used to indicate that there is no maximum. This value overrides any specified in <speech>. Default is once.

4.4.1.5 <speechexit>

The <speechexit> element is invoked when the speech input completes because one of <grammar>, <noinput>, or <nomatch> occurred its maximum number of times.

Attributes:

none
Voice activity detection (VAD) is used to detect voice and silence when speech recognition is not required. Similar to both speech and DTMF, a VAD has different media conditions which it can match. Those conditions can be qualified by a minimum length of time which is required for them to be considered recognized.

Attributes:

starttimer: boolean value which defines whether the timer is started to allow recognition of the initial condition (voice, silence). When set to false, the starttimer event must be received in order for the initial condition to be recognized. The timer does not affect recognition of the transition conditions. Default false.

Events:

starttimer: starts the timer to allow recognition of the initial condition if it has not already been started. Has no effect otherwise.

terminate: terminates voice activity detection.

Shadow Variables:

none

4.5.1 Child Elements

4.5.1.1 <voice>, <silence>, <tvoice>, <tsilence>

Each child element corresponds to a condition which a VAD can detect. The first two detect when voice or silence has been initially present for a minimum length of time since the VAD was started. The second two require that a transition to the voice or silence condition first occur.

Attributes:

len: the length of time the condition must persist in order to be recognized. In the case of <tvoice> and <tsilence>, the length of time applies only to the final recognized condition.

sen: the maximum length of time the condition not being detected may occur without causing the detector to begin measuring that condition.
4.6 <gain>

Gain is used to adjust the gain of a media stream by a specific amount.

attributes:

  incr: an increment, expressed in dB, which will be used to adjust the gain when "louder" and "softer" events are received. Default is 3 dB.

  amt: a specific gain to apply specified in dB.

events:

  mute: self explanatory.

  unmute: self explanatory.

  reset: sets the gain to zero dB.

  louder: makes the audio on a stream louder.

  softer: makes the audio on a stream quieter.

  amt: sets the gain to the specified value between −96 dB and 9 dB.

4.7 <agc>

Automatic gain control is used to have a media server automatically adjust the gain of a media stream.

attributes:

  tgtlvl: the desired target level for AGC specified in dBm0.

  maxgain: the maximum gain that AGC will apply specified in dB.

events:

  mute: self explanatory.

  unmute: self explanatory.

4.8 <clamp>

This element is used to filter DTMF tones from a media stream. Media other than DTMF tones is passed unchanged.
4.9 <relay>

This element is a simple primitive which copies its input to its output.

attributes:
  none.

events:
  none.

5. Examples

5.1 Announcement

The following is a simple announcement scenario. Two recorded audio files are played in sequence followed by generated speech followed by a variable. The results are reported once media generation completes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0">
  <play>
    <audio uri="file://clip1.wav"/>
    <audio uri="http://host1/clip2.wav"/>
    <tts uri="http://host2/text.ssml"/>
    <var type="date" subtype="mdy" value="20030601"/>
  </play>
  <send target="source" event="done" namelist="play.amt play.end"/>
</moml>
```

5.2 Voice Mail Retrieval

Below is an example which shows a simple voice mail retrieval operation consisting of playing a message and allowing the user to pause and resume play using ‘5’ to toggle the state. The operation would terminate when the play completed or the user entered ‘#’.
During the play, the user can advance forward and backward through the message as well as rewinding to the beginning.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0">
  <group topology="star">
    <play>
      <audio uri="file://message.wav"/>
      <playexit>
        <send target="group" event="terminate"/>
      </playexit>
    </play>
    <dtmf max="0">
      <pattern digits="5">
        <send target="play" event="toggle-state"/>
      </pattern>
      <pattern digits="6">
        <send target="play" event="forward"/>
      </pattern>
      <pattern digits="7">
        <send target="play" event="backward"/>
      </pattern>
      <pattern digits="8">
        <send target="play" event="restart"/>
      </pattern>
      <pattern digits="#">
        <send target="play" event="terminate"/>
      </pattern>
    </dtmf>
  </group>
</moml>
```

5.3 Play and Record

A more complex example is a play and record operation. This sources and sinks media and uses voice activity DTMF detection and recognition to influence behavior. Any DTMF input or voice activity will barge the play and cause the record to begin. However, if the prompt was barged with a DTMF digit of '#', the record terminates without starting. When the play terminates, it send a starttimer event to the VAD to allow it to recognize an initial silence condition. The recording will be terminated (without starting) when the VAD detects an initial 3 seconds of silence.

Once resumed (based upon voice detection) the recording may be terminated under several conditions. It will terminate after 5 seconds of silence or after 60 seconds elapses. It will also terminate if a '#' key is recognized. Every aspect of this behavior
can be modified by changing what is recognized and the events which are sent.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0">
  <group topology="star">
    <play>
      <audio uri="file://prompt.wav"/>
      <playexit>
        <send target="vad" event="starttimer"/>
      </playexit>
    </play>
    <dtmf>
      <pattern digits="#">
        <send target="record" event="stop.termkey"/>
      </pattern>
      <detect>
        <send target="play" event="terminate"/>
      </detect>
    </dtmf>
    <vad>
      <voice>
        <send target="play" event="terminate"/>
        <send target="record" event="resume"/>
      </voice>
      <silence len="PT3S">
        <send target="record" event="nospeech"/>
      </silence>
      <tsilence len="PT5S">
        <send target="record" event="stop.finalsilence"/>
      </tsilence>
    </vad>
    <record initial="suspend" maxtime="PT60S" dest="file://record.wav" format="g729">
      <recordexit>
        <send target="group" event="terminate"/>
      </recordexit>
    </record>
    <groupexit>
      <send target="source" event="done"
           namelist="record.len record.end"/>
    </groupexit>
  </group>
</moml>
```

5.4 Speech Recognition

The following simple example requests that a user speak the name of a city and returns the result.
<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0">
    <group topology="star">
        <play>
            <audio uri="file://prompt.wav"/>
        </play>
        <speech>
            <grammar>
                <rule id="city" scope="public">
                    <item>
                        <one-of>
                            <item>vancouver</item>
                            <item>new york</item>
                            <item>london</item>
                        </one-of>
                    </item>
                </rule>
                <match>
                    <send target="group" event="terminate"/>
                </match>
            </grammar>
            <noinput>
                <send target="group" event="terminate"/>
            </noinput>
            <nomatch>
                <send target="group" event="terminate"/>
            </nomatch>
        </speech>
        <groupexit>
            <send target="source" event="done"
                namelist="speech.end speech.results"/>
        </groupexit>
    </group>
</moml>

5.5 Play and Collect

This example prompts a user to enter 4 DTMF digits terminated by the '#' key. The prompt will be barged and the user has 10 seconds to begin entering input or no input will be indicated.

<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0">
    <group topology="star">
        <play>
            <audio uri="file://prompt.wav"/>
        </play>
        <playexit>
            <send target="dtmf" event="starttimer"/>
        </playexit>
</moml>
5.6 User Controlled Gain

This shows an example of nesting groups to create an arbitrary full duplex media control. DTMF is detected on media flowing in one direction and used to adjust the gain applied to media flowing in the opposite direction. Additionally, the stream which is used to detect DTMF has DTMF removed and its gain automatically adjusted before leaving the group. This widget could be used between a conference participant and a conference mixer.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<moml version="1.0">
  <group topology="fullduplex" lhs="foo" rhs="bar">
    <group topology="star">
      <dtmf>
        <pattern digits="1" max="0">
          <send target="gain" event="louder"/>
        </pattern>
        <pattern digits="2" max="0">
          <send target="gain" event="softer"/>
        </pattern>
      </dtmf>
      <group topology="pipe">
        <clamp/>
        <agc tgtlvl="0"/>
      </group>
    </group>
  </group>
</moml>
```
<gain amt="0" incr="5"/>
</group>
</moml>

6. XML Schema

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"

elementFormDefault="qualified"

coreFormDefault="unqualified">
<xs:redefine schemaLocation="moml-grammar-extension.xsd"/>
<xs:element name="moml">
<xs:complexType>
<xs:choice>
<xs:sequence><xs:element ref="group"/>
<xs:element ref="send"/>
<xs:element ref="play"/>
<xs:element ref="record"/>
<xs:element ref="dtmf"/>
<xs:element ref="speech"/>
<xs:element ref="vad"/>
<xs:element ref="gain"/>
<xs:element ref="agc"/>
<xs:element ref="clamp"/>
<xs:element ref="relay"/>
</xs:choice>
<xs:element ref="exit" minOccurs="0"/>
</xs:sequence></xs:choice>
<xs:attribute name="version" type="xs:string"
use="required" fixed="1.0"/>
<xs:attribute name="id" type="xs:ID" use="optional"/>
</xs:complexType>
</xs:element>
<xs:element name="send">
<xs:complexType>
<xs:attribute name="target" type="xs:string"
use="required"/>
<xs:attribute name="event" type="xs:string" use="required"/>
<xs:attribute name="namelist" type="xs:string"
use="optional"/>
</xs:complexType>
</xs:element>
<xs:element name="group">
<xs:complexType>
<xs:attribute name="name" type="xs:string"
use="required"/>
</xs:complexType>
</xs:element>
<xs:sequence>
  <xs:choice maxOccurs="unbounded">
    <xs:element ref="group"/>
    <xs:element ref="play"/>
    <xs:element ref="record"/>
    <xs:element ref="dtmf"/>
    <xs:element ref="speech"/>
    <xs:element ref="vad"/>
    <xs:element ref="gain"/>
    <xs:element ref="agc"/>
    <xs:element ref="clamp"/>
    <xs:element ref="relay"/>
  </xs:choice>
  <xs:element name="groupexit" type="exitType"
    minOccurs="0"/>
</xs:sequence>
<xs:attribute name="topology" type="topologyType"
  use="required"/>
<xs:attribute name="id" type="xs:ID" use="optional"/>
<xs:attribute name="lhs" type="xs:string" use="optional"/>
<xs:attribute name="rhs" type="xs:string" use="optional"/>
</xs:complexType>
</xs:element>
<xs:element name="play">
  <xs:complexType>
    <xs:sequence>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="audio">
          <xs:complexType>
            <xs:attribute name="uri" type="xs:anyURI"
              use="required"/>
            <xs:attribute name="iterations" type="xs:integer"
              use="optional" default="1"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="tts">
          <xs:complexType>
            <xs:attribute name="uri" type="xs:anyURI"
              use="optional"/>
            <xs:attribute name="iterations" type="xs:integer"
              use="optional" default="1"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="var">
          <xs:complexType>
            <xs:attribute name="type" type="xs:string"
              use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="tts">
  <xs:complexType>
    <xs:attribute name="uri" type="xs:anyURI"
      use="optional"/>
    <xs:attribute name="iterations" type="xs:integer"
      use="optional" default="1"/>
  </xs:complexType>
</xs:element>
<xs:element name="var">
  <xs:complexType>
    <xs:attribute name="type" type="xs:string"
      use="required"/>
  </xs:complexType>
</xs:element>
<xs:complexType>
  <xs:attribute name="subtype" type="xs:string"
    use="optional"/>
  <xs:attribute name="value" type="xs:string"
    use="required"/>
</xs:complexType>
</xs:element>
<xs:element name="dtmfgen">
  <xs:complexType>
    <xs:attribute name="level" use="optional"
      default="-6">;
      <xs:simpleType>
        <xs:restriction
          base="xs:nonPositiveInteger">
          <xs:maxInclusive value="0"/>
          <xs:minInclusive value="-96"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="digits" type="xs:string"
      use="required"/>
    <xs:attribute name="dur" type="xs:nonNegativeInteger"
      use="optional" default="100"/>
    <xs:attribute name="interval" type="xs:nonNegativeInteger"
      use="optional" default="100"/>
  </xs:complexType>
</xs:element>
<xs:choice>
  <xs:element name="playexit" type="exitType"
    minOccurs="0"/>
</xs:choice>
<xs:attribute name="id" type="xs:ID" use="optional"/>
<xs:attribute name="interval" type="xs:duration"
  use="optional" default="PT0S"/>
<xs:attribute name="iterations" type="xs:integer"
  use="optional" default="1"/>
<xs:attribute name="initial" type="playState"
  use="optional" default="generate"/>
<xs:attribute name="maxtime" type="xs:duration"
  use="optional"/>
<xs:attribute name="offset" type="xs:duration"
  use="optional" default="PT0S"/>
<xs:attribute name="skip" type="xs:duration"
  use="optional" default="PT3S"/>
</xs:complexType>
</xs:element>
<xs:element name="record">
  <xs:complexType>
    </xs:complexType>
  </xs:element>
</xs:sequence>
<xs:sequence>
  <xs:element name="recordexit" type="exitType"
    minOccurs="0"/>
</xs:sequence>
<xs:attribute name="id" type="xs:ID" use="optional"/>
<xs:attribute name="append" type="xs:boolean"
  use="optional" default="false"/>
<xs:attribute name="dest" type="xs:anyURI" use="required"/>
<xs:attribute name="format" type="xs:string"
  use="required"/>
<xs:attribute name="initial" type="recordState"
  use="optional"/>
<xs:attribute name="maxtime" type="xs:duration"
  use="required"/>
</xs:complexType>
<xs:element name="dtmf">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="pattern" maxOccurs="unbounded">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="exitType">
              <xs:attribute name="digits" type="xs:string"
                use="required"/>
              <xs:attribute name="format" type="xs:string"
                use="optional"/>
              <xs:attribute name="max" use="optional"
                default="1">1</xs:attribute>
              <xs:simpleType>
                <xs:restriction
                  base="xs:nonNegativeInteger">0</xs:restriction>
                <xs:maxInclusive value="16"/>
              </xs:simpleType>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="detect" type="exitType" minOccurs="0"/>
      <xs:element ref="noinput" minOccurs="0"/>
      <xs:element ref="nomatch" minOccurs="0"/>
      <xs:element name="dtmfexit" type="exitType"
        minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID" use="optional"/>
    <xs:attribute name="cleardb" type="xs:boolean"/>
<xs:attribute name="fdt" type="xs:duration" use="optional"/>
<xs:attribute name="idt" type="xs:duration" use="optional"/>
<xs:attribute name="starttimer" type="xs:boolean"
    use="optional" default="false"/>
<xs:attribute name="max" use="optional" default="1">
    <xs:simpleType>
        <xs:restriction base="xs:nonNegativeInteger">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="16"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:sequence>
<xs:element name="speech">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="grammar" type="ext-grammar"
                maxOccurs="unbounded"/>
            <xs:element ref="noinput" minOccurs="0"/>
            <xs:element ref="nomatch" minOccurs="0"/>
            <xs:element name="speechexit" type="exitType"
                minOccurs="0"/>
        </xs:sequence>
        <xs:attribute name="id" type="xs:ID" use="optional"/>
        <xs:attribute name="noint" type="xs:duration"
            use="optional"/>
        <xs:attribute name="norect" type="xs:duration"
            use="optional"/>
        <xs:attribute name="spcmplt" type="xs:duration"
            use="optional"/>
        <xs:attribute name="spincmplt" type="xs:duration"
            use="optional"/>
        <xs:attribute name="confidence" use="optional">
            <xs:simpleType>
                <xs:restriction base="xs:integer">
                    <xs:minInclusive value="1"/>
                    <xs:maxInclusive value="100"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="sens" type="xs:integer" use="optional"/>
        <xs:attribute name="startTimer" type="xs:boolean"
            use="optional" default="false"/>
        <xs:attribute name="max" use="optional" default="1">
            <xs:simpleType>
                <xs:restriction base="xs:nonNegativeInteger">
                    <xs:minInclusive value="0"/>
                </xs:restriction>
            </xs:simpleType>
        </xs:attribute>
    </xs:complexType>
</xs:element>
<xs:complexType>
  <xs:complexContent>
    <xs:extension base="exitType">
      <xs:attribute name="len" type="xs:duration" use="optional"/>
      <xs:attribute name="sen" type="xs:duration" use="optional"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
<xs:element name="tsilence">
  <xs:complexType>
    <xs:complexContent>
      <xs:extension base="exitType">
        <xs:attribute name="len" type="xs:duration" use="optional"/>
        <xs:attribute name="sen" type="xs:duration" use="optional"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
<xs:element name="vad">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      <xs:element name="voice">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="exitType">
              <xs:attribute name="len" type="xs:duration" use="optional"/>
              <xs:attribute name="sen" type="xs:duration" use="optional"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="silence">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="exitType">
              <xs:attribute name="len" type="xs:duration" use="optional"/>
              <xs:attribute name="sen" type="xs:duration" use="optional"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="tvoice">
        <xs:complexType>
          <xs:complexContent>
            <xs:extension base="exitType">
              <xs:attribute name="len" type="xs:duration" use="optional"/>
              <xs:attribute name="sen" type="xs:duration" use="optional"/>
            </xs:extension>
          </xs:complexContent>
        </xs:complexType>
      </xs:element>
    </xs:choice>
  </xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:simpleType>
</xs:complexType>
</xs:restriction>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:attribute name="len" type="xs:duration" use="optional"/>
<xs:attribute name="sen" type="xs:duration" use="optional"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
</xs:choice>
<xs:attribute name="id" type="xs:ID" use="optional"/>
<xs:attribute name="startTimer" type="xs:boolean" use="optional" default="false"/>
</xs:complexType>
</xs:element>
<xs:element name="gain">
<xs:complexType>
<xs:attribute name="incr" use="optional" default="3">
<xs:simpleType>
<xs:restriction base="xs:nonNegativeInteger">
<xs:minInclusive value="1"/>
<xs:maxInclusive value="96"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="amt" use="required">
<xs:simpleType>
<xs:restriction base="xs:integer">
<xs:minInclusive value="-96"/>
<xs:maxInclusive value="96"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
<xs:element name="agc">
<xs:complexType>
<xs:attribute name="tgtlvl" use="required">
<xs:simpleType>
<xs:restriction base="xs:nonPositiveInteger">
<xs:minInclusive value="-40"/>
<xs:maxInclusive value="0"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="maxgain" use="optional" default="10">
<xs:simpleType>
<xs:restriction base="xs:nonNegativeInteger">
<xs:minInclusive value="0"/>
<xs:maxInclusive value="40"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:complexType>
</xs:element>
<xs:complexType>
  <xs:complexContent>
    <xs:extension base="exitType">
      <xs:attribute name="max" use="optional" default="1">
        <xs:simpleType>
          <xs:restriction base="xs:nonNegativeInteger">
            <xs:minInclusive value="0"/>
            <xs:maxInclusive value="16"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element ref="send" minOccurs="0" maxOccurs="unbounded"/>
<xs:element ref="exit" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
<xs:complexType name="ext-grammar">
<xs:complexContent>
<xs:extension base="grammar">
<xs:sequence>
<xs:element name="match" type="exitType"/>
</xs:sequence>
<xs:attribute name="uri" type="xs:anyURI"
use="optional"/>
<xs:attribute name="max" use="optional" default="1">
<xs:simpleType>
<xs:restriction base="xs:nonNegativeInteger">
<xs:minInclusive value="0"/>
<xs:maxInclusive value="16"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
</xs:extension>
</xs:complexContent>
</xs:complexType>
<xs:complexType name="topologyType">
<xs:restriction base="xs:string">
<xs:enumeration value="fullduplex"/>
<xs:enumeration value="star"/>
<xs:enumeration value="pipe"/>
</xs:restriction>
</xs:simpleType>
<xs:complexType name="playState">
<xs:restriction base="xs:string">
<xs:enumeration value="generate"/>
<xs:enumeration value="suspend"/>
</xs:restriction>
</xs:simpleType>
<xs:complexType name="recordState">
<xs:restriction base="xs:string">
<xs:enumeration value="create"/>
<xs:enumeration value="suspend"/>
</xs:restriction>
</xs:simpleType>
</xs:schema>
Security Considerations

MOML is invoked through other languages and protocols. Its security depends on that provided by those environments.

References


Acknowledgments

Adnan Saleem and Yong Xin, both of Convedia, have provided significant insights, ideas, and contributions to this work; and Gilles Compienne and Ben Smith, both of Ubiquity Software, provided important feedback on a pre-release draft. The authors also wish to thank the other Convedia partners and customers that supplied valuable input into and review of this specification.

Authors’ Addresses

Tim Melanchuk
Convedia
4190 Still Creek Drive, Suite 300
Vancouver, BC, V5C 6C6
Canada

email: timm@convedia.com

Garland Sharratt
Convedia
4190 Still Creek Drive, Suite 300
Vancouver, BC, V5C 6C6
Canada

email: gsharratt@convedia.com

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights
might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on the IETF's procedures with respect to rights in standards-track and standards-related documentation can be found in BCP-11. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF Secretariat.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights which may cover technology that may be required to practice this standard. Please address the information to the IETF Executive Director.

Full Copyright Statement

Copyright (C) The Internet Society 2003. All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this paragraph are included on all such copies and derivative works. However, this document itself may not be modified in any way, such as by removing the copyright notice or references to the Internet Society or other Internet organizations, except as needed for the purpose of developing Internet standards in which case the procedures for copyrights defined in the Internet Standards process must be followed, or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by the Internet Society or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.
Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.