Media Objects Markup Language (MOML)
draft-melanchuk-sipping-moml-03

Status of this Memo

Copyright Notice

Abstract

   The Media Objects Markup Language (MOML) is a modular and extensible
   language to define media processing objects which execute on media
   servers. The base language 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 the base
   MOML, or extend 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

   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
   [7]. 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] or protocols such as the Session Initiation
   Protocol (SIP). 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. SIP is used to establish media
   sessions and there are conventions to use the SIP Request-URI to
   invoke common media server services [9].

   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.

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 is intended to be used in different environments. As such, the
language itself does not define how MOML is used. Each environment in
which MOML is used must define how it is used, the set of services
provided and the mechanism for passing information between the
environment and MOML. The specific mechanisms used to realize the
interface between MOML and its environment are platform specific.

This specification defines using MOML with directly with SIP. The
Media Session Markup Language [8] is an example of another
environment which uses MOML.

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. Results must be explicitly requested using a <send> or
<exit> element within 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.

    o   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.

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

    o   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 "terminate" event class.

The following types of primitives are defined within this
specification:

```
        Recognizers      Transformers      Source/Sink
        ------------------------------------------------
           dtmf              agc              play
          faxtone           clamp            record
          speech            gain             dtmfgen
           vad              gate             faxsend
                            relay             faxrcv
```

Primitives have shadow variables, similar to those within VoiceXML
[7], 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 differs from a
similar mechanism in Media Policy Manipulation in the Conference
Policy Control Protocol [3] which explicitly defined connections. The
topology 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:

   o parallel: 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;

   o serial: 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 parallel topology and that on the right has a
serial topology.

```
      /-> P1 --\
     /          \
G(in) +---> P2 ----> G(out)     G(in) --> P1 --> P2 --> P3 --> G(out)
     \          /
      \-> P3 --/
```
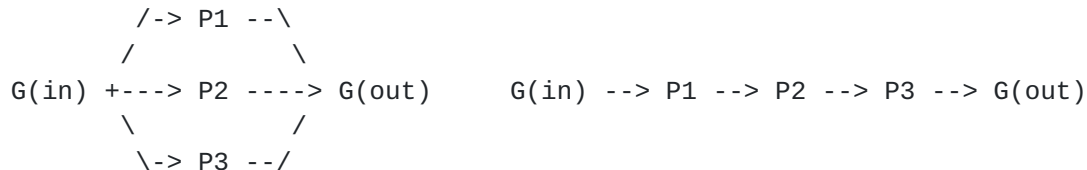
More complex media flows may be created by nesting groups of serial
and parallel topologies within each other. For example, the diagram
below has a group with a serial topology nested within a star
topology.

```
        /-----> P1 -----------------------\
       /                                   \
   Gs(in) +-> Gp(in) --> P2 --> P3 --> Gp(out) -+> Gs(out)
```
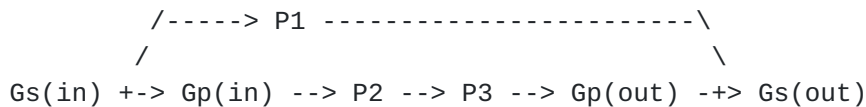
This combination could be used to create record operation where DTMF
was to be clamped from the recording itself, but a DTMF key press is
still used to stop the recording. In this case, P1 would be a DTMF
recognizer, P2 would be a clamp primitive, and P3 a recorder as shown
by the following example. 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.

```
<group topology="parallel">
   <dtmf/>
   <group topology="serial">
      <clamp/>
      <record/>
   </group>
</group>
```

A single schema, "fullduplex" is defined for a two dimensional group.
A full-duplex two dimensional group is 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.

```
        G-A(in1) +-> G2 --> G-B(out1)

        G-A(out2) <-- P1 <-+ G-B(in2)
```
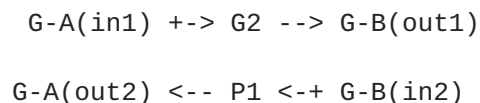
Full duplex groups are symmetrical when both halves are the same.
They are asymmetrical when they differ. Asymmetric groups need to
have a name associated with each side. The left side is defined as
the input of the first child of the full-duplex group combined with
the output of the second child. The right side is reverse. These
sides were labeled 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.

```
<group topology="fullduplex">
   <group topology="parallel">
      <dtmf/>
      <relay/>
   </group>
   <gain/>
</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 terminate event requesting termination.
A terminate event sent to the group causes a terminate event to be
sent to each of its currently active primitives. The <groupexit>
element is not executed until all primitives have processed their
respective terminate 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 often be through MSML but

other languages and protocols such as SIP 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 allow 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 terminate.frowning event when a user looks puzzled. Although no primitive directly defines this event, it will still effect a generic terminate 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. Usage with SIP

MOML may be used directly with SIP for IVR or fax dialog interactions. It can be initially invoked as part of the "Prompt and

Collect" service described in "Basic Network Media Services with SIP"

[9]. That defines service indicators for a small number of well
defined services using the user part of the SIP Request-URI (R-URI).

The prompt and collect service uses "dialog" as the service
indicator. URI parameters further refine the specific IVR request.
This document defines an additional parameter "moml-param" for the
dialog service indicator as follows:

```
dialog-parameters = ";" ( dialog-param [ vxml-parameters ] )
                          | moml-param
dialog-param      = "voicexml=" dialog-url
moml-param        = "moml=" moml-url
```

There are no additional URI parameters when MOML is used as the
dialog language.

MOML defines discrete IVR dialog commands. These commands may be
included directly in the body of the INVITE to the "dialog" service
indicator by using the "cid" [12] URL scheme. This scheme identifies
a message body part which in this case would contain the MOML
command. Note that a multipart message body, containing a single
part, is required even if the INVITE does not contain an SDP offer.
Subsequent MOML requests are sent in the body of SIP INFO messages as
are all messages from a media server.

An example of SIP URI as described above is:

```
   sip:dialog@mediaserver.example.net;\
       moml=cid:14864099865376@appserver.example.net
```

The body part that contained the MOML referenced by the URL would
have a Content-Id header of:

```
   Content-Id: <14864099865376@appserver.example.net>
```

The results of executing an <exit> or <disconnect>, or of executing a
<send> which has a "target" attribute value equal to "source", are
notified in SIP INFO messages using the <event> element. No messages
are sent if execution completes normally without executing one of
these elements.

If there is an error during validation or execution, then a media
server must notify the error as described above and must include the
namelist items "moml.error.status" and "moml.error.description". The
values for these items are defined in section 12.

A restricted subset of MOML can also be used with the "Announcement"
service defined in [9]. This service uses "ann" as the service
indicator and defines parameters that describe an announcement. The

"play=" parameter identifies the URL of a prompt or a provisioned
announcement sequence. The value of the "play=" parameter can refer
to a MOML body part using a "cid" URL as described above. That body
part must only contain the <play> primitive.

Using MOML enhances the announcement service by allowing the client
to specify a sequence of audio segments rather than requiring each
sequence to be provisioned. Moreover, MOML defines a standard set of
variables in contrast to [9] which defines a parameterization
mechanism but does not formally specify any semantics.

If a media server does not understand the "cid" scheme or does not
understand MOML, it must respond with the SIP response code "488 -
not acceptable here". If the MOML body contains elements other than
the <play> primitive, or there are errors during validation, a media
server must respond with a SIP response code "400 - bad request".
Finally, if there is a discrepancy between parameters specified in
the Request-URI and corresponding attributes defined in the MOML
body, the Request-URI parameters must be silently ignored.

Using MOML does not change the operation of the announcement service
from that defined in [9]. When the announcement completes, a media
server issues a SIP BYE request. The INFO method is not used with the
announcement service.

4. Structure and Modularity

MOML is designed to be a modular language. Defining the language in
terms of modules allows different vendors and communities to choose a
specific language subset, or define different language extensions,
for achieving a wide range of applications across a diverse set of
platforms. Modularity combined with namespaces allow independent
development of new extensions.

MOML is structured as a set of modules. Only a single module is
required. That simple core module, moml-core-module, defines a MOML
request to a media server. It consists of the ôprimitiveö
abstraction, an abstract element for control flow, the sequential
execution model, and the <send> element. That is, the core module
allows for the execution of a sequence of one or more media
processing primitives with the ability to notify events to the
invocation environment.

Primitives are divided into four modules. The first, moml-basic-
primitives, defines the basic <play>, <record>, <dtmf>, and <vad>
elements. Another module, moml-transform-primitives, defines the
simple half duplex filters. More advanced primitives are defined in
the speech and fax modules. The speech module depends on the play

module as it extends the capability of <play> by adding synthesized

speech. Finally, the group execution model, which is currently the
only element which changes the flow of control is defined in a
separate module. All of these module are optional although at least
one primitive module is required to have a functional implementation.

The formal process for defining extensions to MOML is to define a new
module. The new module must provide a text description of what
extensions are included and how they work. It must also define an XML
schema file (if applicable) that defines the new module (which may be
through extension or restriction of an existing module). Dependencies
upon other modules must be stated. For example a module that extends
or restricts has a dependency on the original. Finally, the new
module must be assigned a unique name and version.

The types of things which can be defined in new modules are:

   o new primitives

   o extensions to existing primitives (events, shadow variables,
      attributes, content)

   o new recognition grammars for existing primitives

   o new markup languages for speech generation

   o languages for specifying a topology schema

   o new pre-defined topology schemas

   o new variables / segment types (sets & languages)

   o new control flow elements

Modules are assembled together to form a specific MOML profile that
is shared between different implementations. The base MOML profile
which is defined in this documents consists of the moml-core, group,
and basic and transform primitives modules. Speech and facsimile are
examples of optional modules which extend the base language.

Modules which define primitives must define the following for each
primitive within the module:

   o the function which the primitive performs

   o the attributes which may be used to tailor its behavior

   o the events which it is capable of understanding

     o   the shadow variables which provide access to information
         determined as a result of the primitive's operation.

   The mechanism used to insure that a media server and its client share
   a compatible set of modules is not defined. Currently it is expected
   that provisioning will be used, possibly coupled with a future
   auditing capability. Additionally, when used in SIP networks, modules
   could be defined using feature tags and the procedures defined for
   Indicating User Agent Capabilities in SIP [2] used to allow a media
   server to describe its capabilities to other user agents and its
   domain registrar.

5. <moml>

   The root element for MOML. The contents of this element describe
   either a complete execution context for a media resource object or
   the event to be notified to a MOML client.

      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.

6. MOML Core Module

   The core module defines the structural framework and abstractions for
   MOML (via its schema). It also defines the basic elements which are
   not part of the core primitive or control abstractions. These
   elements are defined below.

6.1 Elements Received by a Media Server

6.1.1 <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.

The "target" attribute specifies the recipient of the event. Recipients may be other MOML primitives or groups executing within the object, the object itself, or the environment which invoked MOML. Any target which is unknown within the object is assumed to be destined to the external environment. By convention, the string "source" is used to address that environment but any target name distinct from the MOML namespace may be used.

   Attributes:

      event: the name of an event.

      target: the recipient of the event. The recipient may be a MOML primitive, the currently executing group, or the MOML environment. A primitive is specified by a primitive type, optionally appended by a period '.' followed by the identifier of a primitive. Identifiers are only needed when more than one primitive of the same type exists in the object. The executing group is specified using the token "group". The environment is specified using the token "source", optionally appended by a period '.' followed by any environment specific target.

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

6.1.2 <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.

6.1.3 <disconnect>

Disconnect is similar to <exit> but has the additional semantics of indicating to the context which invoked the MOML object, that it should disconnect from a media server, the media stream associated with the object. The method of disconnection depends upon how the media stream was initially established. If SIP was used, a <disconnect> would cause a media server to issue a BYE request. The request would be sent for the SIP dialog associated with media session on which the MOML object was operating.

   Attributes:

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

6.2 Elements Sent by a Media Server

6.2.1 <event>

The <event> element is used to describe an event and its associated
namelist when MOML is used as a standalone dialog language such as
with SIP. Events are generated and formatted when a <send>, <exit>,
or <disconnect> is executed.

attributes:

name: the type of event. If the event is generated because of
the execution of a <send>, the value must be the value of the
"event" attribute from the <send> element. If the event is
generated because of the execution of an <exit>, the value must
be "moml.exit". If the event is generated because of the
execution of a <disconnect>, the value must be
"moml.disconnect". If the even is generated because of an
error, the value must be "moml.error". Mandatory.

id: the identifier of the MOML object generating the event.
Mandatory.

<event> has two children, <name> and <value>, which contain the name
and value respectively of each namelist item associated with the
event.

7. Group Module

The group module defines a single control flow construct that
specifies concurrent execution. Future modules may define additional
flow control constructs.

7.1 <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.
"parallel" and "serial" 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.

Events:

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

7.2 <groupexit>

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

Attributes:

Events:

8. Basic Primitives Module

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.

8.1 <play>

Play is used to generate an audio stream. It plays in sequence the
media created by the child media elements <audio>, <tts>, and <var>.
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.

Audio may be generated in different languages by specifying the
xml:lang attribute for <play> and/or the child elements of <play>.
The language is inherited by the child elements but each child can
specify its own language. Except for physical audio clips, it is an

error if a language is specified but the media server can not render
the audio in the requested language.

   Attributes:

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

      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.

      iterate: specifies the number of times the media specified by
      the child media elements should be played. Defaults to once
      '1'.

      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).

      xml:lang: specifies the language to use for content which can
      be rendered in different languages.

   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".

8.1.1 Child Elements

8.1.1.1 <audio>

Identifies pre-recorded audio to play. Local URI references may
resolve to a single physical audio clip, a logical clip, or a
provisioned sequence of clips (physical or logical). A logical clip
is one which can be rendered differently based on the language
attribute. Logical clips are provisioned for each of the languages
that a media server supports. Remote URI references are resolved
according to the capabilities of the remote server.

   Attributes:

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

      iterate: specifies the number of times the audio is to be
      played. Defaults to once '1'.

      xml:lang: specifies the language to use when the URI identifies
      a logical clip, either directly, or as part of a sequence.

8.1.1.2 <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.

xml:lang: specifies the language to use when rendering the variable.

8.1.1.3 \<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:

8.2 \<dtmfgen\>

DTMF generator originates one or more DTMF digits in sequence.

Attributes:

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

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.

Events:

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

Shadow Variables:

dtmfgen.end: contains the event which caused DTMF generation to stop.

8.2.1 Child Elements

8.2.1.1 <dtmfgenexit>

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

Attributes:

8.3 <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 terminate event is received or when a nospeech event is received and no audio has yet been recorded. <record> differentiates different types of terminate 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.

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

terminate.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.

terminate.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 terminate. When the record terminates because maxtime is exceeded, end is assigned the value "record.timeexceeded".

8.3.1 Child Elements

8.3.1.1 <recordexit>

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

Attributes:

8.4 <dtmf>

DTMF input fulfils 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 false.

iterate: specifies the number of times the <pattern>, <noinput>, and <nomatch> elements may be executed unless those elements specify differently. The value "forever" may be used to indicate that these may be executed any number of times. 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.

8.4.1 Child Elements

8.4.1.1 <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. Currently, a single built-in format "moml+digits" is defined which allows a match based on either one or more specific digits, or based upon a specific length specification with an optional return key. "moml+digits" is the default.

        iterate: specifies the number of times the <pattern> may be matched. The value "forever" may be used to indicate that <pattern> may be matched any number of times. This value overrides any specified in <dtmf>. Default is once '1'.

8.4.1.2 <detect>

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

    Attributes:

        none

8.4.1.3 <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:

        iterate: specifies the number of times the <noinput> may be triggered. The value "forever" may be used to indicate that <noinput> may be triggered any number of times. This value overrides any specified in <dtmf>. Default is once '1'.

8.4.1.4 <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:

        iterate: specifies the number of times the <nomatch> may be triggered. The value "forever" may be used to indicate that <nomatch> may be triggered any number of times. This value overrides any specified in <dtmf>. Default is once '1'.

8.4.1.5 <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

9. Transform Primitives Module

The transform primitives module gathers together the simple primitives which work as filters on half duplex media streams.

9.1 <vad>

(This is not a transform primitive but not sure where it should go.)

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:

9.1.1 Child Elements

9.1.1.1 <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.

9.2 <gain>

Gain is used to adjust of 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.

9.3 <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.

9.4 <gate>

A simple filter which will pass or halt media, regardless of the format of the media stream, based on the events it receives. <gate> shares the same mute and unmute events for compatibility with the gain primitives <gain> and <agc>.

attributes:

initial: the values "pass" and "halt" define whether media is initially allowed to pass. Default is to pass.

events:

mute: halts media flow through the primitive.

unmute: allows media to pass through the primitive.

9.5 <clamp>

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

attributes:

none.

events:

none.

9.6 <relay>

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

attributes:

none.

events:

none.

10. Speech Module

The speech module defines a standalone primitive for automatic speech recognition <speech> and extends the <play> primitive defined in the basic primitives module to include speech synthesis. As such, this module depends on the basic primitives module.

10.1 <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 in which speech
must begin to be matched, otherwise the associated <nomatch>
element is invoked.

spcmplt: 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 a 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.

iterate: specifies the number of times the <grammar>,
<noinput>, and <nomatch> elements may be executed unless those
elements specify differently. The value "forever" may be used
to indicate that these may be executed any number of times.
Default is once '1'.

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.

10.1.1 Child Elements

10.1.1.1 <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.

iterate: specifies the number of times the <grammar> may be matched. The value "forever" may be used to indicate that <grammar> may be matched any number of times. This value overrides any specified in <speech>. Default is once '1'.

10.1.1.2 <match>

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

10.1.1.3 <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:

iterate: specifies the number of times the <noinput> may be triggered. The value "forever" may be used to indicate that <noinput> may be triggered any number of times. This value overrides any specified in <speech>. Default is once '1'.

10.1.1.4 <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:

      iterate: specifies the maximum number of times the <nomatch> may be triggered. The value "forever" may be used to indicate that <nomatch> may be triggered any number of times. This value overrides any specified in <speech>. Default is once '1'.

10.1.1.5 <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:

10.2 <play>

The <play> element, as defined in the basic primitives module, is extended with a new child element for synthesizing speech. From an XML perspective, <tts> is a member of a media substitution group. See the schema at the end of this document for details.

10.2.1 Child Elements

10.2.1.1 <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.

      iterate: specifies the number of times the text to speech block is to be rendered. Defaults to once '1'.

      xml:lang: specifies the language to use when it is not explicitly specified as an attribute for <speak>.

11. Fax Module

   The fax module defines primitives which allow a media server to
   provide facsimile services.

11.1 <faxdetect>

   Fax tone detection is used to detect the presence of the T.30 CNG
   tone in a media stream. Child elements of <faxtone> are executed when
   the CNG tone is detected.

      Attributes:

         none

11.2 <faxsend>

   The <faxsend> primitive provides the functionality of a calling fax
   terminal. This typically means sending a set of pages. However, it
   can also mean requesting the called terminal to send pages instead
   of, or in addition to, sending pages. The fax images to send are
   defined by the <sendobj> elements, described below.

   Requesting the called terminal to send pages happens when the
   <rxpoll> element is included as part of <faxsend>. This element may
   be included in addition to, or instead of, the <sendobj> element. One
   <sendobj> (at a minimum) or <rxpoll> element must be present. When
   both are present, a media server will first send pages and will then
   poll the other terminal, requesting pages.

   Because fax is a distinct media type, the <faxsend> primitive is not
   expected to interact with other primitives. Rather, it will interact
   using fax protocols with a remote fax terminal (or gateway) and will
   send requested status events to its invoking environment. During fax
   operation, shadow variables are used to record the progress and
   parameters of the varying stages of fax operation.

   Status events are requested by including one or more status request
   elements. These elements correspond to different stages or events in
   fax operation and cause pre-defined events to be sent to the invoking
   environment when they occur. Since the only recipient of these events
   is expected to be a fax application server, requests are simplified
   by associating a pre-defined namelist of shadow variables with each
   event. This decision may be revisited to allowed tailored namelists
   based on further implementation experience. Status requests apply
   both to sending and polling operation.

Attributes:

   lclid: the identifier that a media server uses to identify
   itself.

   minspeed: the minimum acceptable speed to negotiate for the
   operation.

   maxspeed: the maximum speed to negotiate for the operation.
   This attribute is primarily for testing purposes.

   ecm: specifies whether Error Correction Mode (ECM) is allowed
   to be used if supported by the remote terminal. Defaults to
   "true".

Events:

   terminate: terminates the fax send operation.

Shadow Variables:

   fax.rmtid: the identifier of the remote fax terminal.

   fax.rate: the negotiated speed for the operation.

   fax.resolution: identifies the resolution of the image. Both
   metric and inch based resolutions are defined. Metric based
   resolutions are: 75x75, 150x150, 204x98, 204x196, 204x391,
   408x391. Inch based resolutions are: 200x200, 300x300, 400x400,
   600x600.

   fax.pagesize: identifies the negotiated page size. Metric sizes
   are "A3", "A4", "A5", "A6", and "B4". Inch based page sizes are
   "Letter" and "Legal".

   fax.encoding: identifies the image encoding utilized. Valid
   values are "MH", "R", "MMR", and "JPEG".

   fax.ecm: identifies whether ECM operation was used.

   fax.pagebadlines: the number of bad lines in a page.

   fax.objbadlines: the number of bad lines in an object.

   fax.opbadlines: the number of bad lines in an operation.

   fax.objuri: the objuri of the current object.

   fax.resendcount: the number of pages resent due to errors.

fax.totalpages: the number of pages processed or stored.

fax.totalobjects: the count of the objects used in the operation.

fax.duration: the duration of the operation expressed as a duration in seconds and milliseconds (e.g. "23s250ms").

fax.result: contains the reason which caused the fax operation to complete. When the operation completes successfully, the value will be assigned "fax.success". Other values include: "fax.partial", "fax.nofax", "fax.remotedisconnect", "fax.uri.access.error", and "fax.invalid.startpage".

11.2.1 Child Elements

11.2.1.1 <sendobj>

<sendobj> is used to define a fax transmission. There may be multiple instances of the element which will be transmitted in order.

Attributes:

objuri: a URI that points to the fax image that will be transmitted. Mandatory.

startpage: the first page of a multi-page objuri to send.

pagecount: page count.

11.2.1.2 <hdrfooter>

<hdrfooter> describes the header/footer that a media server will put on pages. The header or footer may be defined as the content of the <format> child element. The <format> element is only allowed if the type attribute has a value of "header" or "footer".

Attributes:

type: specifies whether a header or a footer should be put on pages and identifies the source of the header or footer. The following enumerated values may be used:

"header" indicates that the media server should put a header on pages using the contents of the <format> element.

"nohdr" indicates that there should be no header or footer.

          "footer" indicates that the media server should put a footer
                    on pages using the contents of the <format>
                    element.

       style: defines the style of insertion onto a fax page that a
       media server should use for the header or footer. Valid styles
       are "append", "overlay", or "replace".

   <format> is a child of the <hdrfooter> element that defines the style
   format to be used for the header or footer. It uses a "C" language
   style format statement (as shown below) to define the contents and
   layout of the header or footer.

code	length	name	format
%a	3	day of week	3-character abbreviation
%d	2	date	01-31
%m	2	month	01-12
%y	2	year	00-99
%Y	4	year	0000-9999
%I	2	12 hour	01-12
%H	2	24 hour	00-23
%M	2	minute	00-59
%S	2	seconds	00-59
%p	2	AM/PM	AM or PM
%P	2	page number	01-99
%T	2	total pages	01-99
%l	20	local ID (sender)	0-9, + or spaces
%r	20	remote ID (rcvr)	0-9, + or spaces
%%	1	percent	display % in header/ftr

11.2.1.3 <rxpoll>

   <rxpoll> provides the information necessary for a receive polling
   operation to occur. The object(s) to be received are defined by one
   or more <rcvobj> elements. The <rcvobj> is defined further under the
   child elements of <faxrcv>. The <rxpoll> element may also include a
   description of the header/footer that a media server should put on
   received pages. The <hdrfooter> element and it's usage is described
   above.

      Attributes:

       rmtid: specifies the identifier of the remote fax terminal that
       to be associated with a polling operation. A media server must
       not execute a polling operation unless the value of rmtid
       matches that of the connected remote machine.

11.2.1.4 <faxstart>

Requests that an event be sent when fax operation has begun. When
triggered, the following will be executed:

```
<send target="source" event="fax.start"/>
```

11.2.1.5 <faxnegotiate>

Requests that an event be sent when a negotiation has been completed.
Multiple events may be sent each time a DCS frame is sent or
received. When triggered, the following will be executed:

```
<send target="source" event="fax.negotiate"
   namelist="fax.rmtid
      fax.rate
      fax.resolution
      fax.pagesize
      fax.encoding
      fax.ecm"/>
```

11.2.1.6 <faxpagedone>

Requests that an event be sent when a page has been sent or received.
When triggered, the following will be executed:

```
<send target="source" event="fax.pagedone"
      namelist="fax.resolution
      fax.pagesize
      fax.encoding
      fax.pagebadlines
      fax.resendcount"/>
```

11.2.1.7 <faxobjectdone>

Requests that an event be sent when an objuri has been completed.
When triggered, the following will be executed:

```
<send target="source" event="fax.objectdone"
      namelist="fax.objuri
      fax.objbadlines
      fax.resendcount
      fax.totalpages
      fax.result"/>
```

11.2.1.8 <faxopcomplete>

Requests that an event be sent when an operation has been completed.
When triggered, the following will be executed:

```
<send target="source" event="fax.opcomplete"
      namelist="fax.totalpages
      fax.opbadlines
      fax.resendcount
      fax.totalobjects
      fax.duration
      fax.result"/>
```

11.2.1.9 <faxpollstarted>

Requests that an event be sent when a polling operation has started.
When triggered, the following will be executed:

```
<send target="source" event="fax.opcomplete"
      namelist="fax.rmtid
      fax.rate
      fax.resolution
      fax.pagesize
      fax.encoding
      fax.ecm"/>
```

11.3 <faxrcv>

The <faxrcv> primitive provides the functionality of a called fax
terminal. Typically this type of operation is to receive pages.
However, it can include sending pages instead of, or in addition to,
receiving them. The fax objects to receive are defined by the
<rcvobj> elements, described below.

A media server will send pages as a polled terminal when the <txpoll>
element is included as part of <faxrcv>. This element may be included
in addition to, or instead of, the <rcvobj> element. One <rcvobj> or
<txpoll> element must be present. When both are present, a media
server will first receive pages and will then allow the other
terminal to poll the media server, requesting pages.

Because fax is a distinct media type, the <faxrcv> primitive is not
expected to interact with other primitives. Rather, it will interact
using fax protocols with a remote fax terminal and will send
requested status events to its invoking environment. During fax
operation, shadow variables are used to record the progress and
parameters of the varying stages of fax operation.

Status events are requested by including one or more status request
elements. These elements correspond to different stages or events in
fax operation and cause pre-defined events to be sent to the invoking
environment when they occur. Since the only recipient of these events
is expected to be a fax application server, requests are simplified
by associating a pre-defined namelist of shadow variables with each

event. This decision may be revisited to allowed tailored namelists
based on further implementation experience. Status requests apply
both to receiving and polling operation.

    Attributes:

        lclid: the identifier that a media server uses to identify
        itself.

        ecm: specifies whether ECM mode is allowed to be used if
        supported by the remote terminal. Defaults to "true".

    Events:

        terminate: terminates the fax reception operation.

    Shadow Variables:

        <faxrcv> supports the same set of shadow variables as <faxsend>

11.3.1 Child Elements

In addition to the elements defined below, <faxrcv> may also have the
following child elements which were defined under <faxsend>:

    o <hdrfooter>
    o <faxstart>
    o <faxnegotiate>
    o <faxpagedone>
    o <faxobjectdone>
    o <faxopcomplete>
    o <faxpollstarted>

Their meaning and usage is the same as previously defined.

11.3.1.1 <rcvobj>

<rcvobj> is used to define fax objects that a media server will
receive. There may be multiple instances of the element which will be
used in order.

    Attributes:

        objuri: a URI that points to the location that a received image
        is to be stored. Mandatory.

        maxpages: the maximum number of pages that will be stored in
        objuri.

11.3.1.2 **<txpoll>**

<txpoll> provides the information for a polling operation to occur as part of a fax receive operation. Multiple object(s) to be send may be supplied by one or more <sendobj> elements. In the event of multiple occurrences, a media server must select the <sendobj> element whose rmtid attribute matches that of the remote terminal.

The <sendobj> element was defined previously as a child element of <faxsend>. For <txpoll> is extended with an rmtid attribute that specifies the identifier of the remote fax terminal and is used to select the specific <sendobj> to send.

A media server will put a header/footer on transmitted pages based on any <hdrfooter> element included as part of <txpoll>.

Attributes:

12. **Failure Codes**

Failure codes are used to indicate reasons for failures. The appropriate code and description must be passed to the invoking environment on failure.

Request Error (4xx)

400 Bad Request
401 Unknown Element
402 Unsupported Element
403 Missing mandatory element content
404 Forbidden element content
405 Invalid element content
406 Unknown attribute
407 Attribute not supported
408 Missing mandatory attribute
409 Forbidden attribute is present
410 Invalid attribute value

Server Error (5xx)

500 Internal media server error
510 Not in service
511 Service Unavailable
520 No resource to fulfill request
521 Internal limit exceeded

13. Examples

13.1 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 version="1.0" encoding="UTF-8"?>
<moml version="1.0" id="12345">
   <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>
```

13.2 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 version="1.0" encoding="UTF-8"?>
<moml version="1.0" id="12345">
   <group topology="parallel">
      <play>
         <audio uri="file://message.wav"/>
         <playexit>
            <send target="group" event="terminate"/>
         </playexit>
      </play>
      <dtmf iterate="forever">
         <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>
```

13.3 **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 version="1.0" encoding="UTF-8"?>
<moml  version="1.0" id="12345">
   <group topology="parallel">
      <play>
         <audio uri="file://prompt.wav"/>
         <playexit>
            <send target="vad" event="starttimer"/>
         </playexit>
      </play>
      <dtmf>
         <pattern digits="#">
            <send target="record" event="terminate.termkey"/>
         </pattern>
         <detect>
            <send target="play" event="terminate"/>
         </detect>
      </dtmf>
      <vad>
         <voice len="10ms">
            <send target="play" event="terminate"/>
            <send target="record" event="resume"/>
```

```
        </voice>
```

```
          <silence len="3s">
             <send target="record" event="nospeech"/>
          </silence>
          <tsilence len="5s">
             <send target="record" event="terminate.finalsilence"/>
          </tsilence>
      </vad>
      <record initial="suspend" maxtime="60s"
             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>
```

13.4 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" id="12345">
   <group topology="parallel">
      <play>
          <audio uri="file://prompt.wav"/>
      </play>
      <speech>
          <grammar version="1.0">
             <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>
```

13.5 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" id="12345">
    <group topology="parallel">
        <play>
            <audio uri="file://prompt.wav"/>
            <playexit>
                <send target="dtmf" event="starttimer"/>
            </playexit>
        </play>
        <dtmf fdt="10s" idt="16s">
            <pattern digits="xxxx#">
                <send target="group" event="terminate"/>
            </pattern>
            <detect>
                <send target="play" event="terminate"/>
            </detect>
            <noinput>
                <send target="group" event="terminate"/>
            </noinput>
            <nomatch>
                <send target="group" event="terminate"/>
            </nomatch>
        </dtmf>
        <groupexit>
            <send target="source" event="done"
                  namelist="dtmf.digits dtmf.end"/>
        </groupexit>
    </group>
</moml>
```

13.6 **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" id="12345">
   <group topology="fullduplex">
      <group topology="parallel">
         <dtmf>
            <pattern digits="1" iterate="forever">
               <send target="gain" event="louder"/>
            </pattern>
            <pattern digits="2" iterate="forever">
               <send target="gain" event="softer"/>
            </pattern>
         </dtmf>
         <group topology="serial">
            <clamp/>
            <agc tgtlvl="0"/>
         </group>
      </group>
      <gain amt="0" incr="5"/>
   </group>
</moml>
```

14. Change Summary

   The following are the primary changes between this version of the
   draft and the -01 version:

   o   specified the use of MOML directly in SIP (see section 3)

   o   specified the <event> element for notifying events in SIP INFO
       messages

   o   added <gate> transform primitive which can gate the flow of
       media regardless of its format

   o   extended sending events to "source" to allow event names
       specific to the source naming conventions to be included. This
       can allow such features as the source relaying an event to
       another MOML object it has created.

    o modularized the language and specified rules to extend it in
       order to allow it to be independently tailored to different
       environments and platforms (see [section 4](#))

  Between the -01 version and the -00 version the changes were:

    o added primitives to detect, send, and receive fax

    o added "xml:lang" attribute to <play> <audio> <var> and <tts>.
       children of <play> inherit from play unless overridden.

    o allow the uri in <audio> to refer to a logical clip (physical
       determined by language) and sequence as well as a physical clip
       (for local uri references).

    o restructured the schema as a partial step towards
       modularization and the ability to subset and extend the
       language in a standards compliant manner.

    o made <dtmfgen> to be the same level as <play> and not a child
       of <play>

    o changed "pipe" and "star" to be "serial" and "parallel"

    o made all termination events consistently use the root
       "terminate". previously some primitives used the root "stop"

    o changed "max" attribute to "iterate" for the <dtmf>, <pattern>,
       <noinput>, and <nomatch>, and <speech> elements.

    o change "iterations" attribute of <play> and <audio> to
       "iterate".

    o removed explicit "lhs" / "rhs" labeling of full duplex objects

[15](#). Future Work

  Some of the likely functions to be added in future release of MOML
  include:

    o further refinement to the schemas with respect to
       modularization

    o algorithmic tone generation and detection

    o video and multimedia

16. XML Schema

   The base MOML schema defines the <moml> element and includes all of
   the modules which together define the full language. The <moml>
   element defines that a given document may be either a request to a
   media server or an event notified by a media server.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="unqualified" attributeFormDefault="unqualified">
   <xs:include schemaLocation="moml-group-module.xsd"/>
   <xs:include schemaLocation="moml-basic-primitives-module.xsd"/>
   <xs:include
       schemaLocation="moml-transform-primitives-module.xsd"/>
   <xs:include schemaLocation="moml-speech-module.xsd"/>
   <xs:include schemaLocation="moml-fax-module.xsd"/>
   <xs:element name="moml">
      <xs:complexType>
         <xs:choice>
            <xs:group ref="momlRequest"/>
            <xs:element ref="event"/>
         </xs:choice>
         <xs:attribute name="version" type="xs:string"
             use="required" fixed="1.0"/>
         <xs:attribute name="id" type="momlID.datatype"
             use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

   Following is the schema which defines the core module (moml-core-
   module.xsd). It is included by each of the other MOML modules.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
   <xs:include schemaLocation="moml-datatypes.xsd"/>
   <xs:group name="momlRequest">
      <xs:choice maxOccurs="unbounded">
         <xs:group ref="executeType"/>
         <xs:element ref="send" maxOccurs="unbounded"/>
      </xs:choice>
   </xs:group>
   <xs:element name="primitive" type="primitiveType"
       abstract="true"/>
   <xs:complexType name="primitiveType">
      <xs:attribute name="id" type="momlID.datatype"/>
```

```
        </xs:complexType>
```

```
   <xs:element name="control" type="controlType" abstract="true"/>
   <xs:complexType name="controlType"/>
   <xs:group name="executeType">
      <xs:choice maxOccurs="unbounded">
         <xs:element ref="primitive"/>
         <xs:element ref="control"/>
      </xs:choice>
   </xs:group>
   <xs:group name="sendType">
      <xs:choice>
         <xs:choice>
            <xs:element name="exit" type="exitType"/>
            <xs:element name="disconnect" type="exitType"/>
         </xs:choice>
         <xs:sequence>
            <xs:element ref="send" maxOccurs="unbounded"/>
            <xs:choice minOccurs="0">
               <xs:element name="exit" type="exitType"/>
               <xs:element name="disconnect" type="exitType"/>
            </xs:choice>
         </xs:sequence>
      </xs:choice>
   </xs:group>
   <xs:element name="send">
      <xs:complexType>
         <xs:attribute name="event" type="momlEvent.datatype"
             use="required"/>
         <xs:attribute name="target" type="momlTarget.datatype"
             use="required"/>
         <xs:attribute name="namelist" type="momlNamelist.datatype"/>
      </xs:complexType>
   </xs:element>
   <xs:complexType name="exitType">
      <xs:attribute name="namelist" type="momlNamelist.datatype"/>
   </xs:complexType>
   <xs:element name="event">
      <xs:complexType>
         <xs:sequence maxOccurs="unbounded">
            <xs:element name="name" type="xs:string"/>
            <xs:element name="value" type="xs:string"/>
         </xs:sequence>
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="id" type="xs:string" use="required"/>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

The schema for the group module (moml-group-module.xsd) is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
  <xs:include schemaLocation="moml-datatypes.xsd"/>
  <xs:include schemaLocation="moml-core-module.xsd"/>
  <xs:element name="group" substitutionGroup="control">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="executeType"/>
        <xs:element name="groupexit" minOccurs="0">
          <xs:complexType>
            <xs:group ref="sendType"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="id" type="momlID.datatype"/>
      <xs:attribute name="topology" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="serial"/>
            <xs:enumeration value="parallel"/>
            <xs:enumeration value="fullduplex"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The schema for the basic primitives module (moml-basic-primitives-module.xsd) is:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified">
  <xs:include schemaLocation="moml-datatypes.xsd"/>
  <xs:include schemaLocation="moml-core-module.xsd"/>
  <xs:import namespace="http://www.w3.org/XML/1998/namespace"
      schemaLocation="http://www.w3.org/2001/xml.xsd"/>
  <xs:element name="play" substitutionGroup="primitive">
    <xs:complexType>
      <xs:complexContent>
        <xs:extension base="primitiveType">
          <xs:sequence>
            <xs:choice maxOccurs="unbounded">
              <xs:element name="audio">
```

```
                        <xs:complexType>
```

```
                              <xs:attribute name="uri" type="xs:anyURI"
                                  use="required"/>
                              <xs:attribute name="iterate"
                                  type="iterate.datatype" default="1"/>
                              <xs:attribute ref="xml:lang"/>
                          </xs:complexType>
                      </xs:element>
                      <xs:element ref="media"/>
                  </xs:choice>
                  <xs:choice minOccurs="0">
                      <xs:element name="playexit">
                          <xs:complexType>
                              <xs:group ref="sendType"/>
                          </xs:complexType>
                      </xs:element>
                  </xs:choice>
              </xs:sequence>
              <xs:attribute name="interval"
                  type="posDuration.datatype" use="optional"/>
              <xs:attribute name="iterate" type="iterate.datatype"
                  use="optional" default="1"/>
              <xs:attribute name="offset" type="duration.datatype"
                  use="optional"/>
              <xs:attribute name="initial" use="optional"
                  default="generate">
                  <xs:simpleType>
                      <xs:restriction base="xs:string">
                          <xs:enumeration value="generate"/>
                          <xs:enumeration value="suspend"/>
                      </xs:restriction>
                  </xs:simpleType>
              </xs:attribute>
              <xs:attribute name="maxtime"
                  type="posDuration.datatype" use="optional"/>
              <xs:attribute name="skip" type="duration.datatype"
                  use="optional" default="3s"/>
              <xs:attribute ref="xml:lang"/>
          </xs:extension>
      </xs:complexContent>
  </xs:complexType>
</xs:element>
<xs:element name="record" substitutionGroup="primitive">
  <xs:complexType>
      <xs:choice minOccurs="0">
          <xs:element name="recordexit">
              <xs:complexType>
                  <xs:group ref="sendType"/>
              </xs:complexType>
```

```
          </xs:element>
```

```
        </xs:choice>
        <xs:attribute name="append" type="boolean.datatype"
            use="optional" default="false"/>
        <xs:attribute name="dest" type="xs:anyURI" use="optional"/>
        <xs:attribute name="format" use="required">
           <xs:simpleType>
              <xs:restriction base="xs:string"/>
           </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="maxtime" type="posDuration.datatype"
            use="required"/>
        <xs:attribute name="initial" use="optional"
            default="create">
           <xs:simpleType>
              <xs:restriction base="xs:string">
                 <xs:enumeration value="create"/>
                 <xs:enumeration value="suspend"/>
              </xs:restriction>
           </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
</xs:element>
<xs:element name="dtmf" substitutionGroup="primitive">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="primitiveType">
            <xs:sequence>
               <xs:element name="pattern" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:group ref="sendType"/>
                     <xs:attribute name="digits" type="xs:string"
                         use="required"/>
                     <xs:attribute name="format">
                        <xs:simpleType>
                           <xs:restriction base="xs:string">
                              <xs:enumeration value="mgcp"/>
                              <xs:enumeration value="megaco"/>
                              <xs:enumeration
                                  value="moml+digits"/>
                           </xs:restriction>
                        </xs:simpleType>
                     </xs:attribute>
                     <xs:attribute name="iterate"
                         type="iterate.datatype" default="1"/>
                  </xs:complexType>
               </xs:element>
               <xs:element name="detect" minOccurs="0">
                  <xs:complexType>
```

```
                        <xs:group ref="sendType"/>
```

```
                        </xs:complexType>
                    </xs:element>
                    <xs:element name="noinput" type="iterateSendType"
                        minOccurs="0"/>
                    <xs:element name="nomatch" type="iterateSendType"
                        minOccurs="0"/>
                    <xs:element name="dtmfexit" minOccurs="0">
                        <xs:complexType>
                            <xs:group ref="sendType"/>
                        </xs:complexType>
                    </xs:element>
                </xs:sequence>
                <xs:attribute name="cleardb" type="boolean.datatype"
                    default="true"/>
                <xs:attribute name="fdt" type="posDuration.datatype"
                    default="0s"/>
                <xs:attribute name="idt" type="posDuration.datatype"
                    default="4s"/>
                <xs:attribute name="edt" type="posDuration.datatype"
                    default="4s"/>
                <xs:attribute name="starttimer"
                    type="boolean.datatype" default="false"/>
                <xs:attribute name="iterate" type="iterate.datatype"
                    default="1"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:element>
<xs:element name="dtmfgen" substitutionGroup="primitive">
    <xs:complexType>
        <xs:choice minOccurs="0">
            <xs:element name="dtmfgenexit">
                <xs:complexType>
                    <xs:group ref="sendType"/>
                </xs:complexType>
            </xs:element>
        </xs:choice>
        <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="dtmfDigits.datatype"
            use="required"/>
        <xs:attribute name="dur" type="posDuration.datatype"
```

```
                 use="optional" default="100ms"/>
```

```
            <xs:attribute name="interval" type="posDuration.datatype"
                use="optional" default="100ms"/>
        </xs:complexType>
    </xs:element>
    <xs:complexType name="iterateSendType">
        <xs:group ref="sendType"/>
        <xs:attribute name="iterate" type="iterate.datatype"
            default="1"/>
    </xs:complexType>
    <xs:element name="media" type="mediaType" abstract="true"/>
    <xs:complexType name="mediaType">
        <xs:attribute ref="xml:lang" type="xs:language"/>
        <xs:attribute name="iterate" type="iterate.datatype"/>
    </xs:complexType>
    <xs:element name="var" substitutionGroup="media">
        <xs:complexType>
            <xs:complexContent>
                <xs:extension base="mediaType">
                    <xs:attribute name="type" use="required">
                        <xs:simpleType>
                            <xs:restriction base="xs:string">
                                <xs:enumeration value="date"/>
                                <xs:enumeration value="digits"/>
                                <xs:enumeration value="duration"/>
                                <xs:enumeration value="month"/>
                                <xs:enumeration value="money"/>
                                <xs:enumeration value="number"/>
                                <xs:enumeration value="silence"/>
                                <xs:enumeration value="time"/>
                                <xs:enumeration value="weekday"/>
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="subtype" type="xs:string"
                        use="optional"/>
                    <xs:attribute name="value" type="xs:string"
                        use="required"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

The schema for the transform primitives module (moml-transform-primitives-module.xsd) is:

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

```
        elementFormDefault="unqualified"
```

```
 attributeFormDefault="unqualified">
<xs:include schemaLocation="moml-datatypes.xsd"/>
<xs:include schemaLocation="moml-core-module.xsd"/>
<xs:import namespace="http://www.w3.org/XML/1998/namespace"
    schemaLocation="http://www.w3.org/2001/xml.xsd"/>
<xs:element name="vad" substitutionGroup="primitive">
   <xs:complexType>
      <xs:all>
         <xs:element name="voice" type="vadPatternType"
            minOccurs="0"/>
         <xs:element name="silence" type="vadPatternType"
            minOccurs="0"/>
         <xs:element name="tvoice" type="vadPatternType"
            minOccurs="0"/>
         <xs:element name="tsilence" type="vadPatternType"
            minOccurs="0"/>
      </xs:all>
      <xs:attribute name="starttimer" type="boolean.datatype"
         default="false"/>
   </xs:complexType>
</xs:element>
<xs:element name="gain" substitutionGroup="primitive">
   <xs:complexType>
      <xs:attribute name="incr" default="3">
         <xs:simpleType>
            <xs:restriction base="xs:positiveInteger">
               <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:element name="agc" substitutionGroup="primitive">
   <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" 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:element name="gate" substitutionGroup="primitive">
         <xs:complexType>
            <xs:attribute name="initial" default="pass">
               <xs:simpleType>
                  <xs:restriction base="xs:string">
                     <xs:enumeration value="pass"/>
                     <xs:enumeration value="halt"/>
                  </xs:restriction>
               </xs:simpleType>
            </xs:attribute>
         </xs:complexType>
      </xs:element>
      <xs:element name="clamp" substitutionGroup="primitive">
         <xs:complexType/>
      </xs:element>
      <xs:element name="relay" substitutionGroup="primitive">
         <xs:complexType/>
      </xs:element>
      <xs:complexType name="vadPatternType">
         <xs:group ref="sendType"/>
         <xs:attribute name="iterate" type="iterate.datatype"
            default="1"/>
         <xs:attribute name="len" type="posDuration.datatype"
            use="required"/>
         <xs:attribute name="sen" type="posDuration.datatype"
            use="optional"/>
      </xs:complexType>
</xs:schema>
```

Following is the schema for the speech primitives module (moml-speech-module.xsd). Note that several URL were split across several lines for formatting reasons.

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

```
<xs:include schemaLocation="moml-datatypes.xsd"/>
```

```
<xs:include schemaLocation="moml-core-module.xsd"/>
<xs:include schemaLocation="moml-basic-primitives-module.xsd"/>
<xs:include schemaLocation="http://www.w3.org/TR/2002/
    WD-speech-synthesis-20020405/synthesis-core.xsd"/>
<xs:include schemaLocation="http://www.w3.org/TR/
    speech-grammar/grammar-core.xsd"/>
<xs:element name="speech" substitutionGroup="primitive">
   <xs:complexType>
      <xs:complexContent>
         <xs:extension base="primitiveType">
            <xs:sequence>
               <xs:element name="grammar" maxOccurs="unbounded">
                  <xs:complexType>
                     <xs:complexContent>
                        <xs:extension base="grammar">
                           <xs:choice>
                              <xs:element name="match"
                                   type="iterateSendType"
                                   minOccurs="0"/>
                           </xs:choice>
                           <xs:attribute name="uri"
                                type="xs:anyURI"/>
                           <xs:attribute name="iterate"
                                type="iterate.datatype"
                                default="1"/>
                        </xs:extension>
                     </xs:complexContent>
                  </xs:complexType>
               </xs:element>
               <xs:element name="noinput" type="iterateSendType"
                  minOccurs="0"/>
               <xs:element name="nomatch" type="iterateSendType"
                  minOccurs="0"/>
               <xs:element name="speechexit" minOccurs="0">
                  <xs:complexType>
                     <xs:group ref="sendType"/>
                  </xs:complexType>
               </xs:element>
            </xs:sequence>
            <xs:attribute name="noint"
                type="posDuration.datatype"/>
            <xs:attribute name="norect"
                type="posDuration.datatype"/>
            <xs:attribute name="spcmplt"
                type="posDuration.datatype"/>
            <xs:attribute name="confidence">
               <xs:simpleType>
                  <xs:restriction base="xs:positiveInteger">
```

```
                    <xs:maxInclusive value="100"/>
```

```
                            </xs:restriction>
                        </xs:simpleType>
                    </xs:attribute>
                    <xs:attribute name="sens" type="xs:positiveInteger"/>
                    <xs:attribute name="starttimer"
                        type="boolean.datatype" default="false"/>
                    <xs:attribute name="iterate" type="iterate.datatype"
                        default="1"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
    <xs:element name="tts" substitutionGroup="media">
        <xs:complexType mixed="true">
            <xs:complexContent mixed="true">
                <xs:extension base="mediaType">
                    <xs:choice minOccurs="0">
                        <xs:element ref="speak"/>
                    </xs:choice>
                    <xs:attribute name="uri" type="xs:anyURI"
                        use="required"/>
                </xs:extension>
            </xs:complexContent>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Following is the schema for the fax primitives module (moml-fax-module.xsd).

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:include schemaLocation="moml-datatypes.xsd"/>
    <xs:include schemaLocation="moml-core-module.xsd"/>
    <xs:element name="faxdetect" substitutionGroup="primitive">
        <xs:complexType>
            <xs:choice>
                <xs:group ref="sendType"/>
            </xs:choice>
        </xs:complexType>
    </xs:element>
    <xs:element name="faxsend" substitutionGroup="primitive">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="sendobj" type="sendobjType"
                    minOccurs="0" maxOccurs="unbounded"/>
```

```
<xs:element name="hdrfooter" type="hdrfooterType"
```

```
                  minOccurs="0"/>
          <xs:element name="rxpoll" minOccurs="0">
             <xs:complexType>
                <xs:sequence>
                   <xs:element name="rcvobj" type="rcvobjType"
                      maxOccurs="unbounded"/>
                   <xs:element name="hdrfooter"
                      type="hdrfooterType" minOccurs="0"/>
                </xs:sequence>
                <xs:attribute name="rmtid" type="faxid.datatype"
                    use="required"/>
             </xs:complexType>
          </xs:element>
          <xs:group ref="faxstatusrequest"/>
       </xs:sequence>
       <xs:attribute name="lclid" type="faxid.datatype"
          use="optional"/>
       <xs:attribute name="minspeed" type="faxspeed.datatype"
          use="optional"/>
       <xs:attribute name="maxspeed" type="faxspeed.datatype"
          use="optional"/>
       <xs:attribute name="ecm" type="boolean.datatype"
          use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="faxrecv" substitutionGroup="primitive">
    <xs:complexType>
       <xs:sequence>
          <xs:element name="rcvobj" type="rcvobjType" minOccurs="0"
             maxOccurs="unbounded"/>
          <xs:element name="hdrfooter" type="hdrfooterType"
             minOccurs="0"/>
          <xs:element name="txpoll" minOccurs="0">
             <xs:complexType>
                <xs:sequence>
                   <xs:element name="sendobj" type="sendobjType"
                      maxOccurs="unbounded"/>
                   <xs:element name="hdrfooter"
                      type="hdrfooterType" minOccurs="0"/>
                </xs:sequence>
                <xs:attribute name="rmtid" type="faxid.datatype"/>
             </xs:complexType>
          </xs:element>
          <xs:group ref="faxstatusrequest"/>
       </xs:sequence>
       <xs:attribute name="lclid" type="faxid.datatype"
           use="optional"/>
       <xs:attribute name="ecm" type="boolean.datatype"
```

```
                   default="true"/>
```

```
        </xs:complexType>
    </xs:element>
    <xs:group name="faxstatusrequest">
        <xs:all>
            <xs:element name="faxstart" minOccurs="0"/>
            <xs:element name="faxnegotiate" minOccurs="0"/>
            <xs:element name="faxpagedone" minOccurs="0"/>
            <xs:element name="faxobjectdone" minOccurs="0"/>
            <xs:element name="faxopcomplete" minOccurs="0"/>
            <xs:element name="faxpollstart" minOccurs="0"/>
        </xs:all>
    </xs:group>
    <xs:complexType name="hdrfooterType">
        <xs:choice>
            <xs:element name="format" type="xs:string" minOccurs="0"
                maxOccurs="unbounded"/>
        </xs:choice>
        <xs:attribute name="type" type="hdrfooter.datatype"/>
        <xs:attribute name="style" type="hdrfooterstyle.datatype"/>
    </xs:complexType>
    <xs:complexType name="formatType">
        <xs:simpleContent>
            <xs:extension base="xs:string">
                <xs:attribute name="style">
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:enumeration value="append"/>
                            <xs:enumeration value="overlay"/>
                            <xs:enumeration value="replace"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:attribute>
            </xs:extension>
        </xs:simpleContent>
    </xs:complexType>
    <xs:complexType name="rcvobjType">
        <xs:attribute name="objuri" type="xs:anyURI" use="required"/>
        <xs:attribute name="maxpages" type="xs:positiveInteger"/>
    </xs:complexType>
    <xs:complexType name="sendobjType">
        <xs:attribute name="objuri" type="xs:anyURI" use="required"/>
        <xs:attribute name="startpage" type="xs:positiveInteger"/>
        <xs:attribute name="pagecount" type="xs:positiveInteger"/>
    </xs:complexType>
    <xs:simpleType name="faxid.datatype">
        <xs:restriction base="xs:string">
            <xs:pattern value="[0-9+*- ]{20}"/>
        </xs:restriction>
```

```
        </xs:simpleType>
```

```
    <xs:simpleType name="faxspeed.datatype">
       <xs:restriction base="xs:string">
          <xs:enumeration value="2400"/>
          <xs:enumeration value="4800"/>
          <xs:enumeration value="7200"/>
          <xs:enumeration value="9600"/>
          <xs:enumeration value="12000"/>
          <xs:enumeration value="14400"/>
       </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="hdrfooter.datatype">
       <xs:restriction base="xs:string">
          <xs:enumeration value="header"/>
          <xs:enumeration value="footer"/>
          <xs:enumeration value="autohdr"/>
          <xs:enumeration value="nohdr"/>
       </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="hdrfooterstyle.datatype">
       <xs:restriction base="xs:string">
          <xs:enumeration value="append"/>
          <xs:enumeration value="overlay"/>
          <xs:enumeration value="replace"/>
       </xs:restriction>
    </xs:simpleType>
</xs:schema>
```

Following is the schema which defines the basic datatypes used by the
other schemas.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
    <xs:simpleType name="momlID.datatype">
       <xs:restriction base="xs:string">
          <xs:pattern value="[a-zA-Z0-9][a-zA-Z0-9._\-]*"/>
       </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="momlEvent.datatype">
       <xs:restriction base="xs:string">
          <xs:pattern value="[a-zA-Z0-9][a-zA-Z0-9._\-]*"/>
       </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="momlNamelist.datatype">
       <xs:restriction base="xs:string"/>
    </xs:simpleType>
    <xs:simpleType name="dtmfDigits.datatype">
```

```
<xs:restriction base="xs:string">
```

```
                <xs:pattern value="[0-9#*]+"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="iterate.datatype">
            <xs:union memberTypes="xs:positiveInteger">
                <xs:simpleType>
                    <xs:restriction base="xs:negativeInteger">
                        <xs:minInclusive value="-1"/>
                    </xs:restriction>
                </xs:simpleType>
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                        <xs:enumeration value="forever"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:union>
        </xs:simpleType>
        <xs:simpleType name="momlTarget.datatype">
            <xs:restriction base="xs:string">
                <xs:pattern value="[a-zA-Z0-9][a-zA-Z0-9._\-]*"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="boolean.datatype">
            <xs:restriction base="xs:string">
                <xs:enumeration value="true"/>
                <xs:enumeration value="false"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="duration.datatype">
            <xs:restriction base="xs:string">
                <xs:pattern value="(\+|\-)?([0-9]*\.)?[0-9]+(ms|s)"/>
            </xs:restriction>
        </xs:simpleType>
        <xs:simpleType name="posDuration.datatype">
            <xs:restriction base="xs:string">
                <xs:pattern value="(\+)?([0-9]*\.)?[0-9]+(ms|s)"/>
            </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

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol", RFC3261, Internet Engineering Taskforce, June 2002.

[2] J. Rosenberg, H. Schulzrinne, and P. Kyzivat, "Indicating User Agent Capabilities in the Session Initiation Protocol (SIP)", Internet Engineering Taskforce, December 2003. Work in progress.

[3] R. Mahy and N. Ismail, "Media Policy Manipulation in the Conference Policy Control Protocol", Internet Draft, Internet Engineering Taskforce, Feb. 2003. Work in progress.

[4] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation, Oct. 2000.

[5] World Wide Web Consortium, "Speech Recognition Grammar Specification Version 1.0" (SRGS), W3C Candidate Recommendation, June 26, 2002

[6] World Wide Web Consortium, "Natural Language Semantics Markup Language (NLSML) for the Speech Interface Framework", W3C Working Draft, May 2001.

[7] World Wide Web Consortium, "Voice Extensible Markup Language (VoiceXML) Version 2.0, W3C Candidate Recommendation, February 20, 2003

[8] T. Melanchuk, "Media Sessions Markup Language (MSML)", Internet Draft, Internet Engineering Task Force, Feb. 2004. Work in progress.

[9] J. Van Dyke, E. Burger, A. Spitzer, "Basic Network Media Services with SIP", Internet Draft, Internet Engineering Task Force, March 2003. Work in progress.

[10] C. Jennings, SIP Support for Application Initiation, Internet Draft, Internet Engineering Taskforce, Oct. 2002. Work in progress.

[11] A. B. Roach, Session Initiation Protocol (SIP)-Specific Event Notification, RFC 3265, Internet Engineering Taskforce, June 2002.

[12] E. Levinson, "Content-ID and Message-ID Uniform Resource Locators", RFC 2392, Internet Engineering Taskforce, August 1998.

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

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

Copyright Statement

Acknowledgement