FS – Execution Engine

Content

History

Version	Date	Adjustments
A	2006-10-05	First revision (ERMKESE)
B	2006-12-05	Minor changes. Updates about join. Describing sending of service requests. (ERMKESE)
C	2007-06-11	Added ASR and TTS. Added variables related to number completion. Informed what happens if incoming XMP requests are not responded to within validityTime. Updated with info that the application can analyse why a <createcall> failed by checking the network status code (ermkese).
D	2008-08-05	Updated with outboundcallserver. Added VCP Resource Locating (4.5.15) (ekensel/EJEFRID).

1 Terminology

Application	A logical flow of events described in VoiceXML, CCXML and ECMAScript.
CCXML session	A CCXML session is comprised of an executing CCXML document, or sequence of CCXML documents; each concurrently executing CCXML document is a separate session, and can be uniquely identified and referenced.
Session	A session is equivalent to a CCXML session.
Services	Services in this document is the same services that are registered in MCR. I.e. the service is at least used by some other components in the M3 system. It is the Application only that defines these services.

2 Introduction

This document specifies the functions provided by the Execution Engine component.

The main purpose of the Execution Engine is to compile and execute Applications. In order to do so, it provides a set of additional functions such as execution of ECMAScripts, Session management and Application Management.

3 Function Requirements (Commercial)

3.1 Execution of applications implemented in VoiceXML 2.1 and CCXML 1.0

An application implemented in VoiceXML/CCXML and conforming to the VoiceXML /CCXML specifications, see [1] and [2] for details, can be executed by the Execution Engine. Interfaces for compilation and execution of such an application is made available by the Execution Engine, see 4.1.3 and 4.1.4 for details.

The supported version of CCXML 1.0 is the draft 2004-04-30.

Only a subset of CCXML and VoiceXML is supported. See also 4.5 on page 22.

4 Function Requirements (Design Related)

4.1 Exported Interfaces

The exports the following interfaces:

- PlatformAccess/PlatformAccessUtil
- IApplicationExecution
- IApplicationManagement
- IApplicationCompiler
- ISession
- IComponentManager
- ServiceEnabler

4.1.1 PlatformAccess and PlatformAccessUtil

EE exports objects of class PlatformAccess and PlatformAccessUtil to be used in ECMA script in the CCXML and VoiceXML files. The object exist automatically and are called "mas" and "util".

Logically, the application can treat each object as there is a single object per session; manipulations done in CCXML are visible in VoiceXML and in VoiceXML subdialogs, for example.

The classes have functions belonging to different categories, for example mailbox related and profile related. See the javadoc for all details.

4.1.2 IApplicationExecution

The IApplicationExecution interface provides functionality for execution of applications implemented in VoiceXML/CCXML.

Each user session has its own instance of the IApplicationExecution interface, i.e. the control of the execution of a specific application is related to a specific user session.

4.1.2.1 Functions

4.1.2.1.1 start()

It is possible to start Applications developed in VoiceXML and/or CCXML, compiled by the VoiceXML/CCXML compiler. A newly created instance of ISession is returned.

An application is automatically terminated when its stops executing according to its own language (e.g. CCXML) semantics.

4.1.2.1.2 setSession(ISession)

This function sets the ISession object related to this session. This function is preferably invoked before start().

4.1.2.1.3 terminate()

This method terminates the session. This is equivalent to execute CCXML <exit>, that is, all active calls will be disconnected automatically by EE.

4.1.2.1.4 getSession()

Return the ISession related to the IApplicationExecution instance.

4.1.3 IApplicationManagement

The IApplicationManagement interface presents functions for loading an application corresponding to a service name.

4.1.3.1 Functions

4.1.3.1.1 load(service)

Locates an application related to the specified service. Returns a newly create instance of IApplicationExecution.

The relations between service names and applications are expressed in a configuration file.

4.1.4 IApplicationCompiler

The IApplicationCompiler interface provides functions for compiling documents and applications implemented in VoiceXML/CCXML into a format understood by the IApplicationExecution interface.

4.1.4.1 Functions

4.1.4.1.1 compileApplication(applicationURI)

It is possible to compile an application defined in VoiceXML/CCXML into an executable format, understood by the application execution mechanism. The compiled result will by default be stored in-memory.

The applicationURI points to a text file defining all VoiceXML and CCXML documents part of the application. Also, one of the CCXML documents is defined as starting document for executing the application.

4.1.4.1.2 compileDocument(documentURI)

It is possible to do on-the-fly compilation of dynamically generated VoiceXML documents, by specifying its URI.

The documentURI is an URI pointing to the VoiceXML or CCXML file to be compiled.

4.1.5 ISession

The ISession interface represents a user session where any user-related information is stored. It is possible to get and set name/value pairs which may be used from within the application.

One ISession instance is created per session.

4.1.5.1 Functions

4.1.5.1.1 getId()

Retrieve the globally unique ID of the current session.

4.1.5.1.2 setData(name, value)
getData(name)

A set of methods through which it is possible to add, update or retrieve data in the session using a name/value mechanism. If a value exists in the Session for the specified name, the value will be overwritten by the new value.

4.1.5.1.3 *setSessionLogData(name, value)*

Register session data to be propagated to the Mapped Diagnostic Context. The data set in this call enables tracing of a session based on the value of the data set in the call.

4.1.6 IComponentManager

The component manager is acting as an object factory, creating instances of requested components. The manager reads a configuration at start-up, where a binding between a logical name and an actual implementation is made.

A specific component may be registered in the configuration as being singleton or template, singleton meaning that the same instance of the requested component will be re-used for each request, template meaning that a new instance will be created for each request.

The interfaces that can be created and returned by the IComponentManager must contain a default constructor (i.e a constructor that accepts no parameters) and get/set methods for all public members.

The IComponentManager interface is implemented as a singleton.

4.1.6.1 Functions

4.1.6.1.1 create(componentSpecification)

The specified component is created. Depending on the configuration read by the component manager, the returned component will be a singleton or a template.

4.1.7 ServiceEnabler

The ServiceEnabler is an interface that components who enables services should implement. CallManager could be an example of such a component. EE can be configured regarding what service enablers there are, and what application the respective service enabler is related to. At startup, EE initializes the service enablers, which includes passing some data found in configuration to the service enabler, for example TCP/IP port to use. At initialization, the service enabler must return a ServiceEnablerOperate, which EE can give to OperateAndMaintainManager for supervision of the service enabler.

4.1.7.1 Functions

4.1.7.1.1 initService(service, host, port)

This function is implemented by the service enabler and invoked by EE. The service enabler is supposed to initialize its service according to the supplied parameters.

4.2 Imported Interfaces

The EE imports the following interfaces:

- ICallManager. This interface is used for handling in- and outbound calls.
- IConfiguration. This interface is used for accessing configuration parameters.
- IEventDispatcher. This interface is used for sending/receiving events to/from other components.
- ILog. This interface is used for logging, for example debug and error messages.
- IMediaContentManager. This interface is used for accessing installed media packages.
- IMediaObject. This interface is used for playing and recording media.
- IStream. This interface is used for speech recognition.
- ExternalComponentRegister. This interface is used for looking up media translators.
- IMailbox. This interface is used for storing and retrieving mails.
- MediaTranslationManager. This interface is used for speech recognition and text to speech.
- MessageSender. This interface is used for sending mails.
- NumberAnalyzer. This interface is used for number analysis.
- ProfileManager. This interface is used for looking up and modifying user profiles.
- ServiceRequestManager. This interface is used for sending and receving service requests.
- TrafficEventSender. This interface is used for sending traffic events.
- OperateAndMaintain. This interface is used for updating call status information about each session.

4.3 Events

4.3.1.1 Consumed

The Execution Engine consumes the events sent by CallManager, Stream and MediaTranslationManager. The events are the following:

- ConnectedEvent
- DisconnectedEvent
- FailedEvent
- PlayFinishedEvent
- PlayFailedEvent

- RecordFailedEvent
- RecordFinishedEvent
- ErrorEvent
- AlertingEvent
- ProgressingEvent
- JoinedEvent
- UnjoinedEvent
- EarlyMediaAvailableEvent
- EarlyMediaFailedEvent
- NotAllowedEvent
- JoinErrorEvent
- UnjoinErrorEvent

4.3.1.2 Produced

- ApplicationEnded. This event is produced when execution of the application has ended.

4.4 Function specification

4.4.1 Third-party products

The EE must be designed such that used third-party products are possible to replace easily.

This could be solved by encapsulating the third-party product, or make sure that it is used in a localized manner, i.e. not spread out everywhere in the code.

4.4.2 Defining an application

A VoiceXML/CCXML application is defined through a set of VoiceXML/CCXML documents. The set is defined in a configuration file. Each document will be compiled and inserted into the repository of the Application Manager.

One of the CCXML documents is appointed as starting point for the application.

4.4.3 Cleaning up back-end components

In order to cleanup resources held by backend components, EE calls IProfile.close() on every used IProfile object. EE will do this at the following situation:

- After all dialogs and the CCXML application has finished executing according to their corresponding language semantics, for example that <exit> has been executed by the CCXML application.

If this situation does not occur, resources will not be cleaned up, and objects related to the session will remain.

4.4.4 Configuration

EE uses ConfigurationManager for accessing its configuration. EE logs an error message if ConfigurationManager reports a problem with parameter(s).

Information possible to find automatically in the execution environment are indeed found automatically (as opposed to being read from a configuration file).

EE always uses the most recent configuration, as returned by ConfigurationManager.

4.4.5 Information Logging

EE logs the following as information logs:

- Executed tag in the current CCXML or VoiceXML document, including where this tag is located (line number and document name).
- When the execution transitions to another document
- Variable values at assignment and evaluation.
- When a grammar is matched, it is logged what grammar that did match including which rule, and also the value of the resulting utterance.
- All events EE deals with, including events thrown by other components.

4.4.6 Events sent to EE

Execution Engine must be able to handle events received from other components in the same order as they were received[1].

4.4.7 Connection Monitoring

At start of a new Session, EE creates an instance of ConnectionInfo (interface published by OperateAndMaintain manager).

EE will update the ConnectionInfo at the following occasions:

- Start of a session.
- Incoming event from CM, e.g. Alerting. These events are used for setting the call state and the ANI/DNIS etc.
- Actions taken on behalf of the application. These actions are exactly: start of play, start of record, play finished, record finished, play failed.

4.4.8 Starting up a program

EE contains a "main" function and will start up a program (for example MAS), by initiating other components according to a configuration file. EE is not aware of which components/classes this involves, and only acts according to what if finds in the configuration file. The created components/classes are then available by querying the ComponentManager interface.

[1] In other words, EE shall not switch the order of the events due to a faulty thread implementation.

EE creates an instance of the CallManager interface, according to the configuration file.

4.4.9 Error handling

4.4.9.1 No timely response from CallManager

EE must detect[2] the fact that CallManager never sends the event. How long to wait before considering a missing event as an error is configurable. The EE shall deliver an error.connection event to the CCXML application and disconnect the corresponding connection.

4.4.9.2 Incoming calls that stay forever in the ALERTING state

The Execution Engine must be able to detect[3] that the Application did not take an appropriate action on an incoming call in the ALERTING state within a configurable time, and deliver an error.connection event to the application. Also, EE must tell CallManager to disconnect the call.

4.4.9.3 Outgoing calls that CM never sends events about

This requirement applies if the application has specified a timeout[4] for its <createcall> invocation.

If EE executes <createcall> but CM does not inform EE about the outcome via an event within (X + "timeout"), EE will send error.connection to the CCXML application.

X is configurable.

4.4.9.4 Incomplete application

If the Execution Engine can not find all contents of an application, for example since a VoiceXML file is not present, this will be logged, and invoking the missing parts of the application results in error.badfetch for VoiceXML and error.fetch for CCXML.

4.4.10 Application Event interface

EE may deliver Application Events to the Application. Some events and errors are specified in the VoiceXML specification, [1] others are specific to the functionality provided by the platform. The later are provided via PlatformAccess.

The event is local to one user session, i.e. it will only affect one session.

---

[2] A number of calls to CallManager are asynchronous, in the sense that CallManager will respond with a corresponding "finished" or "failed" event when the operation is finished

[3] It is possible to write an application that does not handle an incoming call which has reached the ALERTING state. In this case, the CCXML application will have received the CCXML event "connection.alerting" but not taken appropriate action, for example by answering the call using <accept>.

[4] The <createcall> element has an attribute called "timeout".

4.4.10.1 PlatformAccess events

When the application invokes PlatformAccess, the platform may report a number of errors to the Application. The errors are not part of the VoiceXML specification, but are possible to catch within the Application. See the PlatformAccess javadoc for details.

4.4.11 Implicit Application Variables

The Execution Engine component sets some ECMAScript variables in the Application.

4.4.11.1 Clarification of VoiceXML standard variables.

The ”pi” member of session.connection.redirect[x] can have values 1 (secret number) or 0 (non-secret number).

The reason member of session.connection.redirect[x] is mapped from causes reported by Call Manager as follows:

UNKNOWN → ”unknown”

USER_BUSY → "user busy"

NO_ANSWER → "no reply"

UNAVAILABLE → "mobile subscriber not reachable"

UNCONDITIONAL → "unknown"

TIME_OF_DAY → "unknown"

DO_NOT_DISTURB → "unknown"

DEFLECTION → "deflection immediate response"

FOLLOW_ME → ”unknown”

OUT_OF_SERVICE → ”unknown”

AWAY → ”unknown”

4.4.12 Non-standard VoiceXML session variables

Variables called "number" here have their format according to E.164.

Variables called "user" here have their format according to the userinfo part (see RFC 3261) of a SIP or SIPS URI.

Name	Description
session.connection.calltype	Indicates to the application what type of call it is. Possible values: • voice. Indicates that the call is a voice call. • video. Indicates that the call is a video call. The variable may be undefined if the call type is not known. If there was a call, and hence the variable got the value "voice" or "video", the variable will keep this value event after the call has been hung up.
session.connection._totalinboundbitrate	The total inbound bit rate (for all media types, for example voice+video if it is a call with both voice and video media) for the inbound stream, in bits/second.
session.connection.local.number	This variable defines the B-number (called number) for an incoming call. The value is defined both if the VoiceXML dialog is started in CCXML call states alerting and connected. The telephone number is presented according to the format specified in E.164.
session.connection.remote.number	This variable defines the A-number (calling number) for an incoming call. The value is defined both if the VoiceXML dialog is started in CCXML call states alerting and connected.
session.connection.redirect[].number	This variable defines the C-number (redirecting number) for an incoming call. The value is defined

	both if the VoiceXML dialog is started in CCXML call states alerting and connected.
session.connection.local.user	This variable defines the B-user (called user) for an incoming call. The value is defined both if the VoiceXML dialog is started in CCXML call states alerting and connected.
session.connection.remote.user	This variable defines the A-user (calling user) for an incoming call. The value is defined both if the VoiceXML dialog is started in CCXML call states alerting and connected.
session.connection.redirect[].user	This variable defines the C-user (redirecting user) for an incoming call. The value is defined both if the VoiceXML dialog is started in CCXML call states alerting and connected.
session.connection.redirect[0].reason	This variable defines the reason for the call being redirected. In addition to the values defined by ref. [3] , this variable may also have the value "unconditional", which means that all calls are redirected.
session.connection.redirect[0].pi	This variable defines the presentation indicator for the C-user (redirecting user) for an incoming call.
session.connection.remote.pi	This variable defines the presentation indicator for the A-user (calling user) for an incoming call.
session.connection.remote._numbercomplete	This variable defines the number completion for the A-user (calling user) for an incoming call (remember that the A-number is defined via the session.connection.remote.number variable). If true, the number is

	complete. If false, the number is incomplete. If the value is undefined, it is unknown whether the number is complete or not.

4.4.13 Non-standard CCXML variables

Name	Description
_calltype	Indicates to the application what type of call it is. Possible values: • voice. Indicates that the call is a voice call. • video. Indicates that the call is a video call. The variable may be undefined if the call type is not known. As the _calltype property exists on the connection object, it may for example be access using the syntax evt. connection._calltype. If there was a call, and hence the variable got the value "voice" or "video", the variable will keep this value event after the call has been hung up.
_earlymedia	Member of the connection.progressing event. Is true if the call contains early media. Accessed using the syntax evt.info._earlymedia.
_numbercomplete	Member of the connection.alerting event. See the ._numbercomplete variable in section 4.4.12 for description. Accessed using the syntax evt.connection.remote._numbercomplete.
hints	This is an ECMA structure with the following members: • "pi" (for presentation indicator) • "calltype" (for defining the call type of the outbound call) • "outboundcallserverhost" (for defining the host name of the

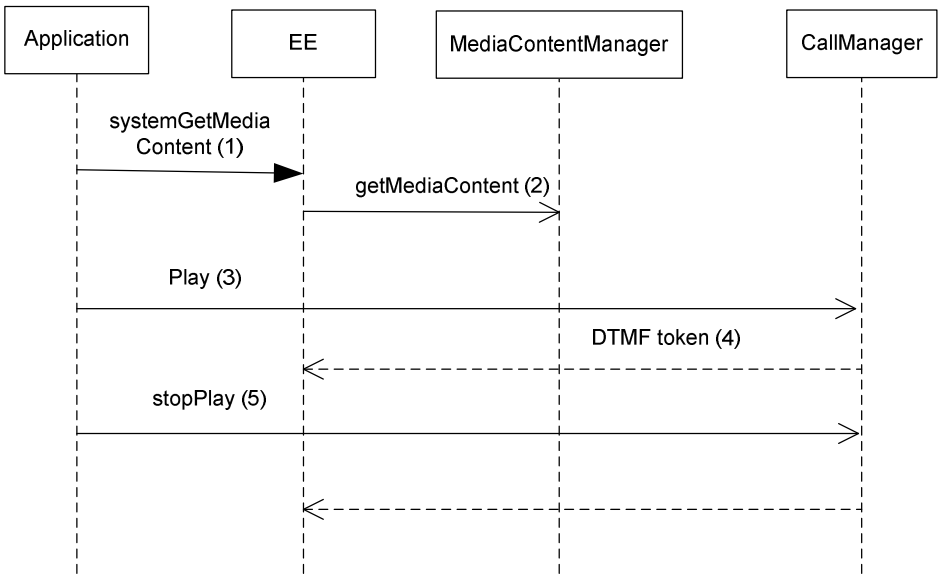
	server to contact when setting up the outbound call. May be either a host name or an IP address.) • "outboundcallserverport" (for defining the port of the server to contact when setting up the outbound call. If the outboundcallserverport is not defined, but outboundcallserverhost is, a default port will be used.)

4.4.14 User interaction through voice or video and DTMF

This diagram shows an example when the application plays a media object from a media content package, and gets interrupted by DTMF from the end user. The DTMF matches a grammar, and EE stops the playing of the media object.

The example VoiceXML code looks like this:

```
<block>
    <var name="ljud"/>
    <script>
        mas.systemSetMediaResources ("en", "male", null);
        ljud = mas.systemGetMediaContent ("prompt", "1", null);
    </script>
    <prompt>
    <audio expr="ljud"/>
    </prompt>
    <exit/>
</block>
```



1. The application invokes systemGetmediaContent on EE.
2. EE asks for a media object array corresponding to the parameters supplied from the application. The media object array is stored in the ECMA variable "ljud".
3. Upon executing the <audio> tag, EE queues the prompt "ljud". When entering VoiceXML waiting state, EE plays the prompt "ljud" by invoking play on CallManager.
4. EE gets notified that a DTMF key has been entered.
5. EE and finds out that the DTMF matches a grammar. EE asks Call to stop playing the media object, and EE waits for playFinished event.
6. EE receives the playFinished and continues to execute the application.

4.4.15 Start Application due to inbound call

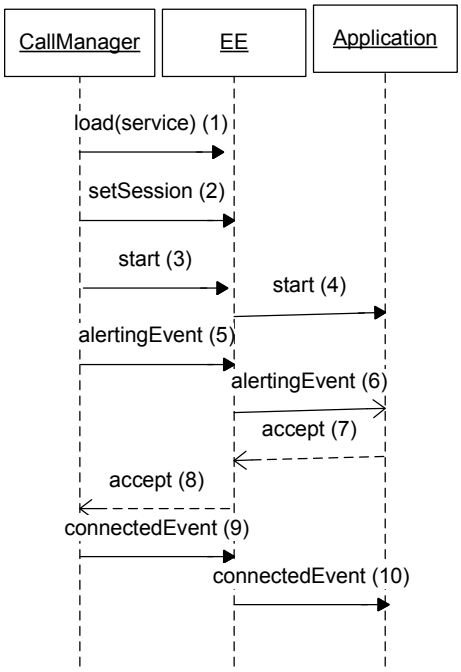


Figure 1 Start Application, Inbound call

1. The Call Manager loads the application using the IApplicationManagement interface. A new instance of the IApplicationExecution interface is created for executing the loaded application.
2. The CallManager sets the session into the IApplicationExecution instance.
3. The CallManager starts the IApplicationExecution.
4. The EE starts the application. This allows the application to initialize variables for example.
5. The CallManager sends an AlertingEvent to EE.
6. EE delivers the alertingEvent to the application.
7. At some later point, the application decides it wants to accept the call.
8. EE calls accept on CallManager.
9. The Callmanager generates a connectedEvent.
10. EE delivers the connectedEvent to the application.

4.4.16 Start Application, outbound call due to notification

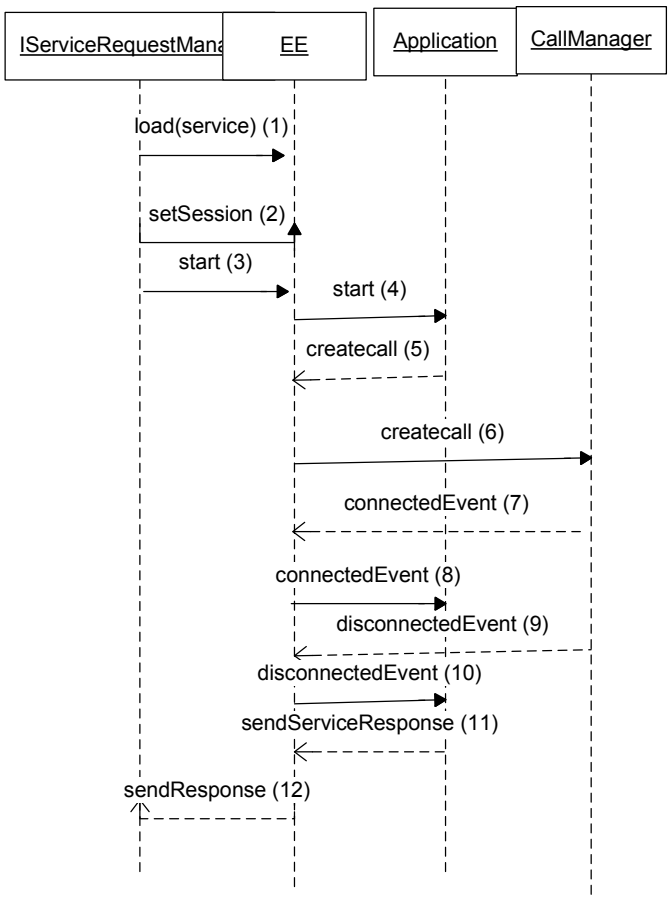


Figure 2 Outbound call, Notification

1. The IServiceRequestManager loads an application based on a requested service in order to make an outdial notification. A new IApplicationExecution instance for the application is created. IServiceRequestManager set the serviceRequest type in ISession.
2. The IServiceRequestManager sets the session into the IApplicationExecution instance.
3. The IServiceRequestManager starts the IApplicationExecution.
4. The EE starts the application. This allows the application to initialize variables for example.
5. The application asks EE to create an outbound call.
6. EE asks CallManager to create an outbound call.
7. When the call is connected, CallManager delivers a connectedEvent to EE.
8. EE delivers a connectedEvent to the application.
9. When the call gets disconnected, CallManager delivers a disconnectedEvent to EE.
10. EE delivers a disconnectedEvent to the application.
11. The application instructs EE to send a service response.

12. EE instructs IServiceRequestManager to send a service response.

4.4.17 Hang up a call

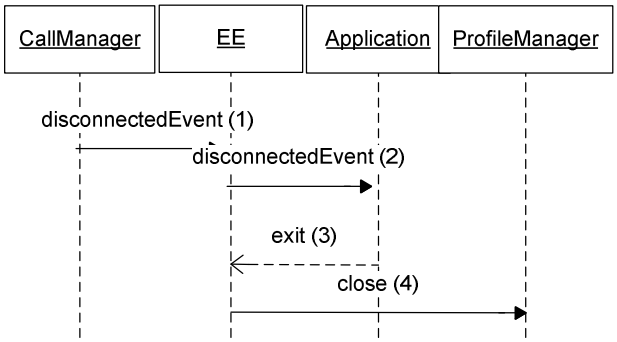


Figure 3 Hang up a call

1. The CallManager issues a disconnected event to the application since the user hung up.
2. EE sends a disconnected event to the application. The application application performs any clean up routines related to call hang-up.
3. The application exits.
4. EE performs clean up routines, for example telling ProfileManager to close all resources that were related to this session.

4.5 Support of standards

This section describes the EE support for VoiceXML/CCXML in cases where we have clarified, narrowed or extended the functionality in the standard.

See also ref. [20] [21] and [22]

4.5.1 Dialogs started from CCXML

It is only supported to run one dialog at a time in a CCXML session. That is, if a dialog has been started using <dialogstart>, you can not perform another <dialogstart> until the first dialog has exited.

4.5.2 Using media resources

To be able to play media resources from a prompt package, the application must invoke PlatformAccess.systemSetMediaResource or PlatformAccess.systemSetEarlyMediaResource.

PlatformAccess.systemSetEarlyMediaResource shall be used when you want to play early media and has these restrictions:

- It must be invoked from CCXML after an alerting event

- It must only be invoked once per inbound call[5]
- The application can not invoke systemSetMediaResource, use the <accept> tag or play media until the application has received the event "earlymediaresourceavailable".

systemSetMediaResource is used when you do not want to play early media, and can be used any number of times. Restrictions:

For the first call in the session, it must be invoked before <accept> for an inbound call, and before <createcall> for an outbound call.

If the application does not use the method above for selecting media resources, the platform will select default resources.

4.5.2.1 Playing Early media to an inbound call

At the alerting event in CCXML, you must call mas.systemSetEarlyMediaResource, and then wait for the platform specific event "com.mobeon.platform.earlymediaresourceavailable". When this event is received, it is possible to play early media, and a dialog doing so may be started. When that dialog has finished playing early media, accept the call and invoke the "real" application.

There are restrictions on what you can do in the early media application. Using form input items would result in undefined behavior.

Below is a sketch of what the CCXML program could look like:

```
<!-- On receiving the alerting event, start set media resources→
<transition event=connection.alerting>
    <script>mas.systemSetEarlyMediaResource(connectionID, 'en', 'voice1',
'video1');</script>
</transition>

<!-- On receiving the earlymediaresourceavailable event, start the application for
early media→
<transition event="com.mobeon.platform.earlymediaresourceavailable">
    <dialogstart src="'earlymedia.vxml'">
</transition>

<!-- On receiving the earlymediaresourcefailed event, choose between accepting the
call and start the real application,or rejecting the call →
<transition event="com.mobeon.platform.earlymediaresourcefailed">
-----
</transition>


<!-- Assume that the early media application exits. Then you accept the call. →
<transition event="dialog.exit">
    <accept/>
</transition>

<!—When the call is connected, start the real application.-->
<transition event="connection.connected">
    <dialogstart src="'therealapplication.vxml'">
</transition>
```

4.5.2.2 Playing media to an inbound call after it is connected

Invoke mas.systemSetMediaResource before <accept>.

Below is a sketch of what the CCXML program could look like:

[5] Usually there is just one inbound call ;)

```
<!-- On receiving the alerting event, start set media resources and accept the call→
<transition event=connection.alerting>
    <script>mas.systemSetMediaResource('en', 'voice1', 'video1');</script>
    <accept/>
</transition>

<!—When the call is connected, start the application.-->
<transition event="connection.connected">
    <dialogstart src="'therealapplication.vxml'">
</transition>
```

4.5.2.3 Playing media to an outbound call (non-transfer)

This is the case when you make a call without the purpose of joining it with another call.

```
<eventprocessor statevariable="state0">

<!—Make a call →

<transition event="ccxml.loaded" name="evt">
    <script>mas.systemSetMediaResource('en', 'voice1', 'video1');</script>
    <createcall dest="112"/>
</transition>

<!—When the call is connected, start the application.-->
<transition event="connection.connected">
    <dialogstart src="'therealapplication.vxml'">
</transition>



</eventprocessor>
```

4.5.2.4 Receiving media from an outbound call during transfer

This section covers both the early media and non-early media scenarios.

This works as in the VoiceXML specifications, with these clarifications.

To have the ring tone of the outbound call played to the original call, the calls can be joined at the progressing stage, otherwise they must be joined at the connected state.

Note: there is no need to invoke PlatformAccess.systemSetMediaResource in this scenario, since it has already been invoked for the first call.

The application should use a state variable to keep track of when it needs to join, according to this example code:

```
<eventprocessor statevariable="state0">

<!—Make a call →

<transition event="dialog.transfer" name="evt">
    <createcall dest="112"/>
</transition>

<!-- On receiving the progressing event, examine if it has early media: →

<transition event="connection.progressing" name="evt">
    <if cond="evt.info._earlymedia == true">

        ← We assume here that C1 and C2 are the network connections
           Corresponding to the inbound and outbound calls →
        <assign name="state0" expr="'joiningEarlyMedia'" />
        <join id1="C1" id2="C2" duplex="'full'">
    </if>
</transition>
```

```
<transition state="'init'" event="connection.connected" name="evt">

    <!—call connected in the init state (there was no early media), then we must join
→
        <join id1="C1" id2="C2" duplex="'full'">
</transition>

<transition state="'joiningEarlyMedia'" event="connection.connected" name="evt">

    <!—call connected in the joiningEarlyMedia state, then we do not have to join
(calls are already joined) →
</transition>


</eventprocessor>
```

4.5.3 <join> element

A <join> between two connection identifiers C1 and C2 in CCXML is only supported in these cases:

- The inbound call C1 is in connected state and the outbound call C2 is in connected state.
- The inbound call C1 is in connected state and the outbound call C2 is in progressing state and the outbound call indicates early media.
- C1 is an inbound call and C2 is a dialog.
- C1 and C2 are both outbound calls in the connected state.

Calls can only be joined full-duplex. Calls can only be joined if they are not joined already[6]. Trying to join two calls where one or both are already joined, will result in an "error.conference.joined" event.

As soon as the calls are joined, the media specified with "transferaudio" attribute will stop playing and the inbound media provided by the network will be played instead (if the network indicates it has inbound media).

Important[7]: if the application first joins two connections and, before an event telling the outcome of the join (for example conference.joined) performs another request on one of the joined calls, the order of events sent to the application can be intermixed.

For example, if the application first joins (C1, C2) and then the call is immediately disconnected (either by the application or the network), **one** of the following outcomes are possible:

- First "conference.joined" then "connection.disconnected"
- First "connection.disconnected" then "conference.joined".
- First "connection.disconnected" then "error.conference.join"

The application should handle the intermix of events by using a state variable in its "eventprocessor" element and ignoring the second event in case "connection.disconnected" is sent first.

[6]This behavior is guaranteed by CallManager.
[7]This behavior is implemented by CallManager.

4.5.4 <script> element

If any of the statements in an ECMA script tag generates an event (for example error.semantic or a PlatformAccess event), the script tag will still be run to completion before the event takes effect. One way to design the application to handle this, is to have the PlatformAccess call as the last statement in the script tag.

4.5.5 <createcall> element

4.5.5.1 Call type

Since it may be impossible to make a video call but possible to make a voice call, it is recommended that the application behaves as follows: try making a call of the same type as the current call type, and if this fails and the call type is video, try making a voice call.

If the createcall invocation results in in a connection.failed event, the application may analyse the reason why createcall failed, by examining the "reason" field of the connection.failed event. The reason field contains the network status code, as defined in ref. [24] .

The presentation indicator and A number may be defined as below (assuming that the createcall is initiated due to a dialog.transfer event from VoiceXML).

```
<!-- On receiving the dialog.transfer event, try making a call with the same call type
as the existing call →

<transition event="dialog.transfer" name="evt">
    <var name="hints"></var>
    <script>
        hints= {pi : evt.values.transfer_local_pi,
                 calltype : in_calltype}
    </script>
     <createcall dest="evt.URI"
                       callerid="evt.values.transfer_ani"
                       connectionid="out_connectionid"
                       timeout="evt.connecttimeout"
                       hints="hints"/>
</transition>

<!—if we failed to make a call, we may retry a voice call. Note that some details
regarding e.g. callerid is omitted. Also, a real application would check the reason
why the first code failed, before making another attempt →

<transition event="connection.failed" name="evt">

    <if cond="in_calltype == 'video'">
        <var name="hints"></var>
        <script>
            hints= {calltype : 'voice' }
        </script>
         <createcall dest="evt.URI"
                           hints="hints"/>
        </if>
</transition>
```

4.5.5.2 Outbound Call Server

The application is able to define what outbound call server to be contacted when setting up the call on behalf of the <createcall> element. The server is defined in the "hints" attribute of <createcall>.

The server may be defined as below (assuming that the createcall is initiated due to a dialog.transfer event from VoiceXML).

```
<!-- On receiving the dialog.transfer event, try making a call with the same call type
as the existing call →

<transition event=”dialog.transfer” name=”evt”>
            <var name="hints"/>
            <var name="out_connectionid"/>

            <script>
                hints= {outboundcallserverhost : evt.values.outboundcallserverhost,
                        outboundcallserverport : evt.values.outboundcallserverport}
            </script>
            <createcall dest="'222'"
                        callerid="'121212'"
                        connectionid="out_connectionid"
                        hints="hints"/>
 </transition>
```

4.5.6 <transfer> element

The application is able to pass transfer-related information from the VoiceXML to the CCXML. There are two ways do do this:

- Via predefined properties
- Via the extensible property com.mobeon.platform.transfer_properties.

The following table lists the predefined properties:

The application is able to define a number of properties, which will be sent to CCXML via the dialog.transfer event. The table below lists the properties and under what name they are transferred in the dialog.transfer event.

Name of property in VoiceXML:	**Transferred in the dialog.transfer event sent to CCXML as:**
com.mobeon.platform.transfer_maxtime	maxtime. Unit: Time Designation.
com.mobeon.platform.transfer_connecttimeout	connecttimeout. Unit: Time Designation.
com.mobeon.platform.transfer_ani	values.transfer_ani
com.mobeon.platform.transfer_local_pi	values.transfer_local_pi

The extensible property com.mobeon.platform.transfer_properties is used to pass arbitrary information to CCXML. The value of com.mobeon.platform.transfer_properties is an ECMA variable, which has all information to pass as sub-properties. The dialog.transfer event will be given the same sub-properties and corresponding values, in the dialog.transfer.

The sub-properties will be put on the “values” sub-property of the dialog.transfer event.

The following example will clarify.

```
<!—The following example shows how an application can pass an IP-address of a server
and the calltype to CCXML. Note that the application could have passed any kind of
information, this is up to the application to decide. →

<form>
```

```
       <var name="tprop"/>  <!—Define a variable to hold the sub-properties →

      ← Set sub-properties “server” and “calltype” on tprop. Thn point out “tprop” as
the name of the variable containing sub-properties to be passed in the dialog.transfer
event →
      <script>
          tprop= {server : "108.109.11.12",
                  calltype : "video"};
          mas.systemSetProperty('com.mobeon.platform.transfer_properties', 'tprop');
      </script>

      <!—Execute a <transfer> →

     <transfer name="outboundcall" bridge="true" destexpr="destinationNumber"
transferaudio="beep.wav">
         <grammar version="1.0" mode="dtmf" root="tc1_main_rule">
            <rule id="tc1_main_rule" scope="public">
               <item>*</item>
            </rule>
         </grammar>
         <filled>
             <log>TRANSFER DONE</log>
         </filled>
      </transfer>
   </form>
```

The “server” and “calltype” are accessed in CCXML according to the example below.

```
<transition event=”dialog.transfer” name=”evt”>
    <log expr=”
    <log expr="'SERVER:'+evt.values.server"></log>
    <log expr="'CALLTYPE:'+evt.values.calltype "></log>

    <!—Here, a createcall using the server and calltype somehow, would take place→
</transition>
```

4.5.7 URI

EE only supports the URI schemes “file” and “http”.

4.5.8 <log> element

The “label” attribute of the log level can be used to set a severity of the log. The possible severities are: debug, info, warn, fatal and error. For example, if “error” is the value of the label, the corresponding log will be logged as an error log.

If an unknown label is used, the log comes out as an info log with the label attached before the log message. If no label is used, the log comes out as an info log.

The “expr” attribute of the log element is evaluated and put before the contents of the log element.

4.5.9 <disconnect> in CCXML

If the application executes <disconnect> at about the same time as the call is hung up from the network side, there may be two connection.disconnected events generated. This is something the application must be aware of.

4.5.10 Sending service requests to an XMP server

By using PlatformAccess, it is possible to send service requests to an XMP server, and retrieve information from the corresponding response. A response will always be available after sending a request, even if a response was never received from the XMP server. In this case, the response will have been internally created, and by examining the "StatusCode", the application is able to see that there was a communication problem.

PlatformAccess has a method for sending the request, and this method will block until there is a response (either received from the XMP server or internally generated). At this point, also a PlatformAccess event may be generated, so the application should send the request as the last operation within one ECMA script expression (for example a script tag) and then verify the result in a second ECMA script expression, in order to allow events sent from the request-sending method in PlatformAccess to take effect before the second ECMA script expression is executed.

See the example below.

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


    <catch event="error">
        <log>caught an error event</log>
        <exit></exit>
    </catch>
    <form id="form2">
        <var name="paramValue"></var>
        <var name="TransactionId"></var>
        <var name="ClientId"></var>
        <var name="StatusCode"></var>
        <var name="StatusText"></var>
        <block>
            <script>
                var parameterNames = ["param1"];
                var parameterValues = ["value1"];
                mas.systemSendServiceRequest( "anita",
                                              "someService",
                                              10,
                                              true,
                                              parameterNames,
                                              parameterValues);
            </script>
            <script>
                paramValue = mas.systemGetServiceResponseParameter( "paramName");
                TransactionId = mas.systemGetServiceResponseHeaderParameter(
                                    "transactionid");
                ClientId = mas.systemGetServiceResponseHeaderParameter( "clientid");
```

```
                StatusCode = mas.systemGetServiceResponseHeaderParameter(
                                        "statuscode");
                StatusText = mas.systemGetServiceResponseHeaderParameter(
                                         "statustext");
            </script>

            <log>paramName:<value expr="paramName"></value></log>
            <log>TransactionId:<value expr="TransactionId"></value></log>
            <log>ClientId:<value expr="ClientId"></value></log>
            <log>StatusCode:<value expr="StatusCode"></value></log>
            <log>StatusText:<value expr="StatusText"></value></log>
            <exit></exit>
        </block>
    </form>
</vxml>
```

4.5.11 Receiving XMP requests from an XMP client

By using PlatformAccess, it is possible to receive service requests from an XMP client, and send a corresponding response.

If the application does not send a response before the XMP parameter "validityTime" of the XMP request has elapsed, the platform will automatically send a response to the XMP client, and try to terminate the application by delivering a "ccml.kill" event to the CCXML document. The application should either:

1. Catch the ccxml.kill event, do any cleanup the application considers necessary, and then exit.
2. Not catch the ccxml.kill event, in which case the application will be automatically terminated by the platform.

Here is an example application which retrieves a parameter from the incoming XMP request , makes an outbound call, and sends an XMP response when the outbound call gets connected:

```
<?xml version="1.0" encoding ="iso-8859-1"?>
<ccxml xmlns="http://www.w3.org/2002/09/ccxml" version="1.0">

    <var name="out_connectionid"></var>
    <var name="serviceVar"></var>

    <eventprocessor>
        <transition event="ccxml.loaded" name="evt">
            <log expr="'CCXML loaded'"></log>
            <script>
                serviceVar = mas.systemGetServiceRequestParameter('kalle');
            </script>
            <if cond="serviceVar != 'kalle_value'">
                <log expr="'ERROR'"></log>
```

```
            </if>
            <createcall dest="'1234'"
                        callerid="'5678'"
                        connectionid="out_connectionid"></createcall>

        </transition>

         <transition event="connection.connected" name="evt">
             <log expr="'CCXML connected'"></log>
             <script>
                 mas.systemSendServiceResponse(12, 'theStatusText', null, null);
             </script>
             <exit></exit>
        </transition>

        <transition event="dialog.exit" name="dlg">
                <exit/>
        </transition>

        <transition event="error.*" name="evt ">
                <log expr="'TCFAIL CCXML: Error = ' + evt.name + ', Reason =
'+evt.reason"/>
                <exit/>
        </transition>
    </eventprocessor>
</ccxml>
```

4.5.12 Automatic Speech Recognition

The application is able to use automatic speech recognition (ASR) by using speech recognition grammars. Only XML Form is supported. ASR is only available for the <field> element, and not for the <record> and <transfer> element (hence it is not possible to have recording or transfer interrupted by user speech).

The recognition will be performed in the language defined in the grammar, using the “xml:lang” attribute.

What elements the application can use when defining the speech grammar is depending on the grammar support in the ASR Engine. The grammar defined by the application will be sent as is to the ASR Engine.

Semantic interpretations are not supported. Matching utterances are assigned to application.lastresult$.utterance and name$.utterance (assuming “name” is the name of the input item variable).

Bargein using speech is not supported. Recognition will start after all prompts have been played.

It is possible to activate both speech and DTMF grammars at the same time. In case of matching speech input, the DTMF buffer will be flushed.

In case of errors, events are thrown as defined in chapter 5.2.6 in ref. [3] .

The following is an example of a VoiceXML document allowing the caller to say "you", "one" or "momma" as input:

```
<?xml version="1.0" encoding ="iso-8859-1"?>

<vxml version="2.1" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">

    <grammar mode='voice' root='root'>
        <rule id='root'>
            <one-of>
                <item>you</item>
                <item>one</item>
                <item>momma</item>
            </one-of>
        </rule>
    </grammar>

    <form id="form1">
        <field name="asr_field">
            <prompt><audio src="beep.wav">Say something</audio></prompt>
            <filled>
                <log>FILLED <value expr="asr_field" /> </log>
                <exit/>
            </filled>
            <noinput>
                <log>FAIL</log>
                <exit/>
            </noinput>
            <nomatch>
                <log>FAIL</log>
                <exit/>
            </nomatch>
            <catch event="error">
                <log>FAIL</log>
                <exit/>
            </catch>
        </field>
    </form>
</vxml>
```

4.5.13 Text to speech

Using PlatformAccess, the application is able to convert text to speech. The application may choose to play arbitrary text. TTS is for example used to play emails. The PlatformAccess interface makes it possible to extract and play both body and headers from the email.

The TTS function is able to play text in the mime types text/plain or text/html. The application should not attempt to play text of other mime types.

In case of a HTML document, all text intended to be read by humans will be played. Nothing will be played regarding the structuring of the HTML document. For example, text enclosed in <table> tags will be played, but information about the table itself (including that there is a table in the text) will not be played.

It is possible to mix playing of sound files and using TTS arbitrarily.

It is possible to barge-in using DTMF during TTS.

If it for some reason is impossible to perform TTS, the behavior will be as specified for audio files in ref. [3] , which implies that no audio is played and no error event is thrown. Execution of the VoiceXML document continues. The platform will log an error if it is impossible to perform TTS.

The following are some reasons for TTS to fail:

- The mime type of the text to TTS is not text/plain or text/html.
- The selected TTS language is not available.
- The maximum number of parallel TTS sessions is already reached.

The application is able to check what TTS languages are available via PlatformAccess.

The following is a simple example of a VoiceXML document performing TTS of an email's subject and body. Note that the example is simplified regarding:

- The application "knows" that there is an email in the user's mailbox. A realistic application would perform many more checks to see what it actually found in the mailbox.
- The application performs too many PlatformAccess calls sequentially in a script tag. Section 4.5.4 explains why this is a bad idea in a realistic application.

```
<?xml version="1.0" encoding ="iso-8859-1"?>
<vxml version="2.1" xmlns="http://www.w3.org/2001/vxml" xml:lang="en-US">


    <catch>
        <log>ERROR</log>
        <exit/>
    </catch>


    <form >
        <var name="languageString" expr="''"></var>


        <script>


                  <!-- first example code to code to find installed TTS languages -->
                  var lang = util.getSupportedTTSLanguages();
                  for(var i=0; i &lt; lang.length; i++){
                    languageString = languageString + " " + lang[i];
```

```
                }

                <!-- Fetch a mail for subscriber 1234 -->
                var mailboxId = mas.subscriberGetMailbox('1234');
                var folderId = mas.mailboxGetFolder(mailboxId, 'inbox');

                <!-- Get emails -->
                var messageIdList = mas.mailboxGetMessageList(folderId, 'email',
'new', 'urgent', 'type', 'fifo');
                var messageArray = mas.mailboxGetMessages(messageIdList);

                <!-- get the first message -->
                var firstMessageId = messageArray[0];
                var messageContentIdArray = mas.messageGetContent(firstMessageId);

                <!-- get mediaobjects for subject and body -->
                var subject = mas.messageGetStoredProperty(firstMessageId,
'subject')[0];
                var subjectMediaObject = util.getMediaObject(subject);
                var bodyMediaObject =
mas.messageGetMediaObject(messageContentIdArray[0]);

                <!-- detag the body to remove e.g. HTML markup -->
                var bodyText = util.convertMediaObjectsToString([bodyMediaObject]);
                var detaggedBodyText = util.deTag(bodyText);
                var detaggedBodyMediaObject = util.getMediaObject(detaggedBodyText);

        </script>

        <block>
            <log>Supported TTS languages are:<value expr="languageString"/></log>
        </block>

        <block>
            <prompt>
                <audio expr="subjectMediaObject"></audio>
                <audio expr="detaggedBodyMediaObject"></audio>
            </prompt>
        </block>

    </form>
</vxml>
```

4.5.14 SIP Message Waiting

The application is able to send a “SIP message waiting request” as described in ref. [23] by using PlatformAccess. Sending the request is asynchronous, and an

event will inform the application of the outcome. Full details are specified in the PlatformAccess API.

The application should use this feature from a CCXML document.

Below is an example:

```
<?xml version="1.0" encoding ="iso-8859-1"?>
<ccxml xmlns="http://www.w3.org/2002/09/ccxml" version="1.0">

    <eventprocessor>
        <transition event="ccxml.loaded" name="evt">
            <script>
                var parameterNames = [];
                var parameterValues = [];

                parameterNames[0] = "sendto";
                parameterValues[0] = "12322";

                parameterNames[1] = "messageaccount";
                parameterValues[1] = "12322";

                parameterNames[2] = "messageswaiting";
                parameterValues[2] = "yes";

                parameterNames[3] = "voicemessagenew";
                parameterValues[3] = "2";

                parameterNames[4] = "voicemessageold";
                parameterValues[4] = "1";

                parameterNames[5] = "faxmessagenew";
                parameterValues[5] = "1";

                parameterNames[6] = "faxmessageold";
                parameterValues[6] = "6";

                parameterNames[7] = "videomessagenew";
                parameterValues[7] = "2";

                parameterNames[8] = "videomessageold";
                parameterValues[8] = "2";

                parameterNames[9] = "emailmessagenew";
                parameterValues[9] = "3";

                parameterNames[10] = "emailmessageold";
```

```
                parameterValues[10] = "7";

                mas.systemSendSIPMessage("messagewaiting",
                                         parameterNames,
                                         parameterValues);
            </script>
        </transition>

        <transition event="com.mobeon.platform.sipmessageresponse" name="evt">

            <log expr="'RESPONSECODE '+ evt.responsecode"/>
            <log expr="'RESPONSETEXT '+ evt.responsetext"/>
            <log expr="'RETRYAFTER '+ evt.retryafter"/>

            <exit/>
        </transition>

        <transition event="error.*" name="evt">
            <log expr="'TCFAIL CCXML: Error = ' + evt.name + ', Reason =
'+evt.reason"/>
            <exit/>
        </transition>
    </eventprocessor>
</ccxml>
```

4.5.15 Resource Locating

Execution Engine can fetch resources from external servers. That means that if the "*src*" attribute on a VXML tag is an URI that points to a file on an external web-server EE must fetch the file with the protocol that is specified in the URI.

Only "*file*" and "*http*" schemes are supported now. If the URI starts with "*http*" it is HTTP that is used to fetch the data, if not it is assumed that the resource is located on the file system. When using "*file*" as the schema, the current working directory is used as the relative root.

The resources are put into a cache that is updated automatically, following the VoiceXML standard rules.

4.5.15.1 Cache

In order to speed up the system and not load the network too much the external content is cached in Execution Engine. VXML specifies a couple of cache parameters which are similar to the cache parameters specified in HTTP 1.1.

- Maxage. Defines a time interval before the data in the cache should be considered "aged". In that case the data should be refreshed.
- Fetchtimeout. Max time to wait on an external server before the connection times out.

For more information how these parameters work see [3] . Other attributes or properties used to manage caching are not supported.

Objects located in the cache that has not been used for a (long) time are removed automatically. Also if the cache is full the objects least used are removed when a new resource is put into the cache. The size of the cache is not configurable.

4.5.15.2 Example code

Here is an example on vxml code that fetches an audio file from an external web server.

```
<?xml version="1.0" encoding="utf-8"?>
<vxml version="2.1" xmlns="http://www.w3.org/2001/vxml">

    <property name="audiomaxage" value="0s"/>
    <property name="fetchtimeout" value="5s"/>
    <form id="main_form">
        <block>
            <prompt>
                <audio src="http://externalhost/c01.wav"/>
            </prompt>
        </block>
    </form>
</vxml>
```

The *audiomaxage* (same function as for *maxage* but for audio) attribute is set to 0s which means that the audio file always should be refreshed from the server.

5 External Operation Conditions

Intentionally left blank.

6 Capabilities

This paragraph is intentionally left blank.

7 References

[1] VoiceXML 2.1 recommendation
http://www.w3.org/TR/2005/CR-voicexml21-20050613/
(as of 2005-09-06)

[2] CCXML 1.0 specification (working draft)
http://www.w3.org/TR/2004/WD-ccxml-20040430/
(as of 2005-08-23)

[3] VoiceXML 2.0 recommendation
http://www.w3.org/TR/2004/REC-voicexml20-20040316/

[4] FS Application And Media Content Package Manager
1/FS-SWU 0044FS

[5] Call Manager
8/FS-MAS 0001

[6] FS Configuration Manager
16/FS-MAS 0001

[7] FS External Component Register
9/FS-MAS 0001

[8] FS Log Manager
2/FS-MAS 0001

[9] FS Mailbox
5/FS-MAS 0001

[10] FS Media Content Manager
14/FS-MAS 0001

[11] FS MediaObject
13/FS-MAS0001

[12] FS Media Translation Manager
15/FS-MAS 0001

[13] FS Message Sender
6/FS-MAS 0001

[14] FS Number Analyzer
12/FS-MAS 0001

[15] FS Operate And Maintain
17/FS-MAS 0001

[16] FS Profile Manager
4/FS-MAS 0001

[17] FS Service Request Manager
10/FS-MAS 0001

[18] FS Stream
7/FS-MAS 0001

[19] FS Traffic Event Sender
11/FS-MAS 0001

[20] CCXML 1.0 Status
3/REPORT-MAS0001

[21] VoiceXML 2.0 Status
1/REPORT-MAS0001

[22] VoiceXML 2.1 Status
2/REPORT-MAS0001

[23] IWD – SIP Message Waiting
2.IWD.MAS0001

[24] IWD – Extensible Messaging Protocol
2/155 19-1/HDB 101 02

8 SRGS DTMF support

8.1 Introduction

There is support for Inline xml DTMF Grammars with some exceptions described below.
Appendix E from The W3C SRGS recommendation is marked with green to show what is supported. Not supported features are marked with red.

8.2 Appendix E: DTMF Grammars

This appendix is normative.

This section defines a normative representation of a grammar consisting of DTMF tokens. A DTMF grammar can be used by a DTMF detector to determine sequences of legal and illegal DTMF events. All grammar processors that support grammars of mode `"dtmf"` must implement this Appendix. However, not all grammar processors are required to support DTMF input.

If the grammar mode is declared as "dtmf" then tokens contained by the grammar are treated as DTMF tones (rather than the default of speech tokens).

There are sixteen (16) DTMF tones. Of these twelve (12) are commonly found on telephone sets as the digits "0" through "9" plus "*" (star) and "#" (pound). The four DTMF tones not typically present on telephones are "A", "B", "C", "D".

Each of the DTMF symbols is a legal DTMF token in a DTMF grammar. As in speech grammars, tokens must be separated by white space in a DTMF grammar. A space-separated sequence of DTMF symbols represents a temporal sequence of DTMF entries.

In the ABNF Form the "*" symbol is reserved so double quotes must always be used to delimit "*" when defining an ABNF DTMF grammar. It is recommended that the "#" symbol also be quoted. As an alternative the tokens "star" and "pound" are acceptable synonyms.

In any DTMF grammar any language declaration in a grammar header is ignored and any language attachments to rule expansions are ignored.

In all other respects a DTMF grammar is syntactically the same as a speech grammar. For example, DTMF grammars may use rule references, special rules, tags and other specification features.

The following is a simple DTMF grammar that accepts a 4-digit PIN followed by a pound terminator. It also permits the sequence of "*" followed by "9" (e.g. to receive a help message).

```
#ABNF 1.0 ISO-8859-1;

mode dtmf;

$digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
public $pin = $digit <4> "#" | "*" 9;
<?xml version="1.0"?>

<grammar mode="dtmf" version="1.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://www.w3.org/2001/06/grammar
                             http://www.w3.org/TR/speech-
grammar/grammar.xsd"
         xmlns="http://www.w3.org/2001/06/grammar">

<rule id="digit">
 <one-of>
   <item> 0 </item>
```

```
    <item> 1 </item>
    <item> 2 </item>
    <item> 3 </item>
    <item> 4 </item>
    <item> 5 </item>
    <item> 6 </item>
    <item> 7 </item>
    <item> 8 </item>
    <item> 9 </item>
  </one-of>
</rule>

<rule id="pin" scope="public">
  <one-of>
    <item>
      <item repeat="4"><ruleref uri="#digit"/></item>
      #
    </item>
    <item>
      * 9
    </item>
  </one-of>
</rule>

</grammar>
```

Inline xml tags

Grammar

Attributes

Name	Supported	Note
root	Yes	
mode	Yes	
scope	No	
weight	n/a	Only applicable when mode is voice
fetchhint	No	
fetchtimeout	No	
maxage	No	
maxstale	No	
xms:lang	n/a	Only applicable when mode is voice
type	No	
src	No	Only inline grammars supported

Rule

There is only support for rules with a single rule-expansion.

Attributes

Name	Supported	Note
id	Yes	
scope	No	

Ruleref

The ruleref tag is supported but can only refer to rules defined within the same grammar tag as the ruleref-tag itself.

Attributes

Name	Supported	Note
type	No	
uri	Yes	See above
special	No	

Item

Attributes

Name	Supported	Note
repeat-prob	No	
repeat	Yes	
weight	n/a	Only applicable when mode is voice
xml:lang	n/a	Only applicable when mode is voice

One-of

Attributes

Name	Supported	Note
xml:lang	n/a	Only applicable when mode is voice

Tag

Not supported for DTMF