Genesys®

Genesys Voice Platform 8.1

CCXML

Reference Manual

About Genesys

Genesys is the world's leading provider of customer service and contact center software - with more than 4,000 customers in 80 countries. Drawing on its more than 20 years of customer service innovation and experience, Genesys is uniquely positioned to help companies bring their people, insights and customer channels together to effectively drive today's customer conversation. Genesys software directs more than 100 million interactions every day, maximizing the value of customer engagement and differentiating the experience by driving personalization and multi-channel customer service - and extending customer service across the enterprise to optimize processes and the performance of customer-facing employees. Go to www.genesyslab.com for more information.

Each product has its own documentation for online viewing at the Genesys Technical Support website or on the Documentation Library DVD, which is available from Genesys upon request. For more information, contact your sales representative.

Notice

Although reasonable effort is made to ensure that the information in this document is complete and accurate at the time of release, Genesys Telecommunications Laboratories, Inc., cannot assume responsibility for any existing errors. Changes and/or corrections to the information contained in this document may be incorporated in future versions.

Your Responsibility for Your System's Security

You are responsible for the security of your system. Product administration to prevent unauthorized use is your responsibility. Your system administrator should read all documents provided with this product to fully understand the features available that reduce your risk of incurring charges for unlicensed use of Genesys products.

Trademarks

The Crystal monospace font is used by permission of Software Renovation Corporation, www.SoftwareRenovation.com.

Technical Support from VARs

If you have purchased support from a value-added reseller (VAR), please contact the VAR for technical support.

Technical Support from Genesys

If you have purchased support directly from Genesys, please contact Genesys Technical Support. Before contacting technical support, please refer to the ***Genesys Care Program Guide*** for complete contact information and procedures.

Ordering and Licensing Information

Complete information on ordering and licensing Genesys products can be found in the ***Genesys Licensing Guide.***

Released by

Genesys Telecommunications Laboratories, Inc. www.genesyslab.com

Document Version: 81gvp_ref_ccxml_12-2012_v8.1.601.00

Table of Contents

Appendix B

Appendix C

Supplements

Index

Preface

Welcome to the *Genesys Voice Platform 8.1 CCXML Reference Manual*. This document provides information about developing call control applications with Call Control Extensible Markup Language (CCXML) on the Genesys Voice Platform (GVP).

This document is valid only for the 8.1 release(s) of this product.

Note: For versions of this document created for other releases of this product, visit the Genesys Technical Support website, or request the Documentation Library DVD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

This preface contains the following sections:

For information about related resources and about the conventions that are used in this document, see the supplementary material starting on page 65.

About GVP

GVP is a group of software components that constitute a robust, carrier-grade voice processing platform. GVP unifies voice and web technologies to provide a complete solution for customer self-service or assisted service.

In the Voice Platform Solution (VPS), GVP 8.1 is fully integrated with the Genesys Management Framework. GVP uses the Genesys Administrator, the standard Genesys configuration and management graphical user interface (GUI), to configure, tune, activate, and manage GVP processes and GVP voice and call control applications. GVP interacts with other Genesys components, and it can be deployed in conjunction with other solutions, such as Enterprise

Routing Solution (ERS), Network Routing Solution (NRS), and Network-based Contact Solution (NbCS).

Intended Audience

This document is primarily intended for users who will be developing call control applications with Call Control Extensible Markup Language (CCXML). It has been written with the assumption that you have a basic understanding of:

- Computer-telephony integration (CTI) concepts, processes, terminology, and applications
- Network design and operation
- Your own network configurations

You should also be familiar with HTML, XML, CCXML, and VoiceXML concepts.

Making Comments on This Document

If you especially like or dislike anything about this document, feel free to e-mail your comments to `Techpubs.webadmin@genesyslab.com`.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this document. Please limit your comments to the scope of this document only and to the way in which the information is presented. Contact your Genesys Account Representative or Genesys Technical Support if you have suggestions about the product itself.

When you send us comments, you grant Genesys a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

Contacting Genesys Technical Support

If you have purchased support directly from Genesys, please contact `Genesys Technical Support`.

Before contacting technical support, please refer to the *Genesys Care Program Guide.*

Document Change History

This section lists topics that are new or that have changed significantly since the first release of this document.

GVP 8.1.6

- Defined the "Intended Audience" on page 8.
- Modified the section "CCXML Example" on page 61.

GVP 8.1.4

- Information about HTTPS support has been added to Chapter 3.

GVP 8.1.2

- Information about MSML Dialogs has been added to Chapter 3.
- Information about creating a new CCXML session or sending the event to an existing session has been added to Chapter 3.
- Information about `<dialogprepare>` was added to Appendix A.
- A new appendix, "MSML Specification", was added to the manual.

Chapter

1 Overview

This chapter describes the Genesys Voice Platform (GVP) Call Control Platform (CCP). It contains the following section:

Introducing Call Control Platform

The Call Control Platform (CCP) component of Genesys Voice Platform (GVP) provides a Call Control Extensible Markup Language (CCXML) interpreter that integrates with existing GVP infrastructure such as the Media Control Platform (MCP) and the Resource Manager (RM). The underlying network protocol for the CCP is SIP, which means that the CCP is also interoperable with other conferencing servers or dialog servers.

Although GVP has traditionally provided extended call control capabilities through Voice Extensible Markup Language (VoiceXML), the development of CCXML provides a standard, xml-based language for scripting call control logic. Like VoiceXML, CCXML is independent of the environment in which it operates, and can run in environments ranging from Voice over IP (VoIP) based softswitch products to integrated residential gateways that manage a single telephone call. Genesys therefore recommends that new call control applications use CCXML.

The CCP currently follows the *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005.*

Chapter

2 Features

This chapter discusses the features of the Call Control Platform. It contains the following sections:

Dialing into the Call Control Platform

The Call Control Platform (CCP) accepts incoming SIP connections on port 5068 by default. You can change the port number by adjusting the configuration variable `sip.transport.x` to modify the port number used by the CCP for receiving incoming connections.

Calling to the Default CCXML Page

By default, all incoming connections will start a new CCXML session with a default URI.

For Windows, the default page is located at `C:\Program Files\GCTI\gvp\VP Call Control Platform 8.1`.

For Linux, the default page is located at `/opt/genesys/gvp/VP_Call_Control_Platform_8.1/CCP_81/config/default.ccxml`.

The default application rejects all inbound connections. You can change the location of the default page by adjusting the configuration parameter `ccpccxml.default_uri`.

Starting a non-Default CCXML Page

To start a different page with an incoming connection, the CCP follows the `netann` convention (currently `http://www.ietf.org/rfc/rfc4240.txt`). The request URI of the incoming request must follow this format:
`sip:ccxml@callcontrolplatform.genesyslab.com;ccxml=http://www.genesyslab.com/page.ccxml`

where:

1. The `userpart` of the Request URI must be `ccxml` and it is case-sensitive.
2. `callcontrolplatform.genesyslab.com` is the host or IP address of the CCP.
3. The Request URI must contain a `ccxml` URI parameter (case-sensitive) and the value is the CCXML page to be started. Other URI parameters can be included and the ordering does not matter. These additional parameters are passed to the CCP, and may be consumed by the CCP, or passed to the application server referred to by the Request URI.

Using Resource Manager to Map CCXML Applications

The Genesys Voice Platform (GVP) Resource Manager (RM), which acts as a SIP proxy, can be used to map CCXML applications by translating the SIP request URI to the `netann` format described in the preceding sub-section.

Here is an example:

The RM translates `sip:1234@10.0.0.123` into:
`sip:ccxml@10.0.0.124:5068;ccxml=file:///usr/local/ccp-ccxml/config/default.ccxml`

where

- `10.0.0.123` is the RM address.
- `10.0.0.124` is the CCP address.

Note: For information about configuring Resource Manager, see the *Genesys Voice Platform 8.1 User's Guide.*

Inbound Connections

This section describes the Call Control Platform features for inbound connections.

Passing URI Parameters to CCXML Applications

When fetching the initial CCXML page, the CCP adds parameters to the initial URL. These URL parameters are found in the incoming SIP Request URI parameters.

For example, a SIP Request URI looks like this:

```
sip:ccxml@ccxmlplatform.genesyslab.com:5068;ccxml=http://www.genesyslab.com/page.ccxml;
   hello=world
```

The initial URL fetch will be:

```
GET http://www.genesyslab.com/page.ccxml?hello=world
```

Call Parameters Accessible in CCXML Applications

Call parameters (or SIP headers) can be made accessible in the CCXML application through the session object. Table 1 shows the connection properties available through the session object.

Table 1: Connection Properties

Connection Properties (Shown via Session Variable)	Description
session.connections[connectionid].local	To: header
session.connections[connectionid].remote	From: header
session.connections[connectionid].protocol.name	sip
session.connections[connectionid].protocol.version	2.0
session.connections[connectionid].protocol.sip.callid	Call-ID header
session.connections[connectionid].protocol.sip.requesturi	Request URI
session.connections[connectionid].protocol.sip.from	From: header
session.connections[connectionid].protocol.sip.to	To: header

183 Session Progressing Response

The CCP sends a `100 Trying` response immediately upon receiving an `INVITE` request. The CCP sends a `200 OK` response when the CCXML application executes the <accept> tag. By default, the 183 Session Progressing response message is not sent.

Setting ccpccxml.sip.send_progressing configuration parameter to 1 instructs the CCP to send 183 Session Progressing along with 200 OK when the <accept> tag is executed on an inbound connection.

A CCXML application can request that the CCP send 183 Session Progressing with a <send> tag. Here is an example:

```
<send target="connectionid"
   targettype="'x-connection'"data="'connection.progressing'"/>
```

Rejecting Incoming Connections

Rejecting an incoming connection with the <reject> tag will cause the CCP to respond with a 480 Temporarily Unavailable response. Using the reason attribute in the <reject> tag will enable the use of the Reason header in the 400 Bad Request response. The header will contain the following:

```
Reason: SIP; cause=480; text="content of the reason attribute"
```

The exact SIP response code used to reject a call can be specified in the hints attribute of the <reject> tag. The responseCode property of the hints object specifies the response code that should be used, as shown below.

```
  <var name="hints" expr="new Object()"/>
  <assign name="hints.responseCode" expr="'400'"/>
. . .
<reject . . . hints="hints"/>
```

The default reject SIP code is configurable throughout the CCP and will be used if the hints attribute is not specified. The parameter mediacontroller.defaultrejectcode is the platform-wide configuration parameter for the default reject code.

Disconnecting Calls

When a connection is connected, executing the <disconnect> tag sends a BYE message on the connection to terminate the call. This applies to both inbound and outbound connections.

Using the reason attribute in the <disconnect> tag enables the use of the Reason header in the BYE message. The header will contain the following:

```
Reason: SIP; cause=200; text="content of the reason attribute"
```

Connection Signals

When a SIP INFO message is received on a connection, the CCP raises a connection.signal event. There will be two properties set in the info property:

- event.info.contenttype—the value of Content-Type header of the SIP INFO message
- event.info.content—the content of the SIP INFO message

Receiving DTMF Digits Through SIP INFO

When a SIP INFO message has a Content-Type of application/dtmf-relay, it implies that it is a DTMF digit. The connection.signal event will also contain the event.info.dtmf property that provides the DTMF digit(s).

Receiving Other Events Through SIP INFO

Other events are stored as text into event.info.contenttype and event.info.content. The SIP INFO message body is stored in event.info.content property, whereas the value of the SIP Content-Type header is stored in event.info.contenttype. Event content can be parsed using ECMAScript, as shown in the example below:

```
INFO sip:101@10.33.2.53;user=phone SIP/2.0
Via: SIP/2.0/UDP 10.33.2.53;branch=z9hG4bKac5906
Max-Forwards: 70
From: "anonymous" <sip:anonymous@anonymous.invalid>;tag=1c25298
To: <sip:101@10.33.2.53;user=phone>
Call-ID: 11923@10.33.2.53
CSeq: 1 INVITE
Contact: <sip:100@10.33.2.53>
X-Detect: Response=CPT,FAX
Content-Type: application/x-detect
Content-Length: xxx

Type = CPT
Subtype = reorder

    <transition event="connection.signal" name="evt">
      <if cond="evt.info.contenttype.toString() ==
'application/x-detect'">
        <script>
          <![CDATA[
       var mystring = evt.info.content.split("\r\n");
       var myType1 = mystring[0].split("=");
                  var myType2 = mystring[1].split("=");
           ]]>
          </script>
            <log expr="myType1[0] + '[' + myType1[1] + ']'"/>
            <log expr="myType2[0] + '[' + myType2[1] + ']'"/>
          <else/>
      <log expr="'Unwanted Event'"/>
       </if>
       <exit/>
     </transition>
```

The log will display: `Type[CPT]`, and `Subtype[reorder]`.

Outbound Connections

When making an outbound connection, provide the SIP URI in the `dest` attribute of the `<createcall>` tag. The CCP uses the given SIP URI to send an `INVITE` request. The SIP Request is sent directly to the destination.

Specifying Custom SIP Headers Through Hints

Custom SIP headers can be sent in an initial outgoing `INVITE` through the `hints` attribute of the `<createcall>` tag. The value of the `hints` attribute must be an ECMAScript object containing a subobject with the name `headers`. The `headers` ECMAScript object can then contain a name/value list of custom SIP header names and the corresponding values:

```
<var name="myhint" expr="new Object()"/>
  <assign name="myhint.protocol" expr="new Object()"/>
  <assign name="myhint.protocol.sip" expr="new Object()"/>
  <assign name="myhint.protocol.sip.headers" expr="new Object()"/>
  <assign name="myhint.protocol.sip.headers['X-Detect']" expr="'Request=CPT, FAX'"/
. . .
  <createcall . . . hints="myhint" . . . />
```

The header name/value specified through the hints will be filtered through an allowed list of custom SIP headers defined by the configuration variable `mediacontroller.sip.allowedunknownheaders`, which is a space delimited list of permitted custom header names.

Note: If the `mediacontroller.sip.allowedunknownheaders` is set to `"*"`, all unknown headers will be sent out. Also, it should be noted that `mediacontroller.sip.allowedunknownheaders="* X-Detect"` will not work; only `"*"` will work in this case.

If the header name specified in the hint has a matching header name in the configuration parameter, the first matching header name from the configuration parameter will be sent out as a custom header name with the value specified from the hint. Header names and values with no matching header name in the configuration parameter will not be sent out.

For example, if the `mediacontroller.sip.allowedunknownheaders` configuration parameter has the value of `X-Detect X-other` when `<createcall>` is called with the preceding hint example, the custom header `X-Detect: Request=CPT,FAX` will be added to the initial `INVITE`.

Table 2 lists some examples of the mapping between hint header names and custom header names sent out in the `INVITE` message:

Table 2: Mapping Examples

Header name in hint	ccpccxml.sip.allowedunknowheaders	Header name in SIP INVITE
X-Detect	X-Detect X-Channel	X-Detect
CPA	A-CPA CPA B-CPA	CPA
CPA	CPA A-CPA B-CPA	CPA
CPA	X-Detect X-Channel	Not sent

Similarly, custom SIP headers in SIP responses that result in a `connection.progressing` or `connection.connected` event (see "Mapping SIP Responses to CCXML Connection Events", below) will be available to the CCXML application if the SIP headers are configured in the `mediacontroller.sip.allowedunknownheaders` configuration parameter.

The custom SIP headers can be obtained from the CCXML application as follows:

```
session.connections[evt.connectionid].protocol.sip.headers['x-channel']
```

where `x-channel` is the SIP header name mentioned above.

Note: When the `mediacontroller.sip.allowedunknownheaders` parameter is changed, the CCP must be restarted to get the latest parameter changes.

Mapping SIP Responses to CCXML Connection Events

All `1xx` responses except `100` received from the outgoing connection result in a `connection.progressing` event.

When a `2xx` response is received, a `connection.connected` event is thrown.

When a non-`2xx` final response (`300–699`) response is received, a `connection.failed` event is thrown.

Disconnecting Progressing Call

When the `<disconnect>` tag is used on an outbound progressing call, the CCP sends a `CANCEL` message on the outgoing call to terminate it.

Call Redirection

This section describes the Call Control Platform features for call redirection.

Redirecting an Incoming Call

Using the <redirect> tag on an incoming call (in the ALERTING state) redirects the call. The CCP sends a 302 Moved Temporarily response. The dest attribute of the <redirect> tag translates to the Contact header in the 302 response.

Redirecting a Connected Call

Using the <redirect> tag on a connected call (this applies to both inbound and outbound calls) redirects the call with a REFER message. The dest attribute of the <redirect> tag translates to the Refer-To header in the REFER message. After the CCP receives a NOTIFY message with a 200 OK message, the call is considered redirected and the connection will be released. The CCXML application will receive a connection.redirected event.

If the redirection fails for any reason, the call receives an error.connection event.

Call Merge

Two connections can merge at the network level (bridging the calls at the switch) when both of them are in a CONNECTED state. The CCP uses the REFER message with Replaces as the mechanism to initiate a call merge feature at the switch. For example:

Assume the first call was connected with:

```
INVITE sip:hi@10.0.0.1 SIP/2.0
Via: SIP/2.0/UDP
From: sip:bye@10.0.0.2
To: sip:hi@10.0.0.1
Max-Forwards: 70
CSeq: 1 INVITE
Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E
Contact: sip:bye@10.0.0.2:5060
Content-Length: 147
Content-Type: application/sdp
```

Assume the second call was connected with:

```
INVITE sip:hello@10.0.0.1 SIP/2.0
Via: SIP/2.0/UDP
From: sip:world@10.0.0.3
To: sip:hello@10.0.0.1
Max-Forwards: 70
```

```
CSeq: 1 INVITE
Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123
Contact: sip:world@10.0.0.3:5060
Content-Length: 147
Content-Type: application/sdp
```

Table 3 describes the Merge SIP call flow.

Table 3: Merge SIP Call Flow

Event	Direction	Message
<merge>	→	REFER sip:bye@10.0.0.2 SIP/2.0 Via: SIP/2.0/UDP 10.0.0.1:5060 From: sip:hi@10.0.0.1 To: sip:bye@10.0.0.2 Cseq: 2 REFER Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E Refer-To: world@10.0.0.3;Replaces=DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123
	←	SIP/2.0 202 Accepted … Cseq: 2 REFER
connection.merged	←	NOTIFY sip:bye@10.0.0.2 SIP/2.0 … Cseq: 3 NOTIFY Event: refer Content-Type: message/sipfrag;version=2.0 Content-Length: 14 SIP/2.0 200 OK
	→	SIP/2.0 200 OK … Cseq: 3 NOTIFY …
	→	BYE sip:bye@10.0.0.2 SIP/2.0 … Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E …

Table 3: Merge SIP Call Flow (Continued)

Event	Direction	Message
	→	BYE sip:world@10.0.0.3 SIP/2.0 … Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123 …
	←	SIP/2.0 200 OK … Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E …
	←	SIP/2.0 200 OK … Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC123 …

Note: If the CCXML application has a `<merge>` tag followed immediately by a `<disconnect>` tag in the same transition, the platform will issue only one `connection.merged` event and one `connection.disconnected` event instead of two `connection.merged` events on both of the connections.

Dialogs

This section describes the dialogs that the Call Control Platform supports.

Preparing Dialogs

The `<dialogprepare>` or `<dialogstart>` tags create a new dialog; the CCP initiates a new SIP dialog to the dialog server. The CCP sends an `INVITE` message to the Resource Manager (configurable with the `mediacontroller.sipproxy` parameter) with the following `netann` request URI:

```
sip:dialog@sipproxy.genesyslab.com;voicexml=http%3F//www.genesyslab.com/page.vxml
```

Where `sipproxy.genesyslab.com` is the value of the configuration parameter `mediacontroller.sipproxy`.

Using `<dialogprepare>` to prepare a dialog will send a connectionless SDP to the dialog server to let the dialog server (Media Control Platform [MCP] in this case) prepare the dialog without starting the audio. When the `INVITE` transaction is `ACK`ed, the dialog is fetched and loaded on the MCP, and then is essentially *on hold.*

A connectionless SDP represents an SDP content that would put the MCP on hold. The SDP content will depend on the device profile configuration of the dialog server.

Passing Dialog Results Back to CCXML

VoiceXML pages can return results back to the CCP by adding content to the `BYE` message. The VoiceXML page can use the `namelist` attribute in the `<exit>` tag to send dialog results back to the CCXML application.

Here is an example in which the VoiceXML application ends the call with `<exit namelist="hello a"/>`:

The MCP sends `BYE` to the CCP:

```
BYE sip:10.0.0.1:5060 SIP/2.0
Via: SIP/2.0/UDP 10.0.0.3
Via: SIP/2.0/UDP 10.0.0.2:5060
From: sip:genesyslab@10.0.0.2
To: sip:10.0.0.1:5060
Max-Forwards: 69
CSeq: 1 BYE
Call-ID: DC9D0D00-F5CD-6037-C2A2-6BDBE04CC38E
Content-Length: 16
Content-Type: application/text

hello=world
a=b
```

The `dialog.exit` event contains:

```
values.hello = 'world'
values.a = 'b'
namelist='hello a'
```

The `dialog.disconnect` event is not currently supported by the CCP. When a VoiceXML application exits, `dialog.exit` will be thrown.

Dialog User Event

The VoiceXML dialog may send a user event to the CCXML application by using the `<send namelist="name type uri"/>` tag. Here is an example of the VoiceXML `<send>` block:

```
<var name="name" expr="'transfer'"/>
<var name="type" expr="'bridge'"/>
<var name="uri" expr="'1111@205.150.90.19'"/>
<gvp:send namelist="name type uri"/>
```

The CCXML session receives the following:

```
15:02:04.416 Int 51030 F9187A00-E558-44C6-61AE-FFA9A066180C-FF326086-ECB5 dlg_event
   7|dialog.user.transfer|DD92E8B2-51AD-4F3F-8C8D-40AFA169EA9B|values.name="transfer";v
   alues.type="bridge";values.uri="1111@205.150.90.19
```

This raises a `dialog.user.transfer` event to the CCXML application that owns the dialog. The event itself contains the following properties:

- `event$.values.name=transfer`
- event$.values.type=bridge
- event$.values.uri=1111@205.150.90.19

Note: The `event$` is a generic name for CCXML events, and in the preceding example, it is `dialog.user.transfer`.

The `contenttype` attribute is not supported by the `<send>` tag if the `namelist` is used.

Dialog-Initiated Blind Transfer

To initiate a dialog-initiated blind transfer, the VoiceXML application must call `<transfer destexpr="number_to_call" bridge="false" type="unsupervised">`.

The following sequence of events occurs:

1. The MCP sends a `REFER` message on the SIP dialog.
2. The CCXML application receives a `dialog.transfer` event. The `type` attribute is `blind` and the `uri` attribute is the `destexpr` in the `<transfer>` tag.
3. The CCXML application executes `<redirect>` to move the call specified in the `dialog.transfer` event.
4. If redirection is successful, the CCXML application sends `telephone.disconnect.transfer` event to the dialog.
5. The CCP sends `NOTIFY (200 OK)` to report the result of the transfer.
6. If redirection fails, the CCXML application sends `error.transfer.noroute` event to the dialog.
7. The CCP sends `NOTIFY (500 Server Internal Error)` to report a transfer failure.
8. The VoiceXML application receives a `telephone.disconnect.transfer` event to end the transfer and the VoiceXML page. The result is recorded in the metrics file of the MCP.

Note: When the inbound call is made through SIP Server, a dialog-initiated blind transfer will only work if the SIP Server has the appropriate DN trunk group set up and enabled to do refer transfer. This is set up by setting the `'refer-enabled=true'` on the SIP Server.

Dialog-Initiated Supervised Transfer

A dialog-initiated supervised transfer is application driven in both the MCP and CCP. The MCP sends a SIP `REFER` message to the CCP when the VoiceXML `<transfer>` is invoked for a supervised transfer. The CCP will throw a `dialog.transfer` event to the application with the `type` attribute of the event set to `blind`.

Note: The `type` attribute is always populated to `blind` whether the request from the VoiceXML is for blind transfer or supervised transfer. The CCP application developer should write the application according to either blind or supervised transfer.

Dialog-Initiated Bridge Transfer

A dialog-initiated bridge transfer is application driven from both the MCP and CCP perspective. Within the VoiceXML application, you can use the `<send>` tag (translating to `SIP INFO`) to inform the CCP of a bridge transfer request.

The CCP application can be written according to the *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005, Appendix D* for bridge transfer.

MSML Dialogs

MSML allows CCXML applications to have additional control over the media operations beyond what VoiceXML can offer. For example, the customer can create a MSML script that performs CPA, and then, based on the CPA result, it can choose to either start the VoiceXML dialog or play a prerecorded prompt.

Genesys CCXML supports MSML dialogs (the dialog packages include Dialog Core, Dialog Base, and Dialog CPA) by extending the CCXML specification. CCMXL does not support the MSML Conference Core package. For the list of supported tags, see Appendix C on page 51.

When `<dialogstart>` is used without `<dialogprepare>`, you must set the type attribute to `application/vnd.radisys.msml+xml`. In this case, the `src` attribute content is ignored. You can specify the MSML body inline as follows:

Note: For `<dialogprepare>`, the same rules for `<dialogstart>` applies on `type`, `src`, and the inline MSML body. The inline MSML body should not contain the `<?xml>` tag.

The developer should include the attribute `xmlns` as shown below so that you can inline MSML markups in the CCXML document.

```
<dialogstart src="." type="application/vnd.radisys.msml+xml" connectionid="..."
   xmlns="urn:ietf:params:xml:ns:msml">
  <msml>
```

```
    <dialogstart type="'application/moml+xml'">
      <play>
        <audio uri="'http://example.com/dictionary.vox'"/>
      </play>
     </dialogstart>
  </msml>
</dialogstart>
```

Alternatively, you can use `<dialogprepare>` before a `<dialogstart>`. If you use `<dialogprepare>`, the `type`, `src` attributes, and the inline MSML body are required by the `<dialogprepare>` tag and must not be repeated in the `<dialogstart>` tag. The MSML information will be sent with the initial `INVITE` generated by the `<dialogprepare>` with an on-hold SDP.

```
<dialogprepare src="." type="application/vnd.radisys.msml+xml"
   connectionid="connectionid" xmlns="urn:ietf:params:xml:ns:msml">
  <msml>
    <dialogstart type="'application/moml+xml'">
      <play>
        <audio uri="'http://example.com/dictionary.vox'"/>
      </play>
     </dialogstart>
  </msml>
</dialogprepare>
.
.
.
<dialogstart prepareddialogid="dialogid"/>
```

Where `connectionid` and `dialogid` are generated by CCXML.

Example of the initial `INVITE` for an MSML dialog

```
INVITE sip:dialog@genesyslab.example.com;
moml=cid:14864099865376@genesyslab.example.com SIP/2.0
     ...
Content-Type: multipart/mixed; boundary=boundary
 --boundary
Content-Type: application/sdp
SDP BODY
--boundary
Content-Id: <14864099865376@genesyslab.example.com>
Content-Type: application/vnd.radisys.msml+xml
```

The resulting MSML body from the preceding example should look like this:

```
<?xml version="1.0"?>
<msml>
  <dialogstart type="'application/moml+xml'">
    <play>
```

```
     <audio uri="'http://example.com/dictionary.vox'"/>
   </play>
  </dialogstart>
</msml>
```

Note that CCP/CCXML does not modify the MSML markups in any way other than placing the XML header at the top.

```
--boundary--
```

Once the dialog is started, Media Server can send MSML information using SIP `INFO` messages.

```
INFO sip:dialog@genesyslab.example.com SIP/2.0
...
Content-Type: application/vnd.radisys.msml+xml
<?xml version="1.0"?>
<msml>
  <event name="msml.dialog.exit" id="conn:1234/dialog:1234"/>
</msml>
```

The MSML body is available through the `dialog.user.msml` event's `info.content attribute`, with `info.contenttype = "application/vnd.radisys.msml+xml"`

The `info.content` will look like the following:

```
<?xml version="1.0"?>
<msml>
  <event name="msml.dialog.exit" id="conn:1234/dialog:1234"/>
</msml>
```

VoiceXML Session Variables

The MCP does not support `session.connection.ccxml` VoiceXML session variables, as mentioned in Appendix D of the CCXML specification.

Conferences

When a CCXML application joins to a conference, the CCP sends an `INVITE` message to the RM with a specially formatted `netann` Request URI:

```
sip:conf=ABCD1234@10.0.0.1;confinstid=ABCD1234;confreserve=3;confmaxsize=3
```

where

- `conf, confinstid` are cluster-wide unique conference identifiers
- `confreserve` is the number of conference participants to reserve for this conference
- `confmaxsize` is the maximum size of this conference

The sum of the `reservedtalkers` and `reservedlisteners` attribute in the `<createconference>` tag represents the number of conference participants to reserve for this conference. The default value can be set using the configuration parameter mediacontroller.conference.defaultreserve.

The maximum size of the conference is equal to confreserve by default. This value can be set in the hints attribute of the <createconference> tag; it is the maxsize property of the hints object.

In a clustered environment where multiple conference servers are available, the CCP relies on GVP RM to forward the requests for the same instance of a conference to the same conference server. This is a feature of the RM.

Note: The <createconference> tag proceeds successfully even if the cluster has no conferences available or no conference servers can serve the requested conference size. A <join> operation may fail due to the preceding reasons and returns error.conference.join event.

Implicit Transcoding and Conferencing

The Resource Manager can be configured with bridging server information, to handle CCXML operations that require implicitly connecting endpoints to a media server.

Note: The MCP can be used as a bridging server.

There are two cases in which a bridging server may be used internally by the CCP:

1. Audio-transcoding between endpoints which do not share common codecs
2. Multiple sessions listening to a single media stream through the use of an RTP splitter/proxy or an implicit conference

The CCP determines whether implicit transcoding or conferencing is required upon evaluation of the CCXML application. The connection of the endpoints to the bridging server will be transparent to the CCXML application.

Implicit Transcoding

When a CCXML session specifies a join between two endpoints that do not share any common audio codecs, the CCP uses the bridging server internally to transcode the media between the endpoints (see Figure 1 on page 29).

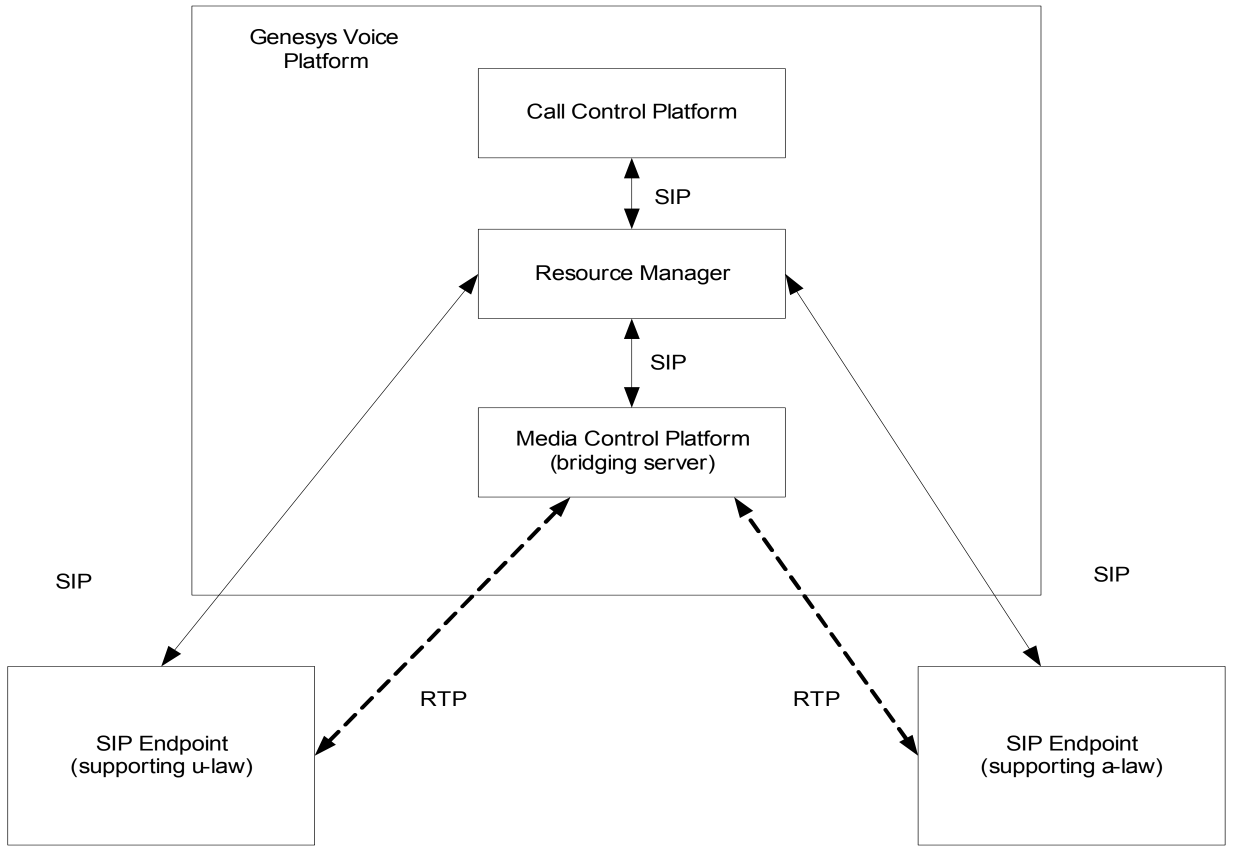
Genesys Voice Platform
Call Control Platform
SIP
Resource Manager
SIP
Media Control Platform (bridging server)
SIP
SIP
RTP
RTP
SIP Endpoint (supporting u-law)
SIP Endpoint (supporting a-law)

Figure 1: Implicit Transcoding

Note: If call legs are being joined implicitly with the `<createcall>` tag, and the call legs do not provide their codec capabilities either in the initial `INVITE` to CCP or in the `200 OK` response to the initial `INVITE` sent by CCP, the CCP will not use the transcoding feature even if a transcoding server has been defined in the `mediacontroller.bridge_server` option. This is because the CCP requires knowledge of the codec capabilities of each call leg before creating bridges so that it can determine whether a bridging server is needed.

The workaround is to connect each call leg separately and join them together after they are in the connected state.

Implicit Conferencing

A CCXML session can specify multiple joins to a single endpoint so that it is required to *split* its output and send to multiple destinations. If the sender does not support splitting its output to send to multiple destinations concurrently, the

CCP internally uses the Bridging Server to do the RTP media splitting as shown in Figure 2.

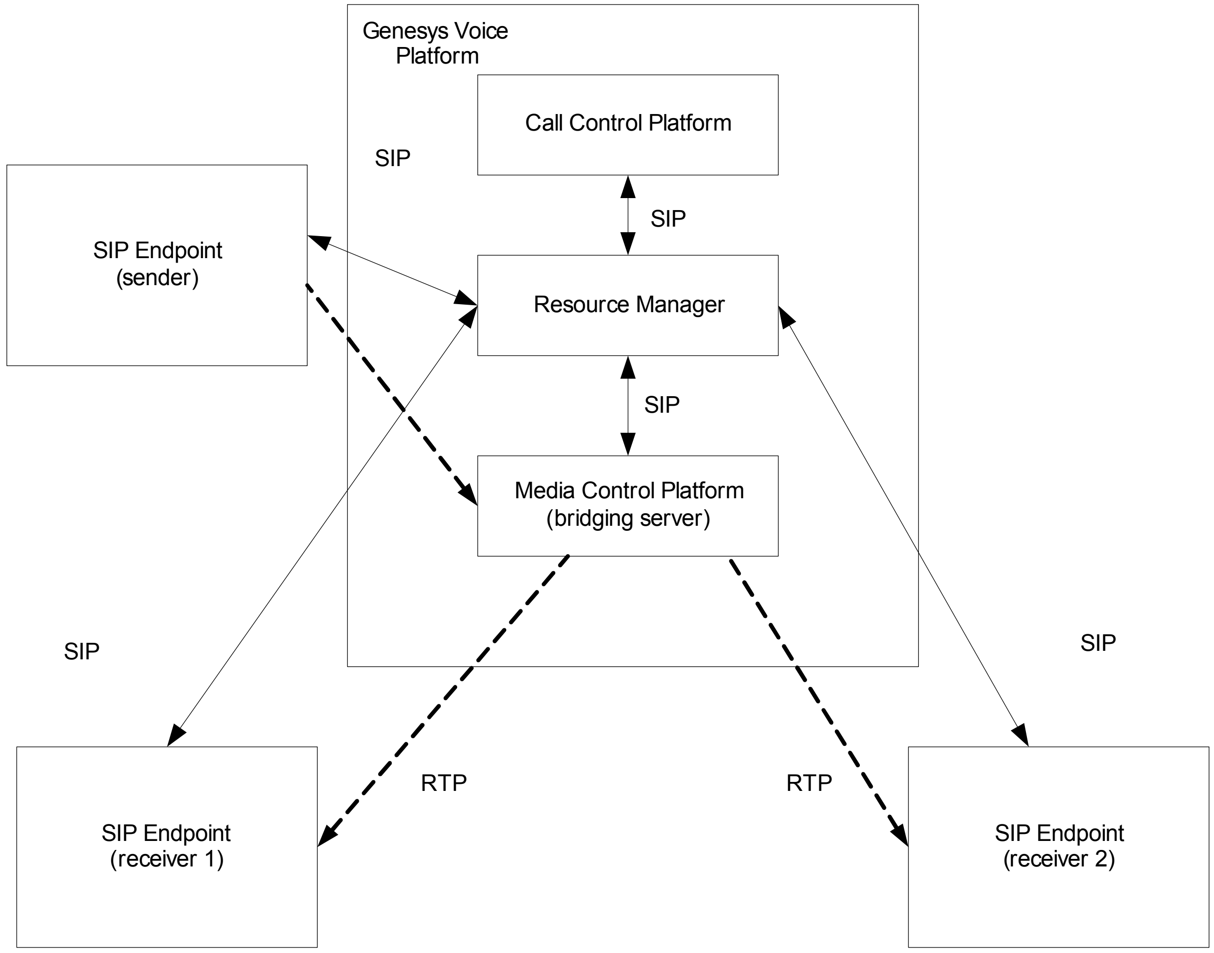

Figure 2: Implicit Conferencing

The bridging server can be accessed directly from the CCP or via the RM as shown in the diagram below. The `mediacontroller.bridge_server` configuration parameter is used to specify the location of the bridging server. If the bridging server is to be accessed via the RM, the location of the RM should be specified in the `mediacontroller.bridge_server` configuration parameter.

The MCP can be configured to act as a bridging server.

Device Profile Configuration

The CCP provides a set of device profiles that reflect Genesys' current knowledge regarding the behavior of various devices that interact with the platform. For additional information about the configuring device profiles, see the *Genesys Voice Platform 8.1 User's Guide.*

Inbound Connections

The CCP provides regular expression matching for incoming connections in order to select the most appropriate Device Profile for these connections.

The CCP will try to match the SIP User-Agent header from the incoming SIP `INVITE` to the SIP Header Name property value that is defined in the Device Profile with the highest precedence first. If there is a match, the matching Device Profile will be assigned to the connection and the CCP will use the Device Profile parameters to determine the correct behavior.

If there is no match with the SIP `Header Name` property value, the CCP will try to match the SIP User-Agent header to the Device Profile with the next precedence. If there are no matches with any preset Device Profile, the Default Inbound Device Profile will be used for the connection.

Outbound Connections

The device profiles of outbound connections, dialogs, and conferences can be specified in CCXML hints. The value of the device profile hint should be the same value as the Device Profile Name from the Device Profile configuration. The example below illustrates the use of hints for an outbound connection. Similarly, the same hint can be used for `<dialogprepare>`, `<dialogstart>`, and `<createconference>`. If the hint is already passed in `<dialogprepare>`, the subsequent `<dialogstart>` should not reconfigure the device profile hint.

Note: The timeout value for `<createcall>` may not work correctly if the value is less than 32 seconds. If the destination does not respond at all (for example, a device is down), the timeout will not occur until after the 32 second period of `INVITE` re-transmission (as dictated by RFC3261) has passed.

```
<var name="myhint" expr="new Object()"/>
<assign name="myhint.deviceprofile" expr="'GVP MCP'"/>
. . .
<createcall . . . hints="myhint"/>
```

Limitations

The CCP supports up to 100 conference participants in a conference.

Forward join to a conference that is running on an MCP will not join properly. Avoid this limitation by always using duplex join to join a conference.

In the `Offer-Answer` case, an incoming connection cannot be joined to two outbound connections if the alerting connection is accepted in a later transition than the joins. Avoid this limitation by accepting the alerting connection in the same transition as the joins or in an earlier transition.

If an existing media loop is completely reversed in the same transition, some media might be missed at some endpoints.

For example, initially A->B->C->A

The CCXML application contains multiple joins to reverse the initial bridges to A<-B<-C<-A.

Avoid this by unjoining the bridges in separate transitions.

The options-support Device Profile parameter should be set to false when the CCP is used in conjunction with Resource Manager (RM) or a SIP Proxy. If the CCP is directly connected to SIP User Agents, the options-support feature can be used to query the SIP capabilities of the devices.

If the Default Conference device profile is not configured or <createconference> refers to the device profile that is invalid, error.conference.create will be thrown. The error.conference.create event in this case will not contain the conferenceid since the <createconference> operation was aborted prematurely.

Chapter

3 Event I/O Processor

The Call Control Platform (CCP) supports three event processors:

- `basichttp`
- `createsession`
- `platform`

The `basichttp` and `createsession` event I/O processors use the HTTP protocol and are based on the *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005.* The platform event I/O processor is a Genesys extension.

The following sections describe the features supported by CCP:

Session Variable

The session variable `session.ioprocessors` is an associative array and contains a list of external event I/O access URIs, which are available to the current session. The array is associative and each key in the array is the type of the event I/O processor. Currently, this array has only two items:

- `session.ioprocessors["basichttp"]`
- `session.ioprocessors["createsession"]`

Receiving Events

An external HTTP client may make an HTTP POST request to the CCXML platform. The URI that accepts the request is exposed as specified in the preceding session.ioprocessors["basichttp"] session variable. The HTTP request is analyzed by the basichttp event I/O processor, resulting in:

- An event being injected into an active CCXML session, or
- No action being taken (an error occurred, or operation not permitted, for example)

The basichttp event I/O processor then reports its result to the external client in the response for the originating HTTP request.

HTTP POST request parameters (within an application/x-www-form-urlencoded body) are used to specify the information that is to be injected into the session. In particular, the parameters shown in Table 4 have special meanings:

Table 4: HTTP Request Parameters

HTTP Parameter Name	Meaning
sessionid	This is the ID of the session destined to receive the event. This parameter is required.
eventname	This is the name of the event to be received by the CCXML session. This parameter is required. Valid event names consist of alphanumeric characters and periods only. The first character of an event name must be a letter.
eventsource	This value specifies a URI to which events may be sent (that is, it may be used as the value of the target attribute in a <send> element). This parameter is optional and can be in any form.

When an event is successfully thrown inside the target session, the evensourcetype property of the event object is set to basichttp and the eventide property contains a unique event id (generated by the basichttp event I/O processor) for the event. When provided in the HTTP request as parameters; eventsourcetype or eventid parameters are ignored.

Other parameters provided in the HTTP request are treated as the event payload. Payload parameter names must be valid ECMAScript variable names. Qualified parameter names (for example, x.y.z) are nested inside parent parameters (for example, y and its parent x). Reserved and payload parameter values must be valid ECMAScript expressions.

The CCP replies to the HTTP request with one of the HTTP response codes shown in Table 5 on page 35.

Table 5: HTTP Response Codes

Response Code	Condition
204	The sessionid parameter matches an existing CCXML session ID, and the event name and payload parameters are valid.
400	One or more parameters has an invalid name or value or there are conflicts (for example, both x and x.y defined).
403	Failure occurs due to other reasons (for example, the session ID does not match an existing CCXML session ID, or the matched session is terminating).

Table 6 describes the event attributes of an event that was successfully received via HTTP request:

Table 6: Event Attributes

Attribute Name	Description
name	The value of the name parameter.
eventid	A unique string identifier for the event generated by the CCP.
eventsource	The value of the eventsource parameter if provided; otherwise this event attribute is undefined.
eventsourcetype	Always has the value basichttp.
<param-name>	For each param-name=value appearing in parameter name in the HTTP request, the parameter param-name appears as a property (or a nested property) in the event object. Its value is set to value.

HTTPS Support

The following GVP configuration parameters must be used:

- ccxmli.ssl must be True.
- ccxmli.ssl.recv.cert_file sets the path and the filename of the SSL certificate to be used for createsession and BasicHTTP.
- ccxmli.ssl.recv.private_key_file sets the path and the filename of the SSL key to be used for createsession and BasicHTTP.
- ccxmli.ssl.recv.password sets the password associated with the certificate and key pair. Required only if the key file is password protected.

- `ccxmli.ssl.recv.protocol_type` must be set to the following appropriate value for SSL or TLS:
  - `SSLv3`, `SSLV2` or `SSLv23`
  - `TLSv1` or an empty string that defaults to TLS

  Note: The HTTPS protocols `SSLv2`, `SSLv3`, and `TLSv1` will also work when the protocol type is set to `SSLv23`.

- For SSLv2 and SSLv3, `ccpccxml.fips_enabled` must be `False`.

Sending Events

The CCXML session may send an event to an external entity by using the `<send>` tag, as described in *Section 9.2.3* of the CCXML specification. Inline content for `<send>` is not currently supported; only the `namelist` attribute is supported.

For an example of how a web application can send an event to a specific CCXML session, see the *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005, Appendix K - Basic HTTP Event I/O Processor,* specifically, K.2 and K.3, which contains the information and the example.

Table 7 describes how the various attributes of `<send>` map to the HTTP request:

Table 7: <send> Attributes

Attribute Name	Meaning
targettype	If this attribute is set to `basichttp`, the message will be routed to the HTTP I/O Processor. **Note:** When the `<send>` element is executed with this attribute, the target CCXML session must belong to the same tenant (sender and receiver tenant IDs matches). Otherwise, `error.send.failed` is thrown with a reason property of `Tenant ID mismatch`.
target	This is the HTTP URL to which a `POST` request will be made.
name (or data)	This is an ECMAScript expression evaluating to the event name. This attribute is required for sending to the `basichttp` event I/O processor. This will become the value of the name parameter of the HTTP request.
xmlns	This attribute is not supported, because the current platform does not support the sending of inline content.

Table 7: <send> Attributes (Continued)

Attribute Name	Meaning
namelist	This is an optional parameter, and if it is defined, its variable names and values are mapped to HTTP parameters.
hints	timeout

The `basichttp` event I/O processor interprets the HTTP response codes in the following way as shown in Table 8:

Table 8: Response Codes

Response Code	Interpretation
2xx	The `<send>` was successfully accepted by the HTTP server and a `send.successful` event is posted to the session issuing the `<send>`.
Any other HTTP response code	The `<send>` was not accepted by the HTTP server and a `error.send.failed` event is posted to the session issuing the `<send>`.

Creating Sessions

An external entity can initiate a new CCXML session using HTTP `POST` via the `createsession` event I/O processor (as per the *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005*).

The access path of the URI used by the `createsession` I/O event processor is configurable and defaults to `/ccxml/createsession`. For example, if the CCP hostname is `server.example.com` (and the default HTTP port 80 is used), the URI for the event I/O processor is `http://server.example.com/ccxml/createsession`. This URI is exposed as specified in the preceding `session.ioprocessors["createsession"]` session variable.

The form url-encoded parameters in the body of the HTTP `POST` request determine the parameters for session creation. The `uri` parameter determines the initial URI of the initial CCXML page for the new session. The optional `eventsource` parameter indicates a URI to which events can be returned using the `basichttp` event I/O processor.

The `eventsource` value is exposed to the session as a property of the `session.values` session variable (that is, `session.values.eventsource`). The remaining parameters are used to create additional properties of the `session.values` object. All parameter values are treated as strings. The property names may be qualified to specify subobjects.

Multiple parameter values (for example, `var=val1&var=val2`) are not supported and result in a `400` HTTP response.

The `type` property of the `session.values` must be set to `createsession`. If this property is specified in the request body parameters, the specified value is ignored.

If `method`, `postbody`, `timeout`, `maxage` and/or `maxstale` parameters are specified, they have the same effect as the equivalent request URI parameters in SIP-initiated session creation.

If the create session request can be completed successfully, then a `200` HTTP response code replies to the request. The response body is an application/x-www-form-urlencoded name-value pair list in which the `session.id` parameter specifies the id of the newly created session.

The `gvp-tenant-id` parameter is a new parameter for creating sessions. The parameter value is in the format, `[Tenant-name].IVR-name`. For example, `[Customer1]. Profile1` specifies the IVR Profile named `Profile1` under the tenant named `Customer1`. Profiles that are created under the Environment tenant (which is the root tenant), use the tenant named `Environment`, for example, `[Environment]. Profile1`.

Creating a new CCXML session or sending the event to an existing session

The following configuration parameters control the address:

- `Ccxmli.createsession.recv.path = "/ccxml/createsession"`
- `Ccxmli.createsession.recv.port = "4892"`
- `Ccxmli.basichttp.recv.path = "/ccxml/basichttp"`
- `Ccxmli.basichttp.recv.port = "4892"`

The following example shows how a web page can create a CCXML session in the GVP platform. If you want to send an event to an existing session, use `/ccxml/basichttp` instead, with the parameter `sessionid` containing the real ID of that session.

```
<link rel="stylesheet" type="text/css" href="test.css"/>

<form action="http://138.120.84.95:4892/ccxml/createsession" method="post">
    <div>
        <label for="uri">uri</label>
        <input type="text" name="uri" id="sessionid"
   value="file:///c:/testpages/external_simpledialog.ccxml"/>
    </div>
    <div>
        <label for="eventsource">eventsource</label>
        <input type="text" name="eventsource" id="eventsource" value="SOURCE"/>
    </div>
    <div>
        <label for="eventsourcetype">eventsourcetype</label>
        <input type="text" name="eventsourcetype" id="eventsourcetype"
   value="createsession"/>
    </div>
    <div>
        <label for="method">method</label>
        <input type="text" name="method" id="method" value="get"/>
    </div>
    <div>
        <label for="timeout">timeout</label>
        <input type="text" name="timeout" id="timeout" value="30s"/>
    </div>
    <div>
        <label for="maxage">maxage</label>
        <input type="text" name="maxage" id="maxage" value="60"/>
    </div>
    <div>
        <label for="maxstale">maxstale</label>
        <input type="text" name="maxstale" id="maxstale" value="30"/>
    </div>
    <div>
        <label for="postbody">postbody</label>
        <input type="text" name="postbody" id="postbody" value="n1=v1&amp;n2=v2"/>
    </div>
    <div>
        <label for="name1">name1</label>
        <input type="text" name="name1" id="name1" value="value1"/>
    </div>
    <div>
        <label for="name2">name2</label>
        <input type="text" name="name2" id="name2" value="value2"/>
    </div>
    <div>
        <label for="complex.name3">complex.name3</label>
        <input type="text" name="complex.name3" id="complex.name3" value="value3"/>
    </div>
    <div>
        <label for="gvp-tenant-id ">gvp-tenant-id</label>
        <input type="text" name="gvp-tenant-id" id="gvp-tenant-id"
   value="CCXMLSimpleDialogLoad"/>
    </div>
    <div id="submit">
        <input type="submit"/>
    </div>
</form>
```

Error Handling

If the event properties are not valid, a `400` response is given to the request. Event properties can be considered invalid if, for example, they are not valid

ECMAScript variable names, or if multiple values are specified or are conflicting (for example, `obj1` and `obj1.x` are both given a value).

If a fetch timeout is specified in the `createsession POST` parameters and the timeout expires before the initial CCXML fetch completes, a `408` response is returned to the `createsession` request.

If the fetch or compilation or initialization of the initial CCXML page URI that is specified by the `uri POST` parameter of a `createsession` request fails for any reason, a `403` response is returned to the request.

If one of the scripts statically referenced by the CCXML page that is specified by the `uri POST` parameter of a `createsession` request cannot be fetched or compiled for any reason, a `403` response is returned to the request.

If the fetch of one of the scripts statically referenced by the page that is specified by the `uri POST` parameter of a `createsession` request times out, a `408` response is returned to the request.

For this example, assume that the value of the session variable `session.ioprocessors["basichttp"]` is `http://ccxml.genesyslab.com/ccxml/basichttp`. When the following HTTP request is made to this platform:

```
POST http://ccxml.genesyslab.com/ccxml/basichttp?sessionid=ccxmlsession1&
eventname=basichttp.myevent&eventsource=http://www.example.org/
ccxmlext&
agent=agent12&site=Orlando HTTP/1.0
. . .[other HTTP headers]. . .
. . .[other HTTP headers]. . .
```

If `ccxmlsession1` (value of the `sessionid` parameter in the preceding HTTP request) matches the session ID of an existing CCXML session, an event with the name `basichttp.myevent` is triggered in the session `ccxmlsession1`. It may be handled as follows:

```
<transition state="'dialogActive'" event="basichttp.*" name="evt">
    <log expr="'Received event'" />
    <log expr="'name=' + evt.name" />
    <log expr="'sourcetype=' + evt.eventsourcetype" />
    <log expr="'source=' + evt.eventsource" />
    <log expr="'agent=' + evt.agent" />
    <log expr="'site=' + evt.site" />
</transition>
```

where:

- `evt.name` would have the value `basichttp.myevent`
- `evt.eventsourcetype` would have the value `basichttp`
- `evt.eventsource` would have the value `http://www.example.org/ccxmlext`
- `evt.agent` would have the value `agent12`
- `evt.site` would have the value `orlando`

Additionally, the CCP responds with a `204` HTTP response code:

```
HTTP/1.0 204 No Data
```

Example of Sending Events via HTTP

Consider the following CCXML code snippet in the CCXML session with session ID `ccxmlsession2`:

```
<script>
    var agent='agent21';
    var site='miami';
</script>
<send target="'http://travel.genesyslab.com/travelagent'" data="'myevent'"
targettype="'basichttp'" namelist="agent site"/>
```

With this CCXML snippet, the following HTTP `GET` request is made:

```
GET http://travel.genesyslab.com/travalagent?sessionid=ccxmlsession2&
eventname=myevent&agent=agent21&site=miami HTTP/1.0
CRLF
```

Appendix

A CCXML Specification Support Notes

This appendix describes the GVP support for CCXML features. The Call Control Platform currently follows the *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005.*

This chapter contains the following section:

Current Support

xmlns Attribute

The xmlns attribute in <ccxml> element is not supported. Additional namespaces specified in the <ccxml> tag are ignored.

The xmlns attribute for the <send> tag is not supported. Therefore, the inline content for <send> is currently not supported; only the namelist attribute is supported.

http-equiv Attribute

The http-equiv attribute in the <meta> element is not supported and is ignored.

<metadata>

The <metadata> tag is not supported and is ignored.

UTF Character set

The CCP does not support ECMAScript scripts or CCXML pages that are authored using the UTF-16 character set.

The CCP does not support the compiling and processing of ECMAScript pages using the UTF-8 character encoding.

<fetch>

If the fetch of the URI specified by the next attribute of `<fetch>` fails for any reason, the `error.fetch` event is thrown. If the URI has a scheme of `http:` the reason property of the event will read: `Fetch failed: <error code> <reason phrase>` where `<error code>` is the HTTP error code and `<reason phrase>` is the HTTP reason phrase in the response.

prepareddialogid Attribute

From the W3 specification; if the `prepareddialogid` attribute is specified and a `connectionid` or `conferenceid` attribute was specified on the prior `<dialogprepare>` element, specifying a different `connectionid` or `conferenceid` on the `<dialogstart>` element will result in the throwing of an `error.dialog.notstarted` event.

Repeated Parameter Names

The W3C specification states that *parameter names may not be repeated within a request. A request with repeated parameter names is considered to be invalid, and should be rejected by the `basichttp` event I/O processor*. Repeated parameter names in an HTTP request to I/O processors currently does not result in a 400 response.

<move>

Conference allocation and the CCXML `<move>` tag do not work across multiple machines.

<join> and <unjoin>

CCXML applications can use `<join>` and `<unjoin>` at any time, except in the case of dialogs, where `<join>` and `<unjoin>` can only be used on dialogs that have been started.

The CCP does not support an early join for an outbound call that is being joined to a conference.

dialog.disconnect Event

The `dialog.disconnect` event is currently not supported by the CCP.

User Event

The user event from a VoiceXML dialog cannot be a multi-level object; only simple name-value pairs are supported.

AAI Feature

CCXML does not support the AAI feature; AAI data passed into the CCP with an incoming Request URI cannot be accessed at an application level.

CCXML does not support emitting AAI in the CDR.

URI Parameters

The CCP platform does not allow for a default set of initial URI parameters to be configured.

HTTPS and Session Cookies

HTTPS and session cookies are not supported by the HTTP server interface of the CCP.

<createccxml>

The `<createccxml>` parameters attribute passed into the created session are not supported. The attribute contains a namelist of CCXML parameters that will be created as properties of the `session.values` session variable in the new session. For example, if the parameters attribute has a value of `foo.bar test`, the values of those variables will be assigned to the `session.values.foo.bar` and `session.values.test` variables in the new session.

If a CCXML session that was created by another session using `<createccxml>` exits for any reason other than executing `<exit>` (for example, it does not catch an `error.*` event), queuing `ccxml.exit` to the parent session is not supported.

Moving a Connection or Dialog

The CCP does not support moving a connection or dialog to a session on a different physical platform.

<createcall>

Referencing a dialog that has been prepared but not started in the `joinid` attribute of `<createcall>` always results in an error, and thus an `error.conference.join` event is not supported.

If call legs are being joined implicitly with the `<createcall>` tag, and the call legs do not provide their codec capabilities either in the initial `INVITE` to CCP or in the `200 OK` response to the initial `INVITE` sent by CCP, the CCP will not use the transcoding feature even if a transcoding server has been defined in the `mediacontroller.bridge_server` option. This is because the CCP requires knowledge of the codec capabilities of each call leg before creating bridges so that it can determine whether a bridging server is needed.

The workaround is to connect each call leg separately and join them together after they are in the connected state.

dialogid Property

If the connection is bridged to two or more dialogs, then the `dialogid` property contains the ID of the dialog that is sending media to the connection. If none of the dialogs are sending media to the connection, the property containing the ID of any one of the bridged dialogs is not supported.

Appendix

B Early Media

This appendix describes Early Media and how GVP supports it. It contains the following sections:

Background

Early Media is the concept of delivering a media stream prior to a call being answered.

In terms of SIP, after a call is in the progress of being setup after an `INVITE` message, media is transmitted prior to the `200 OK` response being generated.

Early Media has many uses, for example:

- Delivery of inband call progress messages, such as announcements.
- Customized ringing tones.
- Ability to avoid media clipping—Media clipping occurs when the user believes that the media session has already been established and begins speaking, but the establishment process has not finished yet, and thus leads to the loss of the first few syllables/words. Early Media helps to avoid such an issue by establishing the media path early.

Announcement Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml xmlns="http://www.w3.org/2002/09/ccxml" version="1.0">
  <!-- Create our ccxml level vars -->
  <var name="in_connectionid" expr="''" />
  <var name="dialogid" expr="''" />
  <var name="timer" expr="''"/>
  <!-- Set our initial state -->
  <var name="currentstate" expr="'state1'" />
  <eventprocessor statevariable="currentstate">
    <!-- Deal with the incoming call -->
    <transition state="state1" event="connection.alerting" name="evt">
      <assign name="in_connectionid" expr="evt.connectionid" />
   <dialogprepare
        src="'file:///usr/local/phoneweb/samples/helloaudio.vxml'"
        dialogid="dialogid"
        connectionid="in_connectionid"/>
      <assign name="currentstate" expr="'state2'"/>
    </transition>
    <transition state="state2" event="dialog.prepared" name="evt">
      <log expr="'Dialog has been prepared'"/>
      <dialogstart prepareddialogid="dialogid"/>
    </transition>
    <transition state="state2" event="send.successful" name="evt">
       <log expr="'send successful'"/>
    </transition>
    <transition state="state2" event="dialog.started" name="evt">
      <log expr="'Dialog has started'"/>
      <send target="in_connectionid" targettype="'x-connection'"
   data="'connection.progressing'"/>
    </transition>
    <transition state="state2" event="dialog.exit" name="evt">
      <log expr="'Dialog has terminated; accepting connection'"/>
      <accept connectionid="in_connectionid" />
      <assign name="currentstate" expr="'state3'"/>
    </transition>
    <transition event="connection.disconnected" name="evt">
      <exit/>
    </transition>
  </eventprocessor>
</ccxml>
```

Remarks

The preceding section is an example of CCXML in which a dialog (`helloaudio.vxml`, a simple VoiceXML page that plays only an audio clip) is established for an incoming call. The call is not accepted until the dialog finishes (that is, `helloaudio.vxml` finishes playing the audio file and terminates), and the call accepting action is handled by the `dialog.exit` event.

The key logic in this application is the line:

```
      <send target="in_connectionid" targettype="'x-connection'"
   data="'connection.progressing'"/>
```

When this is sent to a connection that is in the alerting state, it triggers the CCP to send a `183 Session Progress` message to the call originating side, with a valid SDP component so that a media path can be successfully established. Because the dialog was prepared with `connectionid` set to `in_connectionid`, this allows the media path to be established.

This simple example illustrates how to write an application so that it makes use of the Early Media capability. Advanced CCXML users can modify the preceding to simulate a ringback tone application by doing the following:

1. Use the `<createcall>` tag to create an outbound call.
2. Replace the simple `helloaudio.vxml` with a more sophisticated VoiceXML application, such as one that repeats an audio clip until it is interrupted.
3. Terminate the dialog when the outbound call is connected.
4. Connect the inbound call and then join the two calls together using `<join>` or `<merge>` (whichever is appropriate).

Note: The use of `<createcall>`, `<join>`, `<merge>` and other CCXML elements is outside the scope of this appendix.

Appendix

C MSML Specification

This appendix describes the GVP support for MSML features (RFC 5707). The standard MSML attributes are listed in this appendix, but the descriptions are not provided. These descriptions can be found in the specification at `http://tools.ietf.org/rfc/rfc5707.txt`.

CCXML only supports a subset of the full MSML specification in `<dialogstart/dialogprepare>`. Specifically, CCXML supports Dialog Core, Dialog Base, and Dialog CPA packages only.

This appendix contains the following sections:

Core Package

<msml>

Attributes

`version`

<send>

Attributes

`event`

`target`—The target must be part of the MSML session associated with the request, following the syntax:

conn:connID/dialog:dialogID[/primitive[.primitiveID]]

`valuelist`

`mark`

<result>

Attributes

`response`

`mark`

<event>

Attributes

`name`

`id`

Child Elements

`<name>`

No attributes.

`<value>`

No attributes

Dialog Core Package

<dialogstart>

Attributes

`target`

`src`

`type`

`name`

`mark`

`gvp:confrole`—The valid values are:

`regular` (default)—Customer call leg receives audio from the mixer, and video from the agent/student call leg (or from the file in a Push Video scenario)

`agent/student`—Agent or student call leg receives audio from the mixer, and video from the regular call leg (or from the file in a Push Video scenario) `coach`—Supervisor call leg in a Whisper Coaching scenario receives audio from the mixer, and the same video stream as the agent/student leg.

`monitor`—Supervisor or recording device call leg in a Silent Call Monitoring scenario receives audio from the mixer, and the same video stream as the agent/student leg.

`push`—Media playback device call leg does not receive any media; incoming audio stream is pushed to the mixer, and video stream is pushed to a regular or customer call leg.

`push-all`—Media playback device call leg provides audio to the mixer, and video stream to all call legs in the conference call.

Child Elements

See “Dialog Base Package” on page 54 for details about these elements.

`<play>`

`<dtmfgen>`

`<record>`

`<collect>`

`<cpd>`

<dialogend>

Attributes

`id`

`mark`

<send>

Attributes

`event`

`target`

<exit>

Attributes

`namelist`

<disconnect>

Attributes

`namelist`

<dialogprepare>

The `<dialogprepare>` element is supported as an extension of the MSML dialog core package. It is equivalent to the `<dialogstart>` element except that the dialog does not start until the `start` event is received. When a `start` event is sent to the preparing dialog, the dialog will be joined to its target and starts execution. The `<dialogprepare>` element is only supported for VoiceXML dialogs.

Dialog Base Package

<play>

Attributes

`id`

`iterate`

`maxtime`—This is supported in a single prompt or multiple prompts with iterate equal to 1.

`barge` (optional)—Defaults to `false`.

`cleardb` (optional)—Defaults to `false`.

`offset`—This is supported in a single prompt only, with iterate equal to 1.

`gvp:precheck` (optional)—The valid values are `true` or `false;` defaults to `false.` When this attribute is set to `true` and the audio or video prompt file is not found, GVP replies with `file not found.` When this attribute is set to `false,` there is no pre-check of the availability of the files. If the file cannot be found at prompt play time, the play element will end and the `play.end` shadow variable sets to `error.`

Events

`terminate`

Shadow Variables

`play.amt`

`play.end`—The possible values are:

- `play.complete`
- `play.complete.barge`
- `play.terminated`
- `play.timelimit`
- `play.error`
- `play.killsession`
- `play.unknown`

Child Elements

`<audio>`

Attributes

`uri`

`format`

`iterate`

`<video>`

Attributes

`uri`

`format`

`iterate`

`<playexit>`

No Attributes.

<dtmfgen>

Attributes

`id`

`digits`

`dur`

`interval`

Events

`terminate`

Shadow Variables

`dtmfgen[.id-if-specified].end`—The possible values are:

dtmfgen.complete

dtmfgen.terminated

dtmfgen.error

dtmfgen.killsession

dtmfgen.unknown

Child Elements

`<dtmfgenexit>`

No Attributes.

Child Elements

`<send>`

<record>

Attributes

`id`

`dest`

`format`

`profile`

`level`

`maxtime`

`prespeech`

`postspeech`

`termkey`

Events

`terminate`

Shadow Variables

`record.len`

`record.end`—The possible values are:

- `record.failed.prespeech`
- `record.complete.maxlength`
- `record.complete.postspeech`
- `record.complete.termkey`
- `record.complete.sizelimit`
- `record.error`
- `record.terminated`
- `record.killsession`
- `record.complete.unknown`

`record.recordid`

`record.size`

Child Elements

`<play>`

`<recordexit>`

No Attributes.

<collect>

Attributes

`id`

`cleardb`

`iterate`

Events

`terminate`

Shadow Variables

`dtmf.digits`

`dtmf.len`

`dtmf.last`

`dtmf.end`

Child Elements

`<play>`

`<pattern>`

Attributes

`digits`—The supported format is `max=n` where `n` is a decimal number specifying the number of digits to collect.

`format`—The only supported value is `moml+digits`.

`iterate`

Child Elements

`<send>`

`<dtmfexit>`

No Attributes

Child Elements

`<send>`

Dialog Call Progress Analysis Package

<cpd>

The CPD primitive supports three states of detection, and one non-detection state.

- `preconnect`—Detects pre-connect events.
- `postconnect`—Detects post-connect events.
- `beepdetect`—Detects answering machine beep.
- `buffer`—Does not detect events, but buffers a configurable amount of audio to be used after transitioning to a postconnect state.

Events can be sent to the CPD primitive to change the detection state. The CPD primitive automatically changes from the `postconnect` state to the `beepdetect` state if it detects an answering machine while in the `postconnect` state. The CPD primitive completes when it detects one of the following terminating results:

- `cpd.sit.nocircuit`
- `cpd.sit.reorder`
- `cpd.sit.operationintercept`
- `cpd.sit.cacantcircuit`
- `cpd.sit.custom1 (2,3,4)`
- `cpd.busy`
- `cpd.human`
- `cpd.fax`
- `cpd.beep`

The primitive will execute `<cpddetect>` (if present), and then execute `<cpdexit>`.

Attributes

beeptimeout (optional)—Defines the amount of time, in seconds, for CPD to timeout in the beepdetect state. When the timeout elapses, the child element, <beeptimeout>, is executed. This timeout only applies when the primitive is in the beepdetect state; this timeout is implicitly cancelled when the state changes to another state. When this attribute is not set, beeptimeout defaults to the [msml].cpd.beeptimeout configuration parameter.

connectnosignal (optional)—The valid value is true or false (default). When set to true, and in the preconnect state, the CPD element automatically transitions to the postconnect state when a call is determined to be connected. Otherwise, the CPD element remains in the preconnect state until told otherwise, or the preconnecttimeout event occurs.

id—When sending an event to the CPD element, use the following address: cpd[.id=if-specified].

initial(optional)—Defines the initial detection state. The valid values are:

Preconnect (default)

Postconnect

Beepdetect

postconnectpref (optional)—Defines how postconnect CPD detection prioritizes which results are detected. The valid values are:

default (default)—AM detection is performed as configured by default.

machine—AM detection is performed with the highest probability of answering machine detection.

no_machine—When SIT tones and FAX have not been detected, the connected call is considered answered by a live voice.

voice—AM detection will be performed with the highest probability of live voice detection.

postconnecttimeout (optional)—Defines the amount of time, in seconds, for CPD to timeout in the postconnect state. When the timeout elapses, the child element <cpdsilence> is executed. This timeout only applies when the primitive is in the postconnect state; the timeout is implicitly cancelled when the state changes to another state. If this attribute is not set, it defaults to the [msml].cpd.postconnecttimeout configuration parameter.

record (optional)—Defines whether received media during CPA is recorded. The valid values are:

true—recording will take place

false (default)—recording will not take place

Recordings will be recorded to the directory provided as the value of the [msml].cpd.record.basepath configuration parameter. The format type and file extension will be determined by the value of the [msml].cpd.record.fileext configuration parameter. The name of the recording will be generated at random.

Events

beepdetection—Sets the CPD primitive states to beep detection.

postconnect—Sets the CPD primitive state to postconnect.

terminate—Terminates the <cpd> element.

Shadow Variables

cpd[.id-if-specified].result—String value that specifies the result of CPD. The possible values are:

cpd.sit.nocircuit

cpd.sit.reoder

cpd.sit.operatorintercept

cpd.sit.vacantcircuit

cpd.sit.custom1

cpd.sit.custom2

cpd.sit.custom3

cpd.sit.custom4

cpd.busy

cpd.connect

cpd.human

cpd.fax

cpd.machine

cpd.beep

cpd.preconnect_timeout

cpd.silence

cpd.beeptimeout

cpd[.id-if-specified].result—String value that specifies the reason for terminating the <cpd> element. The possible values are:

cpd.terminated

cpd.completed

cpd.failed

cpd.recordfailed

cpd[.id-if-specified].recfile—Contains the path to the CPA recording. If there is no recording, the value is undefined.

Child Elements

<beeptimeout>—Executed when the beeptimeout is elapsed, meaning CPD did not detect any answering machine beep within the timeout period. The primitive is completed after this element and executes the <cpdexit> element.

No Attributes

<cpddetect>—Executed when a call progress event is detected.

No Attributes

`<cpdexit>`—Invoked when the CPD is completed or is terminated as a result of receiving the terminate event.

No Attributes

`<cpdsilence>`—Executed when the `postconnect` timeout is elapsed, meaning that there is silence on the media stream. The primitive is completed after this element and executes the `<cpdexit>` element.

No Attributes

`<cpdtimeout>`—Executed when the `preconnect` timeout is elapsed. The primitive is completed after this element and executes the `<cpdexit>` element.

No Attributes

Example

The following example initiates the CPD detection at the `preconnect` state, with a five second timeout period.

```
<?xml version="1.0" encoding="UTF-8"?>
<msml version="1.1">
   <dialogstart target="conn:xxxx" name="cpd" type="application/moml+xml">
      <cpd initial="preconnect" preconnecttimeout="5s">
          <cpdtimeout>
            <send target="source" event="done" namelist="cpd.recfile cpd.end cpd.result"/>
         </cpdtimeout>
   </cpd>
   </dialogstart>
</msml>
```

If no media activity was detected during the `preconnect` state, after five seconds, the CPD completes and sends an event with shadow variables. The actual result should appear as the following, in `info.content` for the `dialog.user.msml` event:

```
<?xml version="1.0"?>
<msml version="1.1">
  <event name="msml.dialog.exit" id="conn:xxx/dialog:yyy">
    <name>cpd.recfile</name>
    <value>undefined</value>
    <name>cpd.end</name>
    <value>cpd.completed</value>
    <name>cpd.result</name>
    </value>cpd.preconnect_timeout</value>
  </event>
</msml>
```

CCXML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml xmlns="http://www.w3.org/2002/09/ccxml" version="1.0">

  <!-- Test: SIM_7b: testing dialog.user.msml -->

  <var name="in_connectionid" expr="''" />
  <var name="dialogid" expr="''" />
  <var name="currentstate" expr="'state1'" />
  <var name="mediatypes" expr="'audio'" />
  <script src ="cpd.js"/>
  <eventprocessor statevariable="currentstate">
    <!-- Deal with the incoming call -->
    <transition state="state1" event="connection.alerting" name="evt">
      <assign name="in_connectionid" expr="evt.connectionid" />
      <accept connectionid="in_connectionid" />
      <assign name="currentstate" expr="'state2'"/>
    </transition>
    <transition state="state2" event="connection.connected" name="evt">
      <assign name="currentstate" expr="'state3'"/>
     <dialogprepare
        src="'.'"
        type="'application/vnd.radisys.msml+xml'"
        dialogid="dialogid" connectionid="in_connectionid"
        xmlns="urn:ietf:params:xml:ns:msml">
        <msml version="1.1">
        <dialogstart name="cpd" type="application/moml+xml">
        <cpd initial="preconnect" preconnecttimeout="5s">
        <cpdtimeout>
        <send target="source" event="done" namelist="cpd.recfile cpd.end
        cpd.result"/>
        </cpdtimeout>
        </cpd>
        </dialogstart>
        </msml>
     </dialogprepare>
    </transition>
    <transition state="state3" event="dialog.prepared">
      <assign name="currentstate" expr="'state4'"/>
       <dialogstart prepareddialogid="dialogid" connectionid="in_connectionid"/>
    </transition>
    <transition state="state4" event="dialog.started">
      <assign name="currentstate" expr="'state5'"/>
    </transition>
   <transition state="state5" event="dialog.user.msml">
      <log expr="' content:' + event$.info.content"/>
       <if cond="event$.info.contenttype == 'application/vnd.radisys.msml+xml'">
        <script>
        var cpdresult = parseCPD(event$.info.content);
        </script>
        <log expr="'#PASSED#'"/>
        <log expr="'result : cpd.recfile:'"/> <log expr="cpdresult.recfile"/>
        <log expr ="'cpd.end:'"/> <log expr="cpdresult.end"/>
        <log expr ="'cpd.result:'"/> <log expr="cpdresult.result"/>
        <exit/>
       <else/>
         <log expr="'#FAIL# Incorrent contenttype:' + event$.info.contenttype"/>
       </if>
   </transition>
    <transition state="state5" event="dialog.exit" name="evt">
      <assign name="currentstate" expr="'state6'"/>
      <exit/>
    </transition>
    <transition event="fetch.done">
    </transition>
    <transition event="ccxml.loaded">
    </transition>
    <transition event="*">
        <log expr="'#FAILED# SIM_7b'"/>
        <exit/>
    </transition>
  </eventprocessor>
</ccxml>
```

A line of code early in the above example (`<script src ="cpd.js"/>`) refers to the Javascript file `cpd.js` (see below), which performs a single function: parsing the cpd file.

cpd.js

```
function parseCPD(cpdstring)
{
   var result = new Object();
   result.error = 0;
   var nameOpen = 6;
   var valueOpen = 7;

   if(cpdstring.indexOf("<?xml")!=0)
   {
      result.error = 1;
      return result;
   }
   var end, begin;
   end = cpdstring.indexOf("?>");
   if(end <= 0)
   {
     result.error = 2;
     return result;
   }
   begin = cpdstring.indexOf("<msml");
   if(begin < 0 || begin < end)
```

```
{
   result.error = 3;
   return result;
}
end = cpdstring.substring(begin).indexOf("</msml>");
if(end < 0 || end < begin)
{
   result.error = 4;
   return result;
}
begin = cpdstring.substring(begin).indexOf("<event");
if(begin < 0 || begin > end)
{
   result.error = 5;
   return result;
}
end = cpdstring.substring(begin).indexOf("</event>");
if(end < 0 )
{
   result.error = 6;
   return result;
}
var beginpair = cpdstring.substring(begin).indexOf("<name");
if(beginpair < 0 || beginpair > end)
{
result.error = 7;
   return result;
}
var namevalue=cpdstring.substring(begin+beginpair,end+begin);
result.name = namevalue;

var pairs = namevalue.split("</value>");
if (pairs.length <= 0)
{
   result.error = 8;
   return result;
}
for (var i=0; i<pairs.length; i++)
 {
   begin = pairs[i].indexOf("<name>");
   end = pairs[i].indexOf("</name>");
   if(end <0 )
   {
      result.error = pairs[i];
      return result;
   }
   if( begin <0 )
   {
      result.error = i+11;
      return result;
   }
```

```
      if( end < begin)
      {
         result.error = i+12;
         return result;
      }
      var name = pairs[i].substring(begin+nameOpen,end);
      begin = pairs[i].indexOf("<value>");
      if (begin < 0 || begin < end)
      {
         result.error = i+13;
         return result;
      }
      var value = pairs[i].substring(begin+valueOpen);

      if(name == "cpd.recfile")
         result.recfile = value.toString();
      else if (name=="cpd.end")
         result.end = value.toString();
      else if (name=="cpd.result")
         result.result = value.toString();
      else
         result.error = "wrong cpd Element";
   }
   result.error = "0";
    return result;
}
```

Supplements

Related Documentation Resources

The following resources provide additional information that is relevant to this software. Consult these additional resources as necessary.

Management Framework

- *Framework 8.1 Deployment Guide,* which provides information about configuring, installing, starting, and stopping Framework components.
- *Framework 8.1 Genesys Administrator Deployment Guide,* which provides information about installing and configuring Genesys Administrator.
- *Framework 8.1 Genesys Administrator Help,* which provides information about configuring and provisioning contact center objects by using the Genesys Administrator.
- *Framework 8.1 Configuration Options Reference Manual,* which provides descriptions of the configuration options for Framework components.

SIP Server

- *Framework 8.1 SIP Server Deployment Guide,* which provides information about configuring and installing SIP Server.

Genesys Voice Platform

- *Genesys Voice Platform 8.1 Deployment Guide,* which provides information about installing and configuring Genesys Voice Platform (GVP).
- *Genesys Voice Platform 8.1 User's Guide,* which provides information about configuring, provisioning, and monitoring GVP and its components.
- *Genesys Voice Platform 8.1 Troubleshooting Guide,* which provides troubleshooting methodology, basic troubleshooting information, and troubleshooting tools.

- *Genesys Voice Platform 8.1 SNMP and MIB Reference,* which provides information about all of the Simple Network Management Protocol (SNMP) Management Information Bases (MIBs) and traps for GVP, including descriptions and user actions.
- *Genesys Voice Platform 8.1 Genesys VoiceXML 2.1 Reference Help,* which provides information about developing Voice Extensible Markup Language (VoiceXML) applications. It presents VoiceXML concepts, and provides examples that focus on the GVP Next Generation Interpreter (NGI) implementation of VoiceXML.
- *Genesys Voice Platform 8.1 Legacy Genesys VoiceXML 2.1 Reference Manual*, which describes the VoiceXML 2.1 language as implemented by the Legacy GVP Interpreter (GVPi) in GVP 7.6 and earlier, and which is now supported in the GVP 8.1 release.
- *Genesys Voice Platform 8.1 Application Migration Guide,* which provides detailed information about the application modifications that are required to use legacy GVP 7.6 voice and call-control applications in GVP 8.1.
- *Genesys Voice Platform 8.1 Configuration Options Reference,* which replicates the metadata available in the Genesys provisioning GUI, to provide information about all the GVP configuration options, including descriptions, syntax, valid values, and default values.
- *Genesys Voice Platform 8.1 Metrics Reference,* which provides information about all the GVP metrics (VoiceXML and CCXML application event logs), including descriptions, format, logging level, source component, and metric ID.

Voice Platform Solution

- *Voice Platform Solution 8.1 Integration Guide,* which provides information about integrating GVP, SIP Server, and, if applicable, IVR Server.

Composer Voice

- *Composer 8.1 Deployment Guide,* which provides installation and configuration instructions for Composer.
- *Composer 8.1 Help,* which provides online information about using Composer, an Integrated Development Environment used to develop applications for GVP and Universal Routing.

Open Standards

- *W3C Voice Extensible Markup Language (VoiceXML) 2.1, W3C Recommendation 19 June 2007,* which is the World Wide Web Consortium (W3C) VoiceXML specification that GVP NGI supports.

- *W3C Voice Extensible Markup Language (VoiceXML) 2.0, W3C Recommendation 16 March 2004,* which is the W3C VoiceXML specification that GVP supports.
- *W3C Speech Synthesis Markup Language (SSML) Version 1.0, Recommendation 7 September 2004,* which is the W3C SSML specification that GVP supports.
- *W3C Voice Browser Call Control: CCXML Version 1.0, W3C Working Draft 29 June 2005,* which is the W3C CCXML specification that GVP supports.
- *W3C Semantic Interpretation for Speech Recognition (SISR) Version 1.0, W3C Recommendation 5 April 2007,* which is the W3C SISR specification that GVP supports.
- *W3C Speech Recognition Grammar Specification (SRGS) Version 1.0, W3C Recommendation 16 March 2004,* which is the W3C SRGS specification that GVP supports.

Genesys

- *Genesys Technical Publications Glossary,* which ships on the Genesys Documentation Library DVD and which provides a comprehensive list of the Genesys and computer-telephony integration (CTI) terminology and acronyms used in this document.
- *Genesys Migration Guide*, which ships on the Genesys Documentation Library DVD, and which provides documented migration strategies for Genesys product releases. Contact Genesys Technical Support for more information.
- Release Notes and Product Advisories for this product, which are available on the Genesys Technical Support website at `http://genesyslab.com/support`.

Information about supported hardware and third-party software is available on the Genesys Technical Support website in the following documents:

- *Genesys Supported Operating Environment Reference Manual*
- *Genesys Supported Media Interfaces Reference Manual*

For additional system-wide planning tools and information, see the release-specific listings of System Level Documents on the Genesys Technical Support website, accessible from the `system level documents by release` tab in the Knowledge Base `Browse Documents` Section.

Genesys product documentation is available on the:

- Genesys Technical Support website at `http://genesyslab.com/support`.
- Genesys Documentation Library DVD, which you can order by e-mail from Genesys Order Management at `orderman@genesyslab.com`.

Document Conventions

This document uses certain stylistic and typographical conventions—introduced here—that serve as shorthands for particular kinds of information.

Document Version Number

A version number appears at the bottom of the inside front cover of this document. Version numbers change as new information is added to this document. Here is a sample version number:

80fr_ref_06-2008_v8.0.001.00

You will need this number when you are talking with Genesys Technical Support about this product.

Screen Captures Used in This Document

Screen captures from the product graphical user interface (GUI), as used in this document, may sometimes contain minor spelling, capitalization, or grammatical errors. The text accompanying and explaining the screen captures corrects such errors *except* when such a correction would prevent you from installing, configuring, or successfully using the product. For example, if the name of an option contains a usage error, the name would be presented exactly as it appears in the product GUI; the error would not be corrected in any accompanying text.

Type Styles

Table 9 describes and illustrates the type conventions that are used in this document.

Table 9: Type Styles

Type Style	Used For	Examples
Italic	• Document titles • Emphasis • Definitions of (or first references to) unfamiliar terms • Mathematical variables Also used to indicate placeholder text within code samples or commands, in the special case where angle brackets are a required part of the syntax (see the note about angle brackets on page 69).	Please consult the *Genesys Migration Guide* for more information. Do *not* use this value for this option. A *customary and usual* practice is one that is widely accepted and used within a particular industry or profession. The formula, $x + 1 = 7$ where x stands for . . .

Table 9: Type Styles (Continued)

Type Style	Used For	Examples
Monospace font (Looks like `teletype` or `typewriter text`)	All programming identifiers and GUI elements. This convention includes: • The *names* of directories, files, folders, configuration objects, paths, scripts, dialog boxes, options, fields, text and list boxes, operational modes, all buttons (including radio buttons), check boxes, commands, tabs, CTI events, and error messages. • The values of options. • Logical arguments and command syntax. • Code samples. Also used for any text that users must manually enter during a configuration or installation procedure, or on a command line.	Select the `Show variables on screen` check box. In the `Operand` text box, enter your formula. Click `OK` to exit the `Properties` dialog box. T-Server distributes the error messages in `EventError` events. If you select `true` for the `inbound-bsns-calls` option, all established inbound calls on a local agent are considered business calls. Enter `exit` on the command line.
Square brackets ([ ])	A particular parameter or value that is optional within a logical argument, a command, or some programming syntax. That is, the presence of the parameter or value is not required to resolve the argument, command, or block of code. The user decides whether to include this optional information.	`smcp_server -host [/flags]`
Angle brackets (< >)	A placeholder for a value that the user must specify. This might be a DN or a port number specific to your enterprise. **Note:** In some cases, angle brackets are required characters in code syntax (for example, in XML schemas). In these cases, italic text is used for placeholder values.	`smcp_server -host <confighost>`

Index