Cisco Unified CVP VoiceXML Components

Some VXML Server components require detailed description on how to use them properly, especially when their functionality requires or is extended by programming. You might be able to create a voice application entirely dependent on fixed data, but most dynamic applications require some programming.

The non-developer user needs to be aware of these components and the functions they serve. The application designer needa to understand in what situations various components are required so that a comprehensive specification can be given to a developer responsible for building these components.

This chapter describes these components in more detail, and the typical situations where they would be used. It also describes the Unified CVP concepts used to develop and use the components. The Programming Guide for Cisco Unified CVP VXML Server and Unified Call Studio describes the components that require programming and describes the process of constructing and deploying them. The *Programming Guide for Cisco Unified CVP VXML Server and Cisco Unified Call Studio* provides a comprehensive description of the information that this chapter introduces.

Components

The following components are described in this chapter:

- **Built With Programming**—These components require some programming effort.
- **Call-Specific**—These components are built to be used within individual calls.

- **Custom Configurable Elements**—A developer might want to create their own reusable, configurable elements to supplement the elements that Unified CVP provides.
- **Standard Action and Decision Elements**—For situations where unique, application-specific functionality is needed, which does not require the flexibility and complexity of configurable elements.
- **Dynamic Element Configurations**—For situations where the configuration for a configurable element can only be determined at runtime.
- **Start and End of Call Action**—To perform tasks before each call begins and after each call ends.
- **Hotevents**—To specify the VoiceXML to run when a certain VoiceXML event occurs.
- **Say It Smart Plugins**—To play back additional formatted data or to extend existing Say It Smart behavior.

- **VXML Server-Specific**—These components are built to run on VXML Server as a whole and do not apply to a specific call.
  - **Start and End of Application Actions**—To perform tasks when a Unified CVP voice application is loaded and shuts down.
  - **Loggers**—Plug-ins designed to listen to events that occur within calls to an application and log or report them.
  - **On Error Notification**—To perform tasks if an error causes the phone call to end prematurely.
- **Built Without Programming**—These components do not require high-level programming effort to construct.
  - **XML Decisions**—Unified CVP provides an XML format for writing simple decisions without programming. The exact XML format is detailed in this chapter.
  - **VoiceXML Insert Elements**—This element is used in situations where the developer wants to incorporate custom VoiceXML content into a Unified CVP application. This chapter provides guidelines for building a VoiceXML insert element.

Variables

Unified CVP offers variables as a method for components to share data with each other, in these four forms: global data, application data, session data and element data.

Global Data

A global data variable is globally accessible and modifiable from all calls to all applications. Global data is given a single namespace within VXML Server that is shared across all calls to all applications. If a component changes global data, that change is immediately available to all calls for all applications. Global data can hold any data, including a Java object. The lifetime of global data is the lifetime of VXML Server. Global data is reset if the application server is restarted or the VXML Server web application archive (WAR) is restarted.

Global data typically is used to store static information that needs to be available to all components, no matter which application they reside in. For example, the holiday schedule of a company that applies to all applications for that company.

Application Data

An application data variable is accessible and modifiable from all calls to a particular application. Application data variables from one application cannot be seen by components in another application. Each application is given its own namespace to store application data. If a component changes application data, that change is immediately available to all other calls to the application. Application data can hold any data, including a Java object. The lifetime of application data is the lifetime of the application. Application data would be reset if the application were updated and would be deleted if the application were released.

Application data is typically used to store application-specific information that does not change on a per call basis and is to be available to all calls (for example, the location of a database to use for the application).

Session Data

Session data variables are accessible and modifiable from a single call session. Session data variables in one call cannot be accessed by components handling another call. Each session has its own session data namespace; session data set by one component will overwrite existing session data that has the same name. Session data can hold any data, including a Java data structure. The lifetime of session data is the lifetime of the session or the call. When the call ends, the session data is deleted.

Any component accessed within a call session, including elements, can create, modify, and delete session data. Session data can be created automatically by the system in two ways:

- If the voice browser passes additional arguments to VXML Server when the call is first received, these additional arguments will be added as session data with the arguments' name or value pairs translated to the session data name and value (both as `String` types). For example, if the voice browser calls the URL:

  ```
  http://myserver.com/CVP/Server?audium_application=MyApp&SomeData=1234
  ```

  This session will create session data named SomeData with a value of 1234 in every call session of the application MyApp that starts through this URL.

- If a Unified CVP voice application performs an application transfer to another application and the developer has chosen to pass data from the source application to the destination application, then this data will appear as session data in the destination application (the data is renamed before it is passed to the destination application). Refer to the Call Studio documentation for more information on application transfers.

Element Data

Element data variables are accessible from a single call session and modifiable from a single element within that call session. As the name suggests, element data can only be created by elements (excluding start and end of call events, the global error handler, hotevents, and XML decisions). Dynamic configurations are technically part of an element since they are responsible for configuring an element, so they can also create element data. Only the element that created an element data variable can modify or delete it, though it can be read by all other components. Due to the fact that the variable belongs to the element, the variable namespace is contained within the element, meaning two elements can define element data with the same name without

interfering with each other. To uniquely identify an element data variable, both the name of the element and the name of the variable must be used. Like session data, the lifetime of session data is the lifetime of the session or the call. When the call ends, the element data is deleted.

Component Accessibility

The following table lists each component and its ability to get and set global, application, session, and element data.

	Global Data		Application Data		Session Data		Element Data	
Component	**Get**	**Set**	**Get**	**Set**	**Get**	**Set**	**Get**	**Set**
Configurable Elements	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Standard Elements	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic Configurations	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Start and End of Call Actions	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Hotevents	No	No	No	No	No	No	No	No
Say It Smart Plugins	No	No	No	No	No	No	No	No
Start and End of Applications Actions	Yes	Yes	Yes	Yes	No	No	No	No
Loggers	Yes	No	Yes	No	Yes	No	Yes	No
On Error Notification	No	No	No	No	Yes	No	No	No
XML Decisions	No	No	No	No	No	No	No	No
VoiceXML Insert Elements	No	No	No	No	Yes	Yes	Yes	Yes

Note Hotevents, which are VoiceXML code appearing in the root document, do not have access to any server-side information.

Note A Say It Smart plug-in's purpose is to convert a value into a list of audio files, so it does not need to access server-side information.

Note A Logger's only responsibility is to report or log data and has access to all variables types but cannot set them.

Note On Error Notification classes are given the session data that existed at the time the error occurred.

APIs

To facilitate the development of components requiring programming effort, Unified CVP provides two application programming interfaces (APIs) for developers to use. The first is a Java API. The second API involves the use of XML sent through HTTP, which allows components to be built using programming languages other than Java. Some more complex and tightly integrated components can be built only through the Java API, though in most other aspects, the two APIs are functionally identical. The APIs themselves and the process of building components using either API is fully detailed in the Javadocs published with the software and in the *Programming Guide for Cisco Unified CVP VXML Server and Cisco Unified Call Studio*. The two components that do not require the use of high-level programming, XML decisions, and VoiceXML insert elements are fully explained in this document.

The APIs are used to interface with VXML Server in order to retrieve data or change information. The API provided to each component has slightly different functionality reflecting each component's unique abilities. The following abilities are provided by the API that is common to most components used within a call flow:

- Getting call information such as the ANI, DNIS, call start time, application name, and so on.
- Getting or setting global data, application data, element data, or session data.
- Getting information about the application's settings such as the default audio path, voice browser, and so on.
- Setting the maintainer and default audio path. Changing the maintainer allows multiple people to maintain different parts of a single application. Changing the default audio path allows an application to change the persona or even language of the audio at any time during the call.
- Sending a custom event to all application loggers .

The following table shows which APIs can be used to construct the various components listed.

VXML Server Component	Build with Java API	Build Using XML-over-HTTP API	VoiceXML Knowledge Suggested
Configurable Action and Decision Elements	Yes	No	No
Configurable Voice Elements	Yes	No	Yes
Standard Elements	Yes	Yes	No
Dynamic Element Configurations	Yes	Yes	No
Start or End of Call Actions	Yes	Yes	No
Hotevents	Yes	No	Yes
Say It Smart Plugins	Yes	No	No

Start and End of Application Actions	Yes	No	No
Loggers	Yes	No	No
On Error Notification	Yes	No	No
XML Decisions	N/A	N/A	N/A
VoiceXML Insert Elements	N/A	N/A	Yes

Configurable Elements

Most of the elements in a typical Unified CVP application are prebuilt, reusable elements whose configurations are customized by the application designer. Using a configurable element in a call flow requires no programming or VoiceXML expertise, and because they can encapsulate a lot of functionality, the element greatly simplifies and speeds up the application building process. VXML Server includes dozens of elements that perform common tasks such as collecting a phone number or sending e-mail. A need may exist, however, for an element with functionality not available in the default installation. Additionally, while Unified CVP elements have been designed with configurations that are as flexible as possible, there may be situations where a desired configuration is not supported or is difficult to implement.

To address these issues, a developer can construct custom configurable elements that, once built, can be used and reused. The developer can design the element to possess as large or as small a configuration as desired, depending on how flexible it needs to be. Once deployed, custom elements appear in Builder for Call Studio in the Element Pane and are configured in the same way as Unified CVP Elements.

Due to the level of integration with the Unified CVP software required, only the Java API provides the means for building configurable elements. Using this API, configurable action, decision, and voice elements can be built. Because voice elements are responsible for producing VoiceXML, they use an additional Java API, the Voice Foundation Classes (VFCs). The VFCs are used to abstract the differences between the various voice browsers supported by Unified CVP. The VFCs follow a design that parallels VoiceXML itself, and only a developer familiar with VoiceXML and the process of a voice browser interpreting VoiceXML will be fully suited to use the VFCs to build voice elements.

The *Programming Guide for Cisco Unified CVP VXML Server and Cisco Unified Call Studio* describes the process of building configurable elements including detailing the VFC API for building voice elements.

Standard Action and Decision Elements

Unlike configurable action or decision elements, a standard action or decision element is designed more as a one-off as they satisfy an application-specific purpose. As a result, standard action and decision elements do not require configurations.

There are many situations where a programming effort is required to perform some task specific to an application. Because the task is very specialized, preexisting reusable elements are too general to perform the effort. There would not be an advantage to building a configurable element for this purpose because there is little chance it would be needed anywhere but in this application. The developer would use a standard action or decision element to perform just this task. If the task applies to multiple situations, the developer most likely would put in the extra effort to construct a configurable, reusable element.

Unified CVP provides a means of defining standard decision elements without programming by writing an XML document directly within Builder for Call Studio. Consider format should be investigated when you want simple or moderately complex standard decision elements, and be able to revert to the programming API if the built-in format proves to be insufficient. The XML format that the Builder for Call Studio user interface produces for standard decision elements is described later in this chapter.

Dynamic Element Configurations

Each configurable voice, action, and decision element used in an application must have a configuration. Usually, the configuration will be fixed and functions the same for every caller that visits it. In these situations, the designer using Builder for Call Studio creates this configuration in the Configuration Pane. This configuration is saved as an XML file when the application is deployed.

There are situations, when a configuration for an element depends on information known only at runtime; it is considered to be dynamic. An example would be to configure the Unified CVP audio voice element to play a greeting message depending on the time of the day. The application know only at runtime the exact calling time and then what greeting message to play.

To produce dynamic configurations, programming is required. Dynamic element configurations are responsible for taking a base configuration (a partial configuration created in the Builder for Call Studio), adding to it or changing it depending on the application business logic, and returning the desired element configuration to VXML Server.

Start / End of Call Actions

Unified CVP provides functions to run some code when a phone call is received for a particular application or when the call ends. The end of a call is defined as either a end of call by the caller, a disconnection by the system, a move from one Unified CVP application to another Unified CVP application, or other rarer ways for the call to end such as a blind transfer or session timeout.

The purpose of the start of call action typically is to set up dynamic information that is used throughout the call, for example, the current price of a stock or information about the caller identified by their ANI in some situations. The end of call action typically is used to export information about the call to external systems, perform call flow history traces, or run other tasks that require information on what occurred within the call.

The start of call action is given the special ability to change the voice browser of the call. This change applies to the current call only, and allows for a truly dynamic application. By allowing the voice browser to change, the application can be deployed on multiple voice browsers at once and use a simple DNIS check to output VoiceXML compatible with the appropriate browser. This task can only be done in the start of call action because the call technically has not started when this action occurs.

The end of call action is given the special ability to produce a final VoiceXML page to send to the browser. Even though the caller is no longer connected to the browser by the time the end of call action is run, some voice browsers will allow for the interpretation of a VoiceXML page sent back in response to a request triggered by a disconnect or hang-up event. Typically this page will perform final logging tasks on the browser.

Event Handling

Event Handlers

Events and exceptions occurring in a Call Studio application can be handled by event handlers defined in the applications. To configure Event Handler on an element, you must add an event handler to the element configuration. The following events types are supported by Call Studio:

- VXML Event
- Custom Exception
- Local HotEvent
- Java Exception

Event Handlers can be placed at several levels in a call flow:

- **Element Level**

  VXML Events and Java Exceptions encountered during the processing of an element can be caught at the element level.

- **Subflow Start Element Level**

  Event Handlers placed at Subflow Start element level will be active during the processing of that particular sub flow. Subflow Start level event handlers can be used to handle events that are not handled at the element level.

- **Subflow Call Element Level**

  Event Handlers attached to the Subflow Call element can be used to handle events that are not handled inside a sub flow.

- **Start of Call Element Level**

  Event handlers attached to the Start of Call element act as global event handlers for the application. Any event not handled at the levels described above can be handled at this level. All Hot Links defined in the application act as event handlers at the Start of Call element level. Events escaped from attached event handlers and Hot Events can be trapped by the Error element defined in the application.

  Events generated inside a call flow will be propagated through the Subflow hierarchy until the Start of Call element. Event handler at a lower level has precedence over the event handlers at a higher level. Properties of the event trapped by the event handler can be extracted from the session variable lastException.

Note In addition to VXML Event and Java Exception, event handlers can be configured for Custom Exception and Local Hot Link at the levels described above. However, all event types are not applicable for all elements. Refer to the Element Specifications for Cisco Unified CVP VXML Server and Cisco Unified Call Studio for more details.

Say It Smart Plugins

In VXML Server, developers can create their own Say It Smart plugins. Similar to custom elements, Say it Smart plugins are prebuilt Java classes that when deployed in the Builder for Call Studio can be used as a

new Say It Smart type. As with custom elements, the level of integration required with the Unified CVP software restricts the creation of Say It Smart plugins to the Java API.

Custom Say It Smart plug-ins can be constructed to read back formatted data not handled by Unified CVP Say It Smart plug-ins, such as spelling playback or reading the name of an airport from its three-digit code. Plug-ins can also be created to extend the functionality of existing plug-ins, such as adding new output formats to play the information in another language. For example, a plug-in can define a new output format for the Unified CVP Date Say It Smart plug-in that reads back dates in Spanish.

Refer to the Programming Guide for Cisco Unified CVP VXML Server and Unified Call Studio for a full description of the process of building custom Say It Smart plug-ins.

Start and End of Application Actions

Unified CVP provides functions to run some code when an application is launched or shut down. A start of application action is run when the VXML Server web application archive (WAR) starts up (which occurs when the application server first starts up or the application server reloads the WAR), or the application is updated. An end of application action is run when the application is updated, released, or the web application is shut down (which occurs if the application server reloads or shuts down the web application or the application server itself is shut down).

The start of application action typically sets up global data or application data that is accessed by components within the call flow. Because global and application data's lifetime is the lifetime of the application, and they can contain Java objects, the start of the application action can set up persistent database connections or other communications to external systems that remain connected while the application is running.

Note If an error occurs within the start of the application class, the application deployment will continue unchanged. The designer can specify that an error in a particular start of application class stop the application deployment, if the class performs mandatory tasks that are necessary for the application to run correctly.

The end of application action cleans up any data, database connections, and so on, that are no longer needed once the application is shut down.

Note The end of application action is called even when the application is updated because the update may have changed the data that is needed by the application.

Every application deployed on VXML Server has the ability to define any number of start and end of application actions that are run in the order in which they appear in the application settings.

Loggers

The act of logging information about callers to the system is performed by loggers. An application can reference any number of loggers that *listen* for logging events that occur. These events range from events triggered by a call, such as a caller entering an element or activating a hotlink to administration events such as an application being updated to errors that may have been encountered. Loggers can take the information on these events

and use them as they want. Typically the logger stores that information somewhere such as a log file, database, or reporting system.

VXML Server includes default loggers that store the information obtained from logging events to parseable text log files. A logger might be required with functionality not available in the default installation or a logger that takes the same data and stores it using a different method.

To address these issues, a developer can construct custom loggers that listen for logger events and report them in their own way. The developer can design the logger to use a configuration to customize how the logger functions, depending on how flexible it needs to be. Due to the level of integration with the Unified CVP software required, only the Java API provides the method for building loggers.

Refer to VXML Server Logging in the section entitled *Application Loggers* for descriptions of the loggers included with VXML Server. Refer to the Programming Guide for Cisco Unified CVP VXML Server and Unified Call Studio for a description of the process of building custom loggers.

For additional information on Unified CVP VXML Server Logs, see *Configuring Unified CVP Logging and Event Notification* in the Configuration and Administration Guide for Cisco Unified Customer Voice Portal.

On Error Notification

When errors occur on the VXML Server, the application-specific error voice element decides how to handle the caller. If specified, the on error notification Java class can be configured to be activated when an error occurs. The class is given information about the application and some basic call information to allow the developer to specify the action accordingly. The developer can write this class to perform any function.

The most common purpose for the on error notification class is to perform a custom notification, which indicates at runtime that an error occurred. This notification might involve paging an administrator or integrating with a third-party trouble ticket and notification process. Since the notification usually involves an administrator whose responsibility is the entire VXML Server, the Java class, once specified, applies to any error that prematurely ends a call on any Unified CVP application.

Note This class is used for notification purposes; it does not allow the call to recover from the error.

Note There is no XML API equivalent for the on error notification; if done at all, it must be written in Java.

Unified CVP XML Decisions in Detail

Many commercial applications with decisions driven by business logic use an external rules engine to codify the definition of rules. These rule engines help describe the definition of a rule and then manage the process of making decisions based on the current criteria. VXML Server bundles a rule engine in the standard installation and provides an XML data format for defining decision elements within the framework of a voice application. The XML format is simple enough for an application designer to enter within Builder for Call Studio without requiring separate programming resources.

The main feature of a rule is one or more expressions. An expression is a statement that evaluates to a true or false. In most cases, there are two parts (called terms) to an expression with an operator in between. The terms

are defined by VXML Server to represent all of the most common items necessary to base decisions on in a voice application such as telephony data, element or session data, times and dates, caller activity, user information, and so on. The operators depend on the data being compared. For example, numbers can be compared for equality or greater than or less than while strings can be compared for equality or if it contains something. One type of expression breaks this format: an *exists* expression that itself evaluates to a true or false and does not need anything to compare it to.

Examples:

- Has this caller called before?
- Does the system have a social security number for the user?

Each of these conditions checks for the existence of something that is itself a complete expression.

One or more of these expressions are combined to yield one exit state of the decision element. Multiple expressions can be combined using ands or ors, though not a combination. For example, if the ANI begins with 212 OR if the ANI begins with 646 then return the exit state ‘Manhattan’.

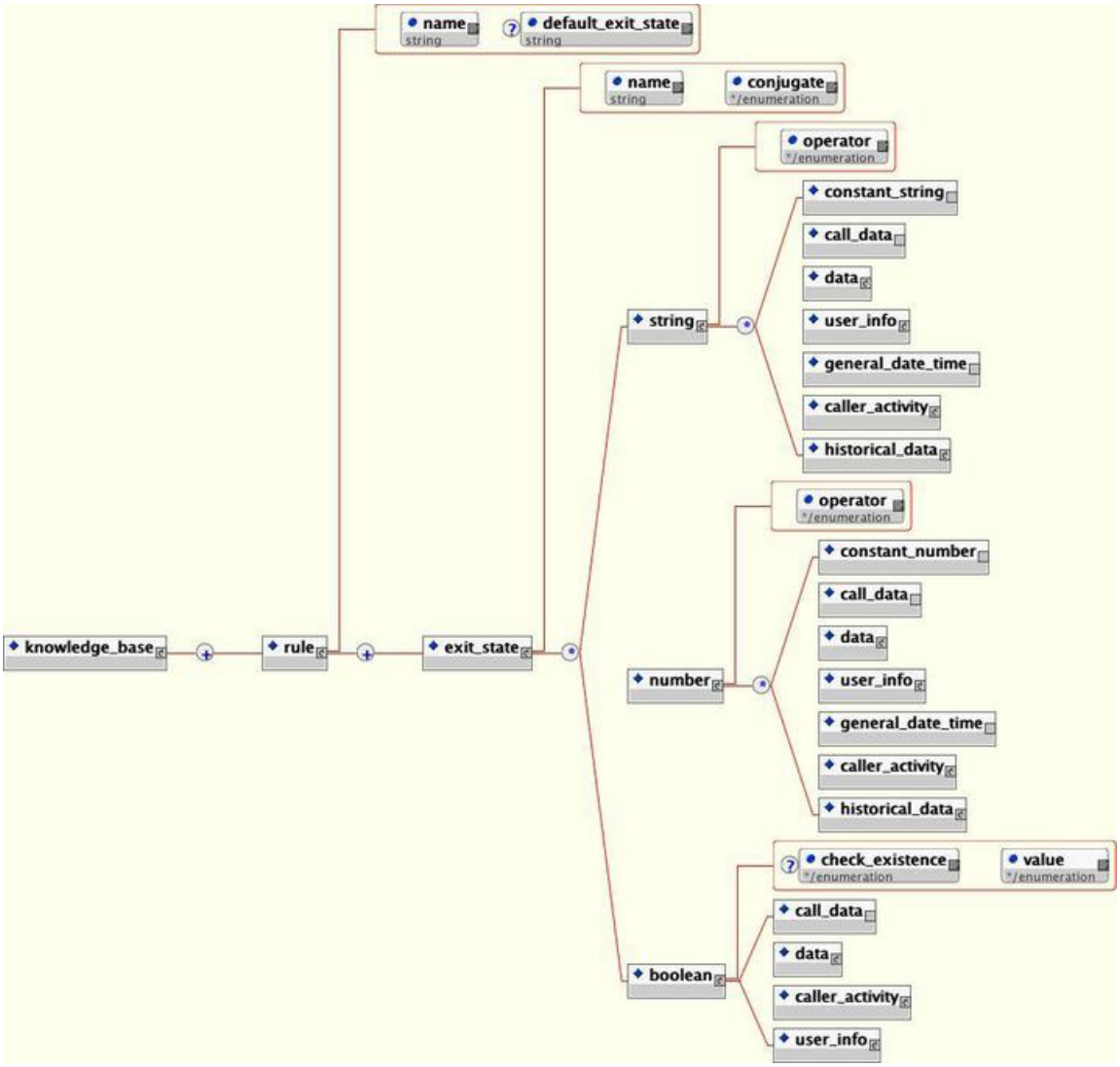
If a combination of ands and ors is desired, multiple expressions that return the same exit state would be used. For example, if the ANI begins with 212 and the user is a gold or platinum customer, then return the exit state ‘discount’ would **not** work as a single rule because the discount would be given to callers with a 212 area code who are gold customers and all platinum customers (there is no way to set precedence).

This would have to be expressed as two rules with the same exit state:

1. If the ANI begins with 212 AND the user is a gold customer, return the exit state discount.
2. If the ANI begins with 212 AND the user is a platinum customer, return the exit state discount.

It is possible to define an exit state that returns when all other exit states fail to apply, called the default exit state. When not specified, all possible cases must be caught by the defined rules. For example, if a rule checks if a number is greater than 5, there should be another rule checking if the number is less than or equal to 5, unless the default exit state is defined. One can even create a set of rules that start from being restrictive, searching for only very specific matches, to progressively less restrictive since the first rule to be true will yield an exit state and no more rules are tested.

Figure 1: Example of Tags for Defining a Decision



The knowledge_base example in the preceding figure shows the main tags of the XML file format for defining a decision. The elements in this XML document are:

- **rule**—This tag names the rule for the decision. There can only be one `<rule>` tag in the document. The tag contains any number of exit states that make up the decision. The optional `default_exit_state` attribute lists the exit state to return if no other exit states apply (essentially an else exit state).
- **exit_state**—This tag encapsulates the expressions that when true, return a particular exit state. The `name` attribute must refer to the same value chosen when the decision element was defined in the Builder for Call Studio. The `conjugate` attribute can be either *and* or *or*. If the exit state contains only one expression the conjugate attribute is ignored. The content of the `<exit_state>` tag is the type of data to be compared, each type containing different kinds of data. There can be any number of children to the `<exit_state>` tag, each representing another expression linked with the conjugate.
- **string**—This tag represents an expression comparing strings. The `operator` attribute can be *contains*, *not_contains*, *ends_with*, *not_ends_with*, *equal*, *not_equal*, *starts_with*, and *not_starts_with*. There can

be only two children to the <string> tag, representing the two terms of the expression. If there are fewer than two, an error will occur. If more, the extra ones are ignored. The content can be tags representing a constant string entered by the developer, data about the call, session and element data, user information, date and time information, the activity of the caller, and historical activity of the user. These tags are fully defined in the following sections.

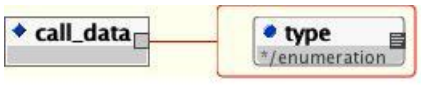
- **number**—This tag represents an expression comparing numbers. The operator attribute can be equal, not_equal, greater, greater_equal, less, and less_equal. There can be only two children to the <number> tag, representing the two terms of the expression. If there are fewer than two, an error will occur. If more, the extra ones will be ignored. The content can be tags representing a constant number entered by the developer, data about the call, session and element data, user information, date and time information, the activity of the caller, and historical activity of the user. These tags are fully defined in the following sections.
- **boolean**—This tag represents an expression which evaluates to a boolean result, requiring only a single term. If the check_existence attribute is yes, and the value attribute is true, it is checking if the data defined by the child tag exists. If check_existence is yes, and value is false, it is checking if the data defined by the child tag does *not* exist. If check_existence is no, the value attribute is used to compare the data defined by the child tag with either true or false. True means the expression is true if the data defined by the child tag evaluates to true. The child tags are a smaller subset of those allowed in <string> and <number>: data about the call, session and element data, user information, or the activity of the caller (each of these is fully defined in the following sections). When testing if the child tag's value is true or false, it must be able to evaluate to a Boolean value. If it cannot, the decision will act as if the rule did not activate.
- **constant_string / constant_number**—These tags store string and number data in the value attribute. The number can be any integer or floating-point number.

Note The number can also be treated as a string. For example, if 1234 starts with 12.

The following sections explain the contents of the individual tags found within the <string>, <number> and <boolean> tags.

<call_data>

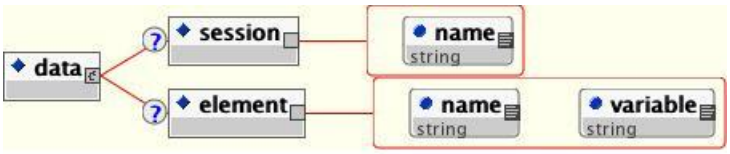
Figure 2: call_data Tag



The call_data Tag figure shows the term that represents information about the current call. The type attribute can be ani, dnis, uui, iidigits, source, appname, duration, language, or encoding. The ANI, DNIS, UUI, and IIDIGITS will be NA if it is not sent by the telephony provider. Source is the name of the application that transferred to this application or null if this application was the first to be called. Duration is the duration of the call up to this point in seconds.

<data>

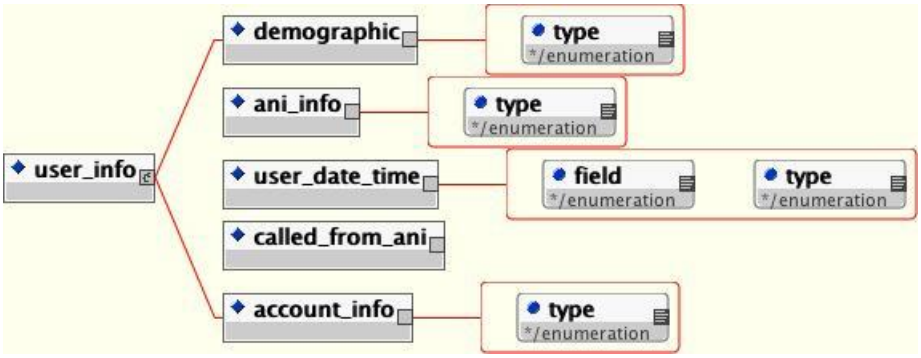
Figure 3: data Tag



The <data> Tag figure shows the term that represents session or element data. The `<session>` tag refers to session data with its name in the `name` attribute. The `<element>` tag refers to element data with the name of the element in the `name` attribute and the name of the variable in the `variable` attribute.

<user_info>

Figure 4: user_info Tag



The <user_info Tag figure shows the term that represents user information.

Note If the application has not been configured to use the user management system, and the call was not associated with a specific UID, using this term will cause an error.

Only one piece of user information can be returned per tag.

The possible user information to be compared is:

- **demographic**—This tag refers to the user's demographic information. The `type` attribute can be name, zipcode, birthday, gender, ssn, country, language, custom1, custom2, custom3, or custom4.
- **ani_info**—This tag refers to the various phone numbers associated with the user account. If the `type` attribute is first, the first number in the list of numbers is returned. This is returned if there was only one number associated with an account. If the attribute is num_diff the total number of different phone numbers associated with the account is returned.
- um_diff

  user_date_time—This tag refers to date information related to the user account. The `type` attribute indicates which user-related date to access and the `field` attribute is used to choose which part of the date to return. `Type` can be last_modified (indicating the last time the account was modified), creation

(indicating the time the account was created), and last_call (indicating the last time the user called). `Field` can be hour_of_day (which returns an integer from 0 to 23), minute (which returns an integer from 0 to 59), day_of_month (which returns an integer from 1 to 31), month (which returns an integer from 1 to 12), day_of_week (which returns an integer from 1 to 7 where 1 is Sunday), or year (which returns the 4 digit year).

- **called_from_ani**—This tag returns *true* if the caller has previously called from the current phone number, *false* if not.
- **account_info**—This tag refers to the user's account information. The `type` attribute can be pin, account_number, or external_uid.

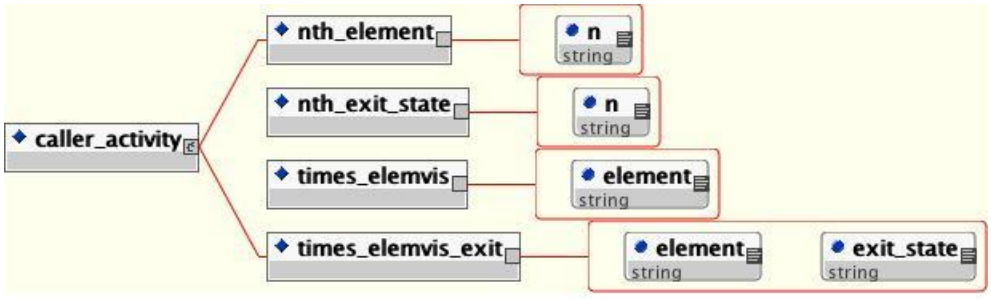
<general_date_time>

Figure 5: general_date_time Tag

The general_date_time> Tag figure shows the term that represents general date information. The `type` attribute indicates which date to access and the `field` attribute is used to choose which part of the date to return. `Type` can be current (indicating the current date/time) or call_start (indicating the time the call began). `Field` can be hour_of_day (which returns an integer from 0 to 23), minute (which returns an integer from 0 to 59), day_of_month (which returns an integer from 1 to 31), month (which returns an integer from 1 to 12), day_of_week (which returns an integer from 1 to 7 where 1 is Sunday), or year (which returns the 4 digit year).

<caller_activity>

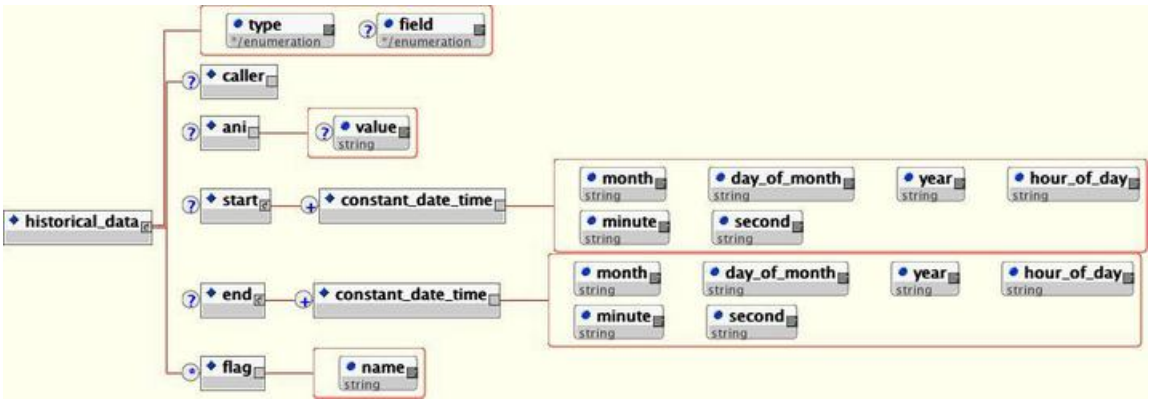
Figure 6: caller_activity Tag



The <caller_activity> Tag figure shows the term that represents the activity of the caller in the current call. The `<nth_element>` tag returns the nth element visited by the caller where the attribute `n` is the number (starting at 1). The `<nth_exit_state>` tag returns the exit state of the nth element visited by the caller where the attribute `n` is the number (starting at 1). The `<times_elemvis>` tag returns the number of times the caller visited the element whose name is given in the `element` attribute. The `<times_elemvis_exit>` tag returns the number of times the caller visited the element whose name is given in the attribute `element`, which returned an exit state whose name is given in the `exit_state` attribute.

<historical_data>

Figure 7: <historical_data> Tag



The <historical_data> Tag figure shows the term that represents the historical activity of the user associated with the call on the current application.

Note If the application has not been configured with a user management database, using this term will cause an error.

The `type` attribute determines what kind of value is returned. A value of num means that the value returned is the number of calls matching the criteria defined by the children tags. A value of last_date_time means that the value returned is the last date/time a call was received matching the criteria defined by the children tags. A value of first_date_time returns the first date/time a call was received that matched the criteria.

The `field` attribute is used if the `type` attribute is first_date_time or last_date_time and indicates which part of the date to compare. `Field` can be hour_of_day (which returns an integer from 0 to 23), minute (which returns an integer from 0 to 59), day_of_month (which returns an integer from 1 to 31), month (which returns an integer from 1 to 12), day_of_week (which returns an integer from 1 to 7 where 1 is Sunday), or year (which returns the 4 digit year). The children tags are used to turn on various criteria to add to the search.

The different search criteria are:

- **caller**—If this tag appears, the search looks for calls made by the current caller only. If it does not appear, it will search all calls made by all callers.

Note If the call was not associated with a specific UID, an error will occur if this tag is used.

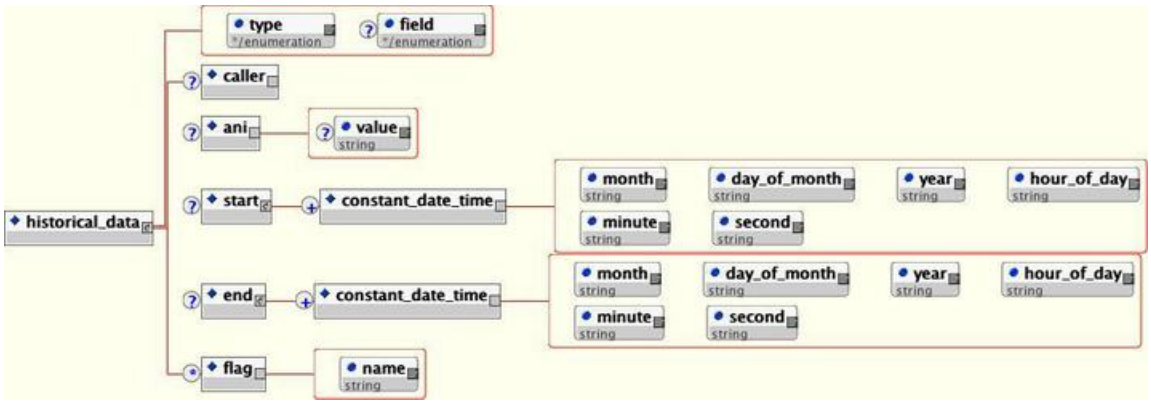
- **ani**—If this tag appears, the search looks for calls made by the ANI specified in the `value` attribute. If the `value` attribute is not included, the ANI of the current caller is used.
- **start**—If this tag appears, the search looks for calls whose start date/time are between two times specified by successive `<constant_date_time>` children tags. The attributes of `<constant_date_time>` define the specific date to use. The `month` attribute must be an integer from 1 to 12. The `day_of_month` attribute must be an integer from 1 to 31. The `year` attribute must be a four digit integer. The `hour_of_day` attribute

must be an integer from 0 to 23. The `minute` attribute must be an integer from 0 to 59. The `second` attribute must be an integer from 0 to 59.

- **end**—If this tag appears, the search looks for calls whose end date/time are between two times specified by successive `<constant_date_time>` children tags. See `<start>` (the previous bullet) for the description of the `<constant_date_time>` tag.
- **flag**—If this tag appears, the search looks for calls where a flag with the name given in the `name` attribute was triggered.

XML Decision Example1

Figure 8: <historical_data> Tag



The <historical_data> Tag figure shows the term that represents the historical activity of the user associated with the call on the current application.

Note If the application has not been configured with a user management database, using this term will cause an error.

The `type` attribute determines what kind of value is returned. A value of num means that the value returned is the number of calls matching the criteria defined by the children tags. A value of last_date_time means that the value returned is the last date/time a call was received matching the criteria defined by the children tags. A value of first_date_time returns the first date/time a call was received that matched the criteria.

The `field` attribute is used if the `type` attribute is first_date_time or last_date_time and indicates which part of the date to compare. `Field` can be hour_of_day (which returns an integer from 0 to 23), minute (which returns an integer from 0 to 59), day_of_month (which returns an integer from 1 to 31), month (which returns an integer from 1 to 12), day_of_week (which returns an integer from 1 to 7 where 1 is Sunday), or year (which returns the 4 digit year). The children tags are used to turn on various criteria to add to the search.

The different search criteria are:

- **caller**—If this tag appears, the search looks for calls made by the current caller only. If it does not appear, it will search all calls made by all callers.

Note If the call was not associated with a specific UID, an error will occur if this tag is used.

- **ani**—If this tag appears, the search looks for calls made by the ANI specified in the `value` attribute. If the `value` attribute is not included, the ANI of the current caller is used.
- **start**—If this tag appears, the search looks for calls whose start date/time are between two times specified by successive `<constant_date_time>` children tags. The attributes of `<constant_date_time>` define the specific date to use. The `month` attribute must be an integer from 1 to 12. The `day_of_month` attribute must be an integer from 1 to 31. The `year` attribute must be a four digit integer. The `hour_of_day` attribute must be an integer from 0 to 23. The `minute` attribute must be an integer from 0 to 59. The `second` attribute must be an integer from 0 to 59.
- **end**—If this tag appears, the search looks for calls whose end date/time are between two times specified by successive `<constant_date_time>` children tags. See `<start>` (the previous bullet) for the description of the `<constant_date_time>` tag.
- **flag**—If this tag appears, the search looks for calls where a flag with the name given in the `name` attribute was triggered.

XML Decision Example2

An application named Example2 randomly chooses two letters of the alphabet. The letters are chosen by an action element named *GetRandomLetter* and stored in element data named *letter1* and *letter2*.

A decision element named *IsCallerAWinner* would be needed which has three exit states:

- For a user whose name begins with either letter.
- For users whose name does not begin with the letters.
- For users whose name is not in the records (this could be an error or could prompt the application to ask the user to register on the website).

Even if the application assumes that all users will have their names on file, it is advisable to add this third exit state be sure. In this example, the default exit state is set to when the users do not match.

The rules of *IsCallerAWinner* decision element are:

Rule Number	Expression	Exit State
1	The caller's name begins with the value stored in the element *GetRandomLetter* with the variable name *letter1* or begins with the value stored in the element *GetRandomLetter* with the variable name *letter2*.	is a winner
2	The caller's name does not begin with the value stored in the element *GetRandomLetter* with the variable name *letter1* and does not begin with the value stored in the element *GetRandomLetter* with the variable name *letter2*.	not a winner
3	The caller's name does not exist.	no name

The Unified CVP decision element XML file is named *IsCallerAWinner* and will be saved in `%CVP_HOME%\VXMLServer\applications\Example2\data\misc`.

The XML file content will be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE knowledge_base SYSTEM "../../../../dtds/DecisionKnowledgeBase.dtd">
<knowledge_base>
   <rule name="NameStartsWith" default_exit_state="not a winner">
      <exit_state name="no name" conjugate="and">
         <boolean value="false" check_existence="yes">
            <user_info>
               <demographic type="name"/>
            </user_info>
         </boolean>
      </exit_state>
      <exit_state name="is a winner" conjugate="or">
         <string operator="starts_with">
            <user_info>
               <demographic type="name"/>
            </user_info>
            <data>
               <element name="GetRandomLetter" variable="letter1"/>
            </data>
         </string>
         <string operator="starts_with">
            <user_info>
               <demographic type="name"/>
            </user_info>
            <data>
               <element name="GetRandomLetter" variable="letter2"/>
            </data>
         </string>
      </exit_state>
  </rule>
</knowledge_base>
```

Note The *no name* exit state is listed first. This is because before we try to analyze the user's name, we have to first know that it exists. We check if the name *does not exist* first and if it fails, it means that the name exists and we can continue.

Note The second exit state must check if the name begins with the first *or* second letter but the last exit state must check if the name does not begin with the first *and* second letter.

XML Decision Example3

An application named Example3 is designed to trigger a flag named *account menu* when a caller chooses to manage their account. As of June 15, 2004, the menu options were changed for the account menu. We want to tell people the options have changed, but only if know they have visited that part of the application before June 15. If not, there is no reason to say anything because the caller is experiencing this for the first time. A decision element is needed that distinguishes between those to play the changed audio to from those who should encounter the menu ordinarily. The rule must deal with the day, month, and the year, making sure that callers from previous years and future years are handled correctly as well. Because the current state of the

XML format does not allow date comparisons, a way must be determined to make this restriction. The solution is to use multiple rules which progressively get more restrictive in a process-of-elimination method. Because all conditions are to be handled, the rule must include those who do not hear the changed message using the same scheme (there is no need to use the default exit state).

The rules of *account menu* decision element are:

Rule Number	Expression	Exit State
1	The year the last time the caller triggered the flag *account menu* is earlier than 2004.	play changed
2	The year the last time the caller triggered the flag *account menu* is later than 2004.	normal
Note	At this time, if the above two rules were not triggered, the caller triggered the flag in the year 2004.	
3	The month of the year the last time the caller triggered the flag *account menu* is less than 6.	play changed
4	The month of the year the last time the caller triggered the flag *account menu* is greater than 6	normal
Note	At this time, if the above two rules were not triggered, the caller triggered the flag in June 2002.	
5	The day of the month the last time the caller triggered the flag *account menu* is less than or equal to 15.	play changed
6	The day of the month the last time the caller triggered the flag *account menu* is greater than 15.	normal

The Unified CVP decision element XML file is named *DoesCallerNeedMenuChanges* and will be saved in `%CVP_HOME%\VXMLServer\applications\Example3\data\misc`.

The content of the XML file will be:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE knowledge_base SYSTEM "../../../../dtds/DecisionKnowledgeBase.dtd">
<knowledge_base>
   <rule name="NewMessageTest">
      <exit_state name="play changed" conjugate="and">
         <number operator="less">
             <historical_data type="last_date_time" field="year">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="2004"/>
         </number>
      </exit_state>
      <exit_state name="normal" conjugate="and">
         <number operator="greater ">
             <historical_data type="last_date_time" field="year">
                <caller/>
                <flag name="account menu"/>
             </historical_data>
             <constant_number value="2002"/>
```

```
            </number>
        </exit_state>
        <exit_state name="play changed" conjugate="and">
            <number operator="less">
                <historical_data type="last_date_time" field="month">
                    <caller/>
                    <flag name="account menu"/>
                </historical_data>
                <constant_number value="6"/>
            </number>
        </exit_state>
        <exit_state name="normal" conjugate="and">
            <number operator="greater">
                <historical_data type="last_date_time" field="month">
                    <caller/>
                    <flag name="account menu"/>
                </historical_data>
                <constant_number value="6"/>
            </number>
        </exit_state>
        <exit_state name="play changed" conjugate="and">
            <number operator="less_equal">
                <historical_data type="last_date_time" field="day_of_month">
                    <caller/>
                    <flag name="account menu"/>
                </historical_data>
                <constant_number value="15"/>
            </number>
        </exit_state>
        <exit_state name="normal" conjugate="and">
            <number operator="greater">
                <historical_data type="last_date_time" field="month">
                    <caller/>
                    <flag name="account menu"/>
                </historical_data>
                <constant_number value="6"/>
            </number>
        </exit_state>
    </rule>
</knowledge_base>
```

VoiceXML Insert Elements

There are certain situations in a voice application where a developer may want to include prewritten VoiceXML into their Unified CVP application. The developer may want fine-level control over a specific voice function at the VoiceXML tag level without getting involved with constructing a custom configurable element in Java. Additionally, the developer may want to integrate VoiceXML content that has already be created and tested into a Unified CVP application.

These situations are handled by a *VoiceXML insert element*.

- **VoiceXML Insert Element**—A custom element built in VoiceXML providing direct control of lower-level voice dialog at the price of decreased flexibility.

VoiceXML insert elements contain VoiceXML code that the developer makes available as the content of a VoiceXML `<subdialog>`. The content can be in the form of static VoiceXML files, JSP templates, or even dynamically generated by a separate application server. A framework is provided to allow seamless integration of VoiceXML insert elements with the rest of the call flow.

The use of VoiceXML insert elements can cause the following results:

- the loss of being able to seamlessly switch between different voice browsers
- some greater processing overhead involved with integration with the rest of the call flow
- the added complexity of dealing with VoiceXML itself rather than creating an application with easy-to-use configurable elements

VoiceXML insert elements can have as many exit states as the developer requires, with a minimum of one.

Insert Element Restrictions

The following restrictions apply to a VoiceXML insert element. An insert element conforming to these restrictions will be assured full integration with the Unified CVP application. These restrictions will be clarified later.

- The insert element cannot define its own root document, a root document generated by VXML Server must be used.
- The variables to return to VXML Server, including the exit state, must conform to a strict naming convention.
- When using the `<return>` tag, Unified CVP-specified arguments must be returned along with the custom variables.

Note Use a VoiceXML insert element only in a top-level subdialog. You cannot use a VoiceXML Insert element in a nested subdialog.

Insert Element Inputs

As with any element in the application, an insert element needs to be able to access information about the call such as element and session data, call data (such as the ANI), and even information found in the user management database if the application is configured to use one. Generally, this information is available in the Java or XML API. Because an insert element is written in VoiceXML, this information must be made available for the insert element to use from within the VoiceXML.

Unified CVP achieves this by creating VoiceXML variables in the root document containing all the desired information. The variable names conform to a naming convention so that the Insert element developer can refer to them appropriately. This is one reason why Unified CVP requires the use of the VXML Server-generated root document.

In order to reduce the number of variables appearing in the root document, the application designer is given the option of choosing which input groups are passed to the insert element. Additionally, the designer can individually choose which element and session data to pass. By minimizing the inputs to only the data required by the insert element, the overhead involved in using an Insert element is minimized.

These are the input types:

- **Telephony**—This information deals with telephony data. The inputs start with audium_telephony_.
  - **audium_telephony_ani**—The phone number of the caller or NA if not sent.

- **audium_telephony_dnis**—The DNIS or NA if not sent.
- **audium_telephony_iidigits**—The IIDIGITS or NA if not sent.
- **audium_telephony_uui**—The UUI or NA if not sent.
- **audium_telephony_area_code**—The area code of the caller's phone number. Will not appear if the ANI is NA.
- **audium_telephony_exchange**—The exchange. Will not appear if the ANI is NA.

- **Call**—This information deals with the call. The inputs start with audium_call_.
  - **audium_call_session_id**—The session ID.
  - **audium_call_source**—The name of the application which transferred to this one. Will not appear if this application is the first application in the call.
  - **audium_call_start**—The start time of the call in the format "DAY MNAME MONTH HH:MM:SS ZONE YEAR" where DAY is the abbreviated day of the week (for example, Wed), MNAME is the abbreviated name of the month (for example, Jun), HH is the hour (in military time), MM is the minute, SS is the seconds, ZONE is the time zone (for example, EDT), and YEAR is the four-digit year.
  - **audium_call_application**—The name of the current application.

- **History**—This information provides the history of elements visited so far in the call. The inputs start with audium_history_.
  - **audium_history**—This entire content of the element history (including exit states) is contained in this variable. The format is [ELEMENT]:[EXITSTATE]|..|[ELEMENT]:[EXITSTATE] where ELEMENT is the name of the element and EXITSTATE is the name of the exit state of this element. The order of the element/exit state pairs is consistent with the order in which they were visited. This will not appear if this insert element is the first element in the call.

- **Data**—This is the element and session data created so far in the call.
  - **audium_[ELEMENT]_[VARNAME]**—The element variable where ELEMENT is the name of the element and VARNAME is the name of the variable.

Note Both the element and variable names will have all spaces replaced with underscores. There may be no instances of this input if no element variables exist when this insert element is visited. For example, the variable audium_MyElement_the_value is element data named he value from the element MyElement.

  - **audium_session_[VARNAME]**—This is a session variable whose name is VARNAME.

Note The variable name will have all spaces replaced with underscores. The value is expressed as a string even if the type is not a string (the `toString()` method of the Java class is called). There may be no instances of this input if no session variables exist when this insert element is visited.

- **User Data**—This element information associated with the caller. It will only appear if the application has associated the call with a UID and a user management database has been set up for this application. The data will appear in the input exactly as in the database. The inputs start with user_.
  - **user_uid**—The UID of the user.
  - **user_account_number**—The account number of the user.
  - **user_account_pin**—The PIN of the user.
  - **user_demographics_name**—The name of the user.
  - **user_demographics_birthday**—The birthday of the user.
  - **user_demographics_zip_code**—The zip code of the user.
  - **user_demographics_gender**—The gender of the user.
  - **user_demographics_social_security**—The social security number of the user.
  - **user_demographics_country**—The country of the user.
  - **user_demographics_language**—The language of the user.
  - **user_demographics_custom1**—The value of the first custom column.
  - **user_demographics_custom2**—The value of the second custom column.
  - **user_demographics_custom3**—The value of the third custom column.
  - **user_demographics_custom4**—The value of the fourth custom column.
  - **user_account_external_uid**—The external UID of the user.
  - **user_account_created**—The date the account was created in the format. The value is in the format "DAY MNAME MONTH HH:MM:SS ZONE YEAR".
  - **user_account_modified**—The date the last time the account was modified. The value is in the format "DAY MNAME MONTH HH:MM:SS ZONE YEAR".
- **User By ANI**—Historical information about the phone number of the caller with regards to this application. It will only appear if a user management database has been set up for this application. The inputs start with user_by_ani_.
  - **user_by_ani_num_calls**—The number of calls made by this phone number.
  - **user_by_ani_last_call**—The last call made by the phone number. Will not appear if there were no calls made by this phone number in the past.

Insert Element Outputs

As with any element, VoiceXML insert elements can create element and session data, set the UID of the user to associate with the call, send custom logging events, and can return one of a set of exit states. As with voice elements, insert elements can have internal logging of caller activity and have global hotlinks and hotevents activated while the caller is visiting the Insert element. All of these actions involve variable data set within the Insert element and returned to VXML Server. These are crucial in order to properly integrate with the rest of the elements in the application.

These are the return arguments:

- **audium_exit_state** - The exit state of this VoiceXML insert element. The value of this variable must be exactly as chosen in the Builder for Call Studio when defining the insert element.
- **element_log_[VARNAME] / element_nolog_[VARNAME]** - These create new element data for this VoiceXML insert element whose name is VARNAME and which either sends a logging event to log the element data value or not, respectively. The data type will be assumed to be a string. The variable name cannot include spaces.
- **session_[VARNAME]** - This creates a new session variable whose name is VARNAME. The data type is assumed to be a string. The variable name cannot include spaces. If the variable name already exists, the old value will be replaced with this one. If the old data type was not a string, the new data type will be a string.
- **custom_[NAME]** - This sends a custom logging event whose contents is the action named NAME and the value of the variable being the description.
- **set_uid** - This associates the UID passed to the call.
- **audium_hotlink, audium_hotevent, audium_error, audium_action** - These four Unified CVP variables are created in the root document and must be passed along in the return `namelist`. The content of each deals with the occurrence of any global hotlinks, hotevents, errors, or actions (for example, a hang-up) while in this insert element. Because the subdialog has its own context and root document, this data has to be explicitly passed for any of these events to be recognized by VXML Server. The developer should not alter the contents of these variables.
- **audium_vxmlLog** - This variable contains the raw content for an interaction logging event. Adding to the interaction log is not required; the `audium_vxmlLog` variable can be passed empty. In order for VXML Server to parse the interaction data correctly, a special format is required for the content of the `audium_vxmlLog` variable.

  The format for interaction logging is:

  ```
  |||ACTION$$$VALUE^^^ELAPSED
  ```

  Where: ACTION is the name of the action.

  The following bullets list the possible action names and the corresponding contents of VALUE:

  - **audio_group** - Indicates that the caller heard an audio group play. VALUE is the name of the audio group.
  - **inputmode** - Reports how the caller entered their data, whether by voice or by DTMF key presses. VALUE should be contents of the `inputmode` VoiceXML shadow variable.
  - **utterance** - Reports the utterance as recorded by the speech recognition engine. VALUE should be the contents of the utterance VoiceXML shadow variable.

- **interpretation** - Reports the interpretation as recorded by the speech recognition engine. VALUE should be the contents of the interpretation VoiceXML shadow variable.
- **confidence** - Reports the confidence as recorded by the speech recognition engine. VALUE should be the contents of the confidence VoiceXML shadow variable.
- **nomatch** - Indicates the caller entered the wrong information, incurring a nomatch event. VALUE should be the count of the nomatch event.
- **noinput** - Indicates the caller entered nothing, incurring a noinput event. VALUE should be the count of the noinput event.

- ELAPSED is the number of milliseconds since the VoiceXML page was entered. The root document provides a JavaScript function named `application.getElapsedTime(START_TIME)` which returns the number of milliseconds elapsed since the time specified in `START_TIME`.

The root document created by VXML Server for use in all VoiceXML insert elements contains a VoiceXML variable named `audium_element_start_time_millisecs` that must be initialized with the time in order for the elapsed time intervals to be calculated correctly. This variable need only be initialized once in the *first VoiceXML page* of the insert element. All subsequent pages in the VoiceXML insert element must **not** initialize the variable because VXML Server requires the elapsed time from the start of the element, not the page. In VoiceXML, the line to appear must look like:

```
<assign name="audium_element_start_time_millisecs" expr="new Date().getTime()" />
```

For best results, this line should appear as early as possible in the first page, preferably in a `<block>` in the first `<form>` of the page, certainly before any additional logging is done.

In VoiceXML, setting the value of an existing variable requires the `<assign>` tag. Because the expression contains a JavaScript function, the `expr` attribute must be used. Additionally, in order to avoid overwriting previous log information, the expression must append the new data to the existing content of the variable. For example, to add to the interaction log the fact that the *xyz* audio group was played, the VoiceXML line would look like:

```
<assign name="audium_vxmlLog" expr="audium_vxmlLog + '|||audio_group$$$xyz^^^'
+application.getElapsedTime(audium_element_start_time_millisecs)"/>
```

In another example, the utterance of a field named xyz is to be appended to the log. The VoiceXML would look like:

```
<assign name="audium_vxmlLog" expr="audium_vxmlLog +'|||utterance$$$'+ xyz.$utterance +
'^^^' + application.getElapsedTime(audium_element_start_time_millisecs)"/>
```

Root Document

The subdialog context written by the developer must refer to a Unified CVP-generated root document. This is essential for proper integration of the VoiceXML insert element with VXML Server. The root document call must look like:

```
"/CVP/Server?audium_vxml_root=true&calling_into=APP&
namelist=element_log_value|RTRN1|RTRN2|…"
```

Where APP is the application name and RTRNX represents the names of all the element data, session data, and custom log entries (delimited by '|' characters) the insert element returns, using the same naming convention described in the outputs section.

The purpose for this requirement is related to how events are handled within the root document. The Unified CVP-generated root document catches events such as the activation of a global hotlink or a hangup, which

then requires the call flow to leave the insert element. The insert element, however, may have created element and session data or added custom content to the log. This information is stored in VoiceXML variables that would be deleted once the subdialog context is exited. The root document needs to know which VoiceXML variables to send along to VXML Server when one of these events is triggered so that it can store them accordingly. In order to avoid problems that might occur if a global hotlink or hotevent was activated right after the insert element began, the variables to be returned should be declared as near the start of the VoiceXML insert element as possible, even if they are not assigned initial values.

Note The ability to use a standard ampersand in the root document URL instead of escaping it (as &) is voice browser dependent. Most browsers will accept the escaped version so try that version first.

Note If the insert element does not need to send back any data in the `namelist` parameter, only the `element_log_value` variable need be included (the parameter should look like this: "...namelist=element_log_value").

Example of Insert Elements

In the example, a block is used to log the playing of the `initial_prompt` audio group. After this action, some inputs passed to it from VXML Server are played. Once played, it creates two element variables named `var1` and `var2` and a session variable named `sessvar`. After this action, it goes through a field that catches a number, and when done saves the utterance to the activity log and returns the exit state *less* if the number is less than 5 and *greater_equal* otherwise. The `<return>` tag returns the exit state, log variable, the four variables from the root document (error, hotlink, hotevent, and action), the two element data variables, the session data variable and a custom log entry (the number captured).

Also note that these last four variables are also passed to the root document call in the `<vxml>` tag so that events triggered within the insert element will correctly pass the data if it was captured by then.

Note The VoiceXML shown here may not function on all browsers without modification.

```
<?xml version="1.0"?>
<vxml version="2.0" application="/CVP/Server?audium_vxml_root=true&calling_into=MYAPP&
namelist=element_log_var1|element_nolog_var2|session_sessvar|custom_custlog">
   <form id="testform">
      <block>This is the initial prompt
          <assign name="audium_element_start_time_millisecs" expr="new Date().getTime()"/>
          <assign name="audium_vxmlLog" expr="'|||audio_group$$$initial_prompt^^^'
+application.getElapsedTime(audium_element_start_time_millisecs)"/>
      </block>
      <block>In the VoiceXML element.
        The ani is <value expr="audium_telephony_ani"/>.
        The element history is <value expr="audium_history"/>.
        User by ani num calls is <value expr="user_by_ani_num_calls"/>.
        Element data foo from element first <value expr="audium_first_foo"/>.
        Session variable foo2 <value expr="audium_session_foo2"/>.
      </block>
      <var name="element_log_var1" expr="'log me'"/>
      <var name="element_nolog_var2" expr="'do not log me'"/>
```

```
      <var name="session_sessvar" expr="'session_data_value'"/>
      <field name="custom_custlog" type="number">
        <property name="inputmodes" value="voice" />
        <prompt>Say a number.</prompt>
        <filled>
          <assign name="audium_vxmlLog" expr="audium_vxmlLog + '|||utterance$$$' +
custom_custlog.$utterance + '^^^'
+application.getElapsedTime(audium_element_start_time_millisecs)"/>
          <if cond=" custom_custlog < 5">
            <assign name="audium_exit_state" expr="'less'"/>
            <else/>
            <assign name="audium_exit_state" expr="'greater_equal'"/>
          </if>
          <return namelist="audium_exit_state audium_vxmlLog audium_error audium_hotlink
audium_hotevent audium_action element_log_var1 element_nolog_var2 session_sessvar
custom_custlog" />
        </filled>
      </field>
  </form>
</vxml>
```