



IP Telephony Applications: Core Development Concepts

Cisco Unified Application Environment 2.3

1. ABOUT THIS GUIDE

This section identifies the intended audience for this guide and lists the typographical conventions in effect.

1.1. Intended Audience

This guide is intended for use by programmers wishing to learn how to develop telephony applications for the Cisco Unified Application Environment. You should be familiar with the basics of the Cisco Unified Application Designer as a prerequisite to this document.

1.2. Typographical Conventions

The following typographical components are used for defining special terms and command syntax:

Convention	Description
Bold typeface	Represents literal information such as <ul style="list-style-type: none">• Information and controls displayed on screen, including menu options, windows dialogs and field names• Commands, file names, and directories• In-line programming elements, such as class names and XML elements when referenced in the main text
<i>Italic</i> typeface	Italics typeface is used to denote <ul style="list-style-type: none">• A new concept• A variable element such as <i>filename.mca</i>. In this example, <i>filename</i> represents the filename and .mca is the extension.• A reference to a chapter or section heading
<code>Courier</code> typeface	Denotes code or code fragments
UPPERCASE	Denotes keys and keystroke combinations such as CTRL+ALT+DEL.

2. MAKING AND RECEIVING CALLS

A basic feature of the Cisco Unified Application Environment is of course the ability to make and receive IP phone calls. While the reader will be aware that the UAE will implicitly or explicitly make use of call control protocols such as H.323, CTI, SCCP, and SIP, developing a telephony application for the UAE platform does not require an intimate understanding of the various protocols. In fact, using the UAE *CallControl* API, it is possible to design a script which can leverage any or all of these protocols without specifying any protocol-specific information at design time.

NOTE: The UAE CallControl API abstracts IP telephony protocols into a simple, unified interface

3. EXAMPLE 1: HANDLING AN INCOMING CALL

When a call is routed to the UAE as an incoming call, the call is presented to the application layer as a *triggering event*. A triggering event causes the application runtime to launch an instance of an application script matched to that event. This script can then choose to accept, answer, or reject the call. Most applications will want to answer a call, so let's begin our discussion there.

Sample Code

In this and the other tutorials contained in this guide, the reader can either open the accompanying complete sample project in the Cisco Unified Application Designer, or can create the sample application from scratch by following along with the tutorial. The completed application for this current tutorial can be found in the **HandleInboundCall** script of **ReceiveCalls.max..**

NOTE: The executable unit of a application is the script. An application, or project, can contain any number of scripts.

3.1. Create a Project and Script

To create a new sample project, open the Application Designer and select New Project from the File menu. The dialog as shown in Figure 1 is shown. Name the project *ReceivingCalls* as shown.

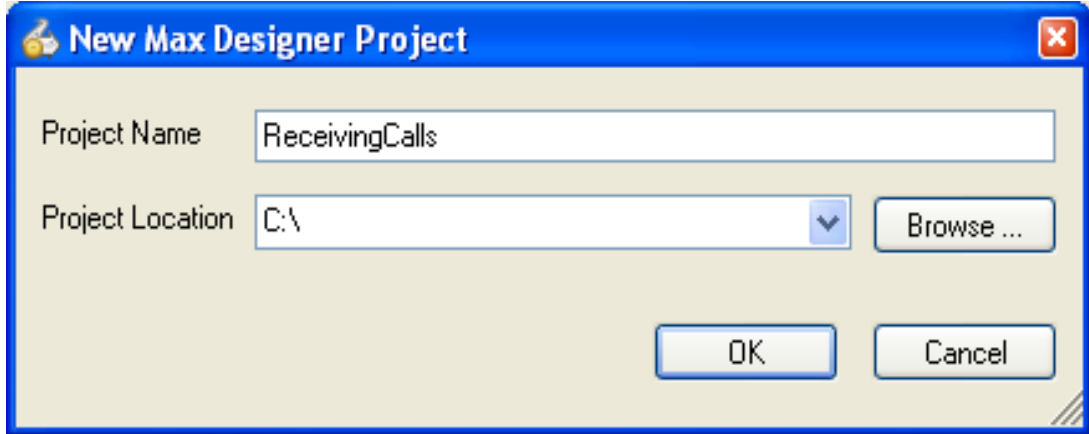


Figure 1: Creating a new project

Once your new application has been created, you will add a script to the project to handle an incoming call. Select *Add Script* from the *File* menu, the *Project* menu, or from the Project Explorer. The *New Application Script* dialog appears as shown in Figure 2. Here you will select the triggering event for your script, which will be *Metreos.CallControl.IncomingCall* for this example.

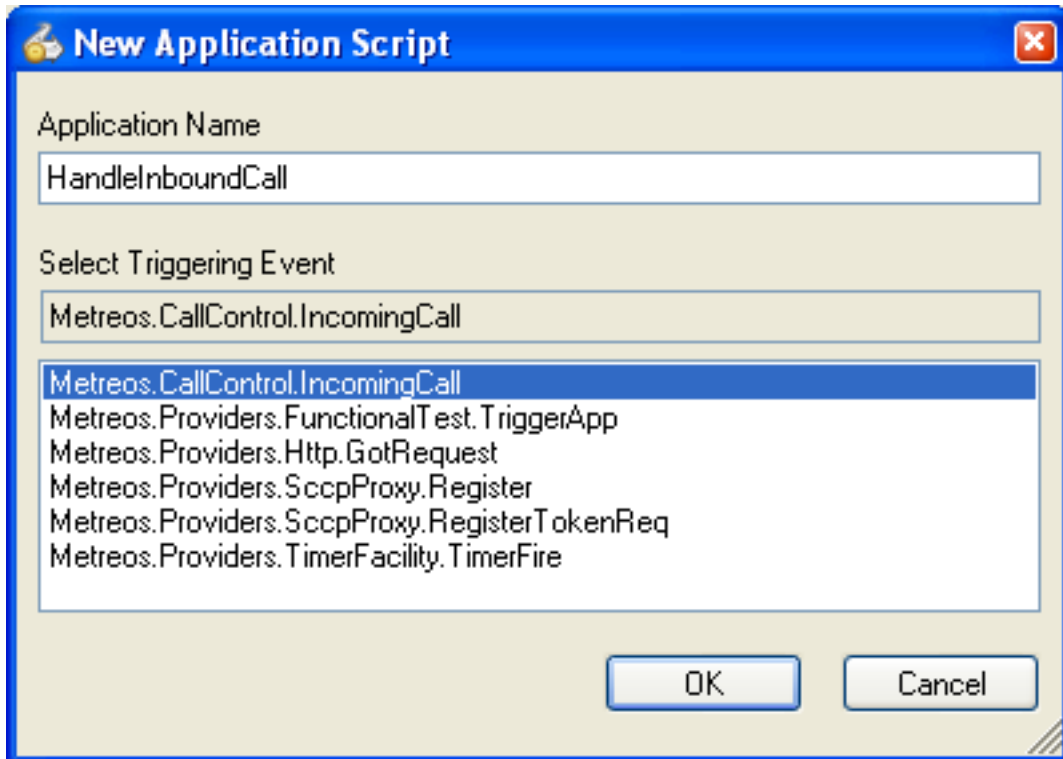


Figure 2: Creating a new script

3.2. Answering a Call

With the needed steps taken to prepare a script for development, we can focus on the act of answering an inbound call.

NOTE: Each call made and received by the UAE is given a unique identifier. Every action and event in the UAE CallControl API uses this identifier to indicate which call it should operate on.

The *Metreos.CallControl.IncomingCall* event identifies to the application which call is being handled by passing to it the unique identifier for the call as a parameter of the event. The name of this parameter is *CallId*.

NOTE: Events in the development environment are accompanied by information qualifying the specific event as event parameter fields.

To gain programmatic access to the *CallId* of the incoming call, you will create a local variable in the function handling the *IncomingCall* event, further specifying that the variable should be initialized by the event parameter **CallId**. Name the variable **incomingCallId**.

Figure 3 through Figure 7 illustrate this process.

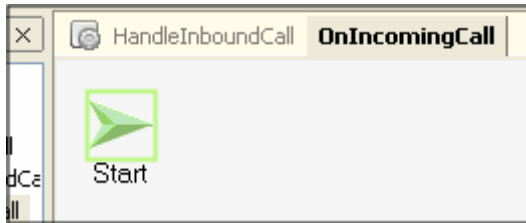


Figure 3: Choose the OnIncomingCall tab to switch to the function canvas.

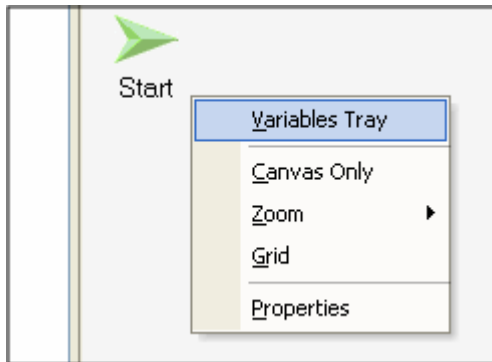


Figure 4: Right-click anywhere on the canvas

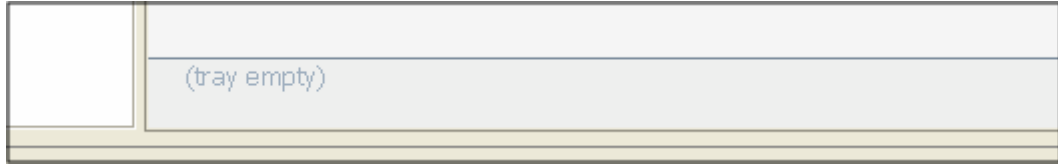


Figure 5: Notice that the variable tray appeared at the bottom of the canvas

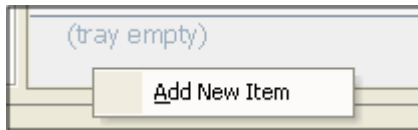


Figure 6: Right-click in the variable tray to add a new variable.

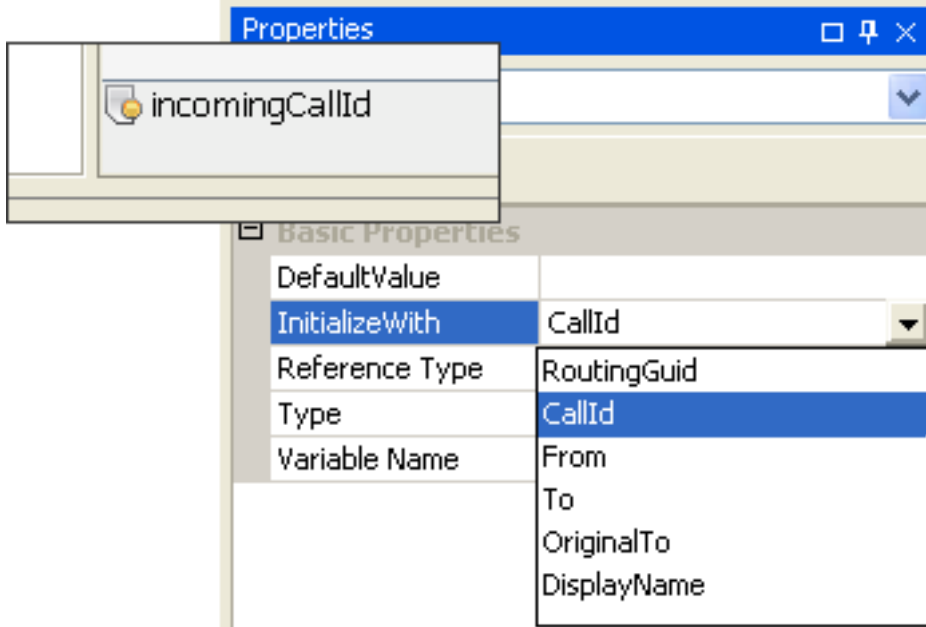


Figure 7: Initializing a variable with the CallId event parameter

With the *CallId* parameter now accessible in our local variable, we can use the *AnswerCall* action to ... answer the call! From the *CallControl* tool group in the Application Designer toolbox, drag *AnswerCall* onto the canvas.

With the *AnswerCall* action selected on the canvas, the properties of that action will be visible in the Designer's Property Grid. Change the *InitializeWith* parameter of the *incomingCallId* variable to *CallId*, by selecting *CallId* from the dropdown, as illustrated in Figure 8.

The Application Server will, upon receiving the call, automatically create a connection to a Media Engine associated with this application¹. During this creation process, the transmit and receive audio streams associated with the call are synchronized with the Media Engine connection. To put this another way, media sent by the call terminates to a Cisco Unified Media Engine, and the Media Engine sends the media to the IP endpoint specified by the call. Once the *AnswerCall* action has completed and returned a result to the application, the application can make use of any of the capabilities of the Media Engine connection

NOTE: Every Media Engine connection is given a unique identifier. Every action and event in the UAE.MediaControl API uses this identifier to indicate which Media Engine connection it should operate on.

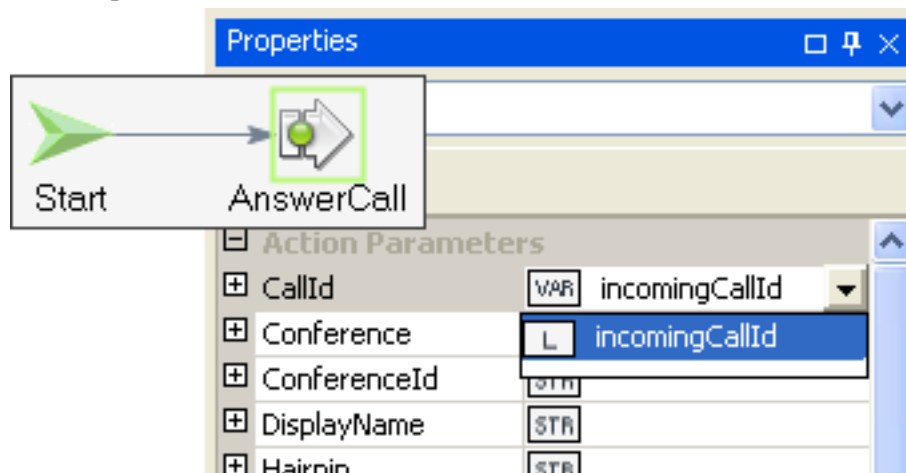


Figure 8: Answering a call

The *AnswerCall* action returns the Media Engine *ConnectionId* as a result parameter, which can be stored in a variable, as shown in Figure 9.

¹ Unless the call is specified as a peer-to-peer call, which is not covered here.

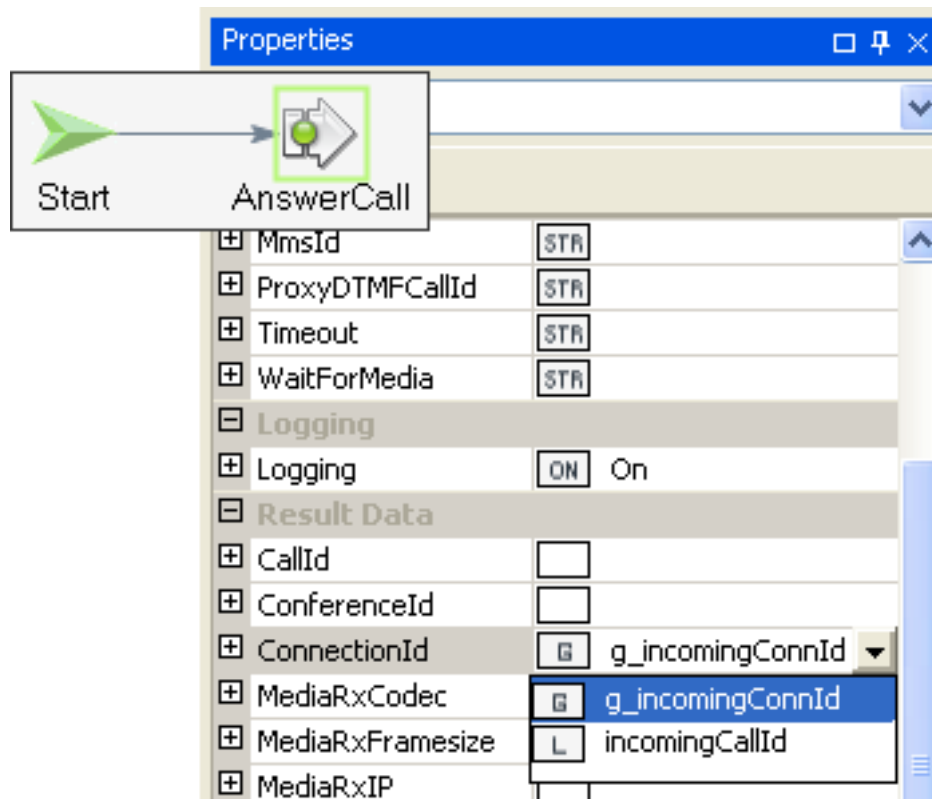


Figure 9: Storing connection ID

In this case we used a global variable in which to store *ConnectionId*, so that any function in the script can access the value. The following steps Figure 10 through Figure 12 show how to make a global variable.

NOTE: Global and local variables can be named whatever you wish. This document uses the g_ convention when naming a global variable.

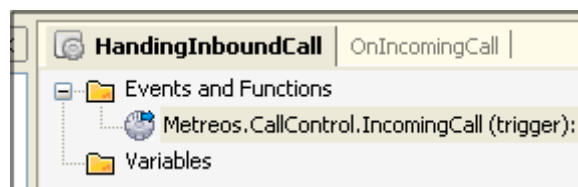


Figure 10: Focus the global canvas

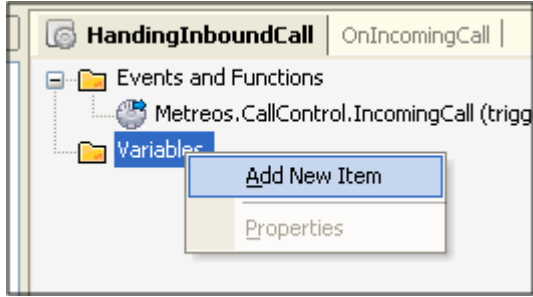


Figure 11: Right-click on Variables node

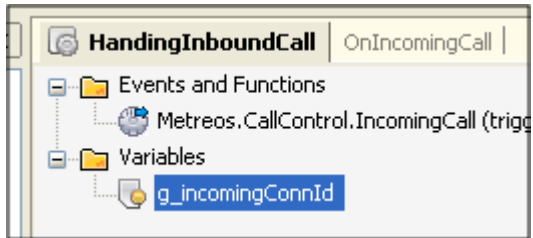


Figure 12: Name the global variable

To illustrate use of the Media Engine, let us add a Play action after our *AnswerCall* action to stream some text-to-speech to the caller. Drag a Play action onto the canvas from the Media Control toolbox group.

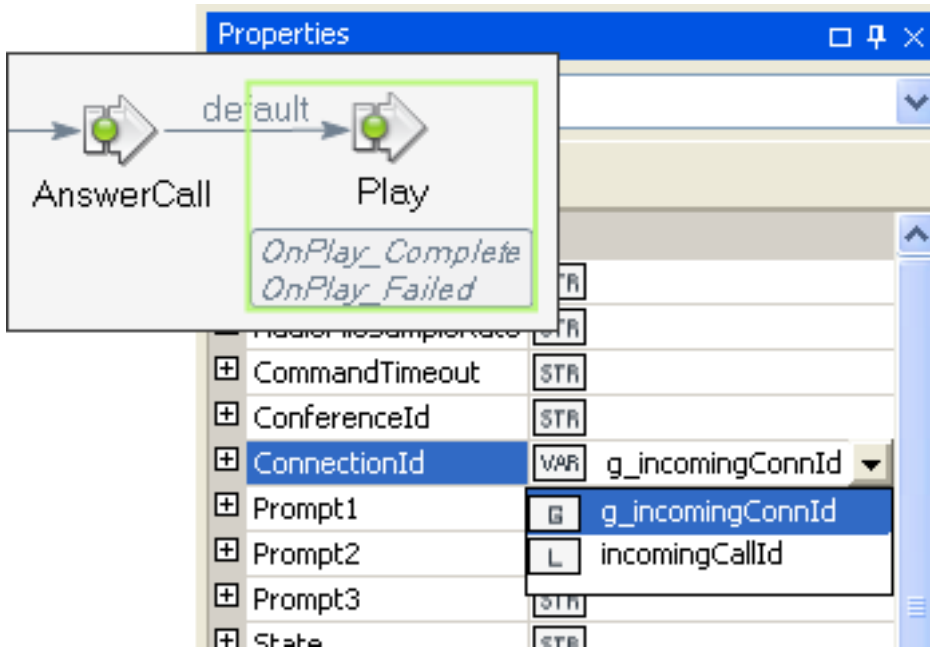


Figure 13: Preparing a Play action

Indicate to the Play command the connection to which it should play by setting the *ConnectionId* property as shown in Figure 14. We will now specify a text string for the Media Engine to convert to speech a stream to the caller. The Play action lets you specify up to three prompts, which can be a mixture of TTS strings and pre-recorded wave files. For this example, we will provide a string value only for **Prompt1**. For this example, we specify “Streaming audio is easy” as the value of that property, as illustrated in Figure 14.

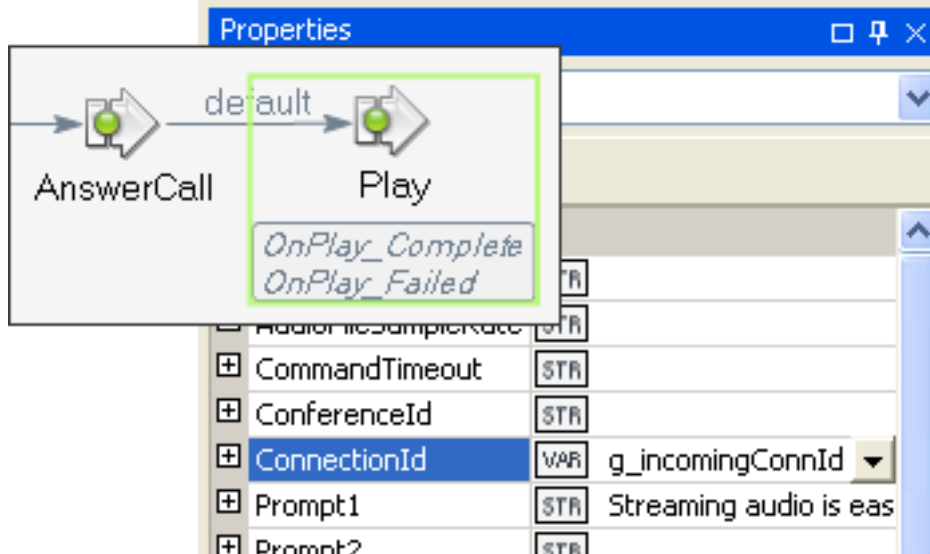


Figure 14: Specifying TTS

Because the play action can take an arbitrarily long time to complete, it is an asynchronous action, meaning the final response of the action is treated as a separate event. In our example, this is obvious by observing that two new functions appeared as soon as the *Play* action was dropped on the canvas: *OnPlay_Complete* and *OnPlay_Failed*. One of these event handler functions will be called once the *Play* terminates.

We will eventually want to issue a *Hangup* action once the *Play* completes, rounding out our example. Before we move on though, we should finish the *OnIncomingCall* function. We must end execution of this function such that control is relinquished and the script can then handle other events.

Only one event can be processed at a time for a given script.

End the script by dragging an *EndFunction* action from the Application Components tool group onto the canvas and link it after the *Play* action, as shown following.

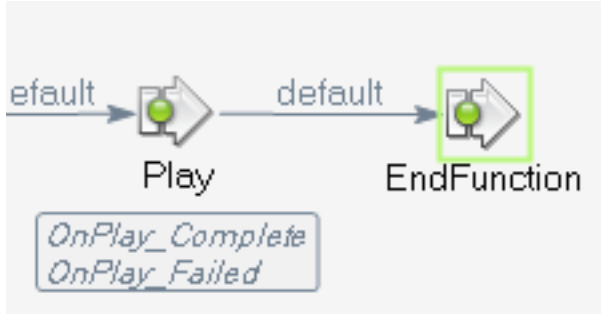


Figure 15: Placing the script into a wait state

We must also store the *CallId* of the incoming call in a global variable if we wish to use it later in another function, such as *OnPlay_Complete*. Create a new global variable named *g_incomingCallId* for this purpose.

Referring back to the *AnswerCall* action, we can leverage the *CallId* **result parameter** to provide an easy means of storing the *CallId* value passed in via *incomingCallId*, to global variable *g_incomingCallId*. This is possible because the *AnswerCall* action, as a convenience to the developer, returns the same *CallId* passed in through the action parameter to the result parameter *CallId*. In situations where an action does not provide this facility, the *Assign* action can be used instead. Figure 16 illustrates this process.

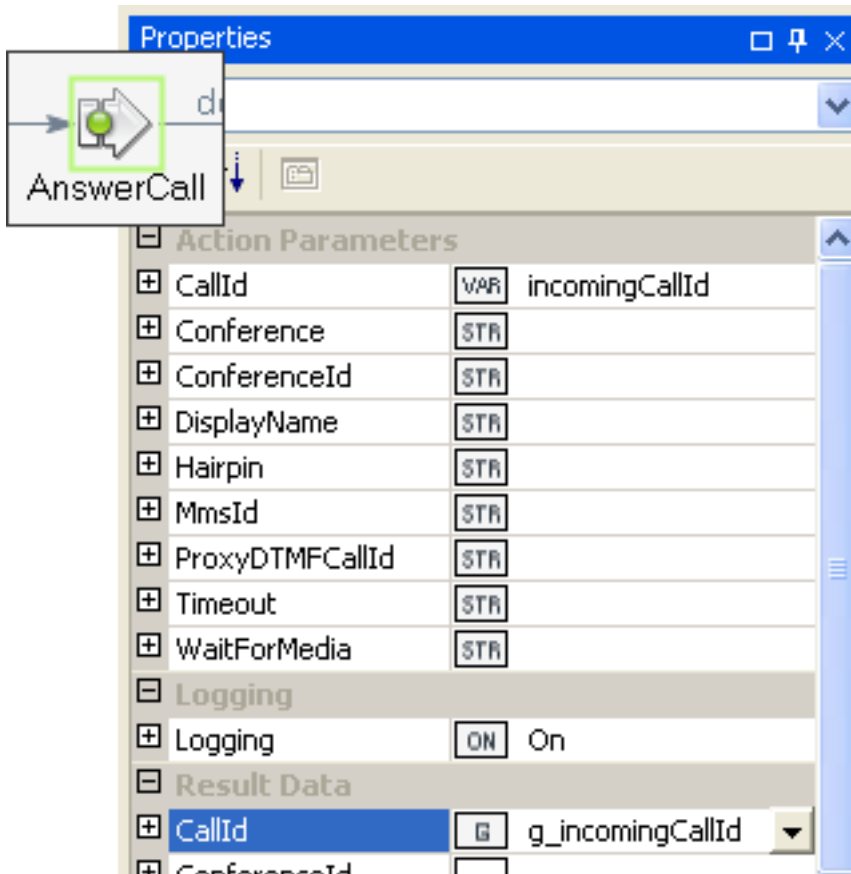


Figure 16: Storing CallId to a global variable

NOTE: Some CallControl actions return CallId as a convenience, so that the incoming CallId can easily be copied to a global variable.

With that out of the way, let us now turn to hanging up the call once the *Play* command has completed. By inserting a *Hangup* action as the first node in the *OnPlay_Complete* function, and specifying the *CallId* by using the *g_incomingCallId* variable, as shown in Figure 17, the call will be hung up immediately after the play completes.

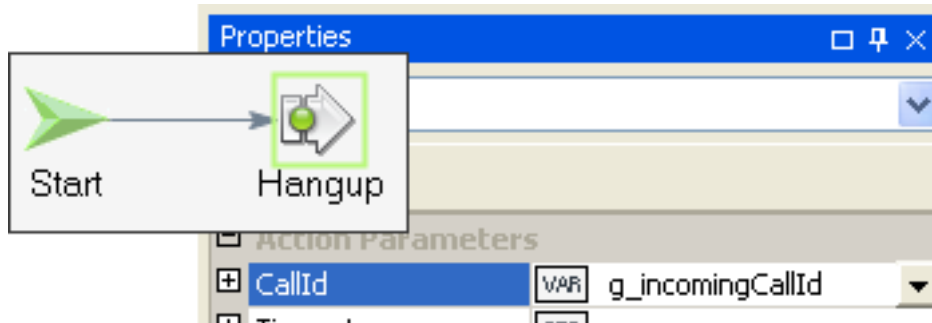


Figure 17: Hanging up the call

Once the call is hung up, there is nothing left for this script to do, and there are no further possible events that could cause execution to continue. A script instance must be explicitly destroyed, so place and link an *EndScript* action on the canvas.

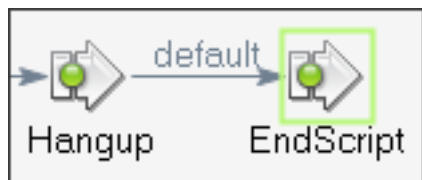


Figure 18: Ending the script

To simplify this example, duplicate what we have done so far in the *OnPlay_Complete* handler in the *OnPlay_Failed* handler: That is, *Hangup*, followed by *EndScript*.

The application in its present form can be built and deployed on the UAE. If you are new to this process, now would be a good time to deploy the application and test it.

4. EXAMPLE 2: BRIDGING MEDIA OF TWO CALLS

With the UAE platform you have two means of bridging media of two disparate calls.

1. **Media Conference:** Place the Media Engine connections for both calls in a conference.
2. **Peer-to-Peer** Bridge the media directly between the IP endpoints.

Each bridging technique has its tradeoffs.

Pros and Cons of Bridging Media using the Conferencing Method

Pro:

- **Control:** once in a media conference, you can easily add more participants, record, play conference-wide announcements, control volume of the individual participants, and so on.
- **No Unnecessary Conference Resources Used:** when only two participants are in the conference, no Media Engine conference resources are used, which is an advantage since conference resources are a licensed expense.

Con:

- **Latency:** there is an introduced latency of 70ms when hairpinning (2 parties in the conference), and 170ms when not (more than 2 parties in the conference)
- **Expense:** requires conference port capabilities (when more than 2 parties), an additional licensed expense. Requires 2 'RTP' port capabilities even when only 2 parties.
- **Possible Audio Blip:** there will be a delay of perhaps 100ms when promoting a 2-party hairpinned conference to a media conference, resulting in a blip in the audio which may or may not be noticeable.

Pros and Cons of Bridging Media using the Peer-to-Peer Method

Pro:

- **No added latency**
- **Possible Audio Blip:** switching to a media conference requires that both participants have their media renegotiated, resulting in an audible gap in the media.
- **Cost Effective:** no Media Engine connections are used

Con:

- **Limits Protocol:** not all combinations of call control protocols support peer-to-peer.

The key to knowing which type of media bridging to use is through understanding the needs of your application. For example, if you are building an application which will often have three or more participants, it may make sense to never use peer-to-peer because of the audible media gap.

The simplest type of media bridging to demonstrate is media conferencing. Using the script we created last time as a base, we can quickly add a second call to our script logic and bridge the two calls together.

Sample Code

The completed application for this current tutorial can be found in the **BridgingCalls** script of `ReceiveCalls.max`.

If you are following along with the Application Designer, you can easily jump start where we left off by creating a new project, and add the `HandleIncomingCall` script from the last example by navigating to *File > Add Script > Existing Script*.

4.1. Making a Call

To begin, let us no longer hang up once the `OnPlay_Complete` event fires. Instead, we can make a call with the goal of bridging the media of the newly formed outbound call with the inbound call.

After deleting the *Hangup* and *EndScript* actions, the first action of the `OnPlay_Complete` event handler now becomes a *MakeCall* action. The only parameter that absolutely must be specified, **To**, determines the number offered to `CallManager` when the call is placed. Specify a number here that makes sense for your own deployment.

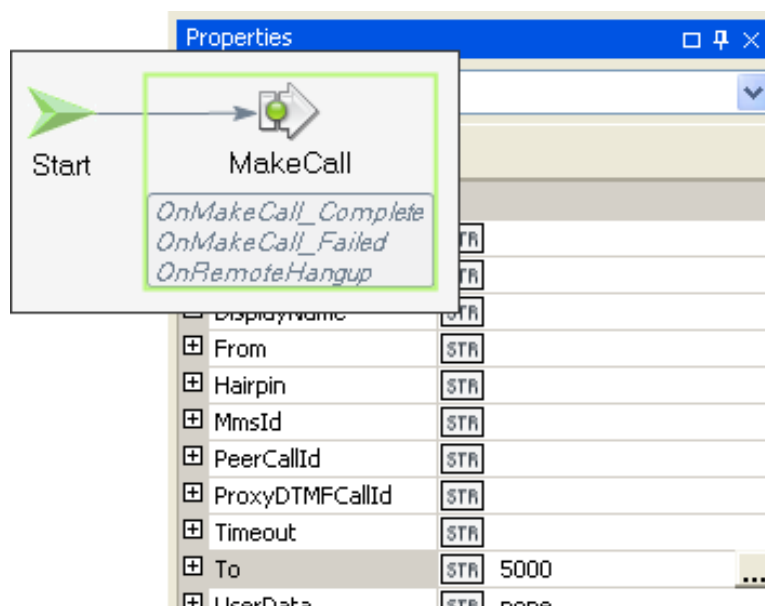


Figure 19: Making a call

When a `MakeCall` is issued to the runtime, it is assigned a **CallId**, as are incoming calls. As with the inbound call, we should store this *CallId* value in a global variable, `g_outboundCallId` so it is easily accessible later.

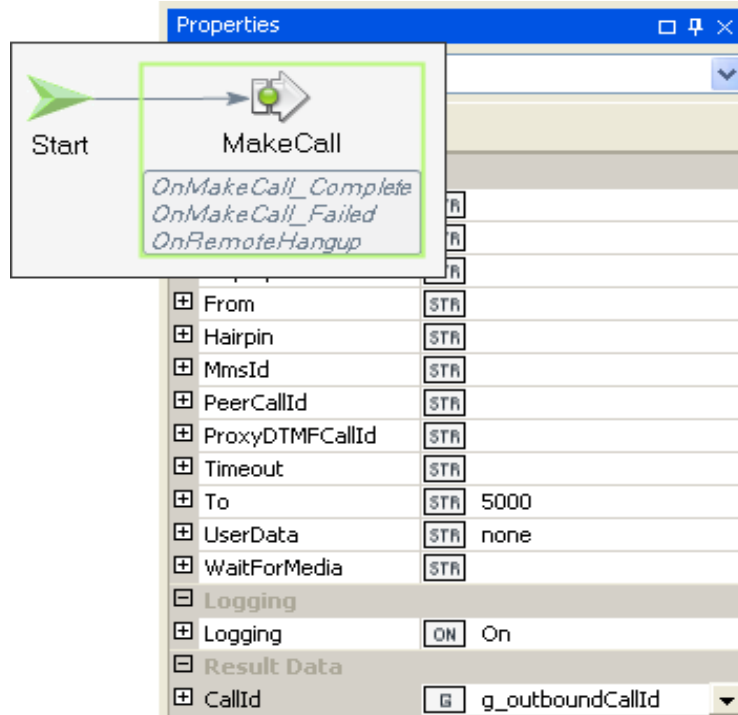


Figure 20: Saving Outbound CallId

MakeCall is an asynchronous action, so we do not know the final outcome of the call until the *OnMakeCall_Complete* or *OnMakeCall_Failed* event fires. With nothing else to do in the *OnPlay_Complete* handler, let us now apply an *EndFunction* action to place the script back into a wait state. Figure 21 illustrates this process.

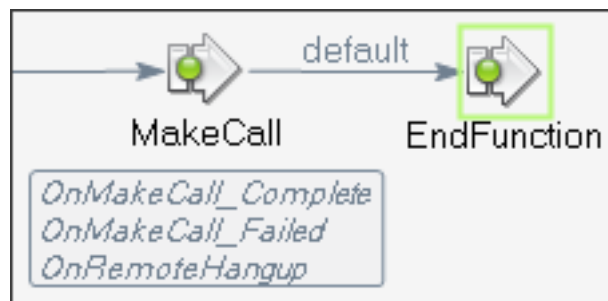


Figure 21: Placing script in wait state

With regards to media connections, the same rule applies as with the *AnswerCall* action: when the *MakeCall* action completes, a Media Engine connection has been automatically created and is accessible by the developer. If and when the outbound call is answered by the dialed party, the *OnMakeCall_Complete* event handler will fire, and the Media Engine connection then becomes available for that call.

To access this value, make a local variable in the *OnMakeCall_Complete* event handler which is initialized with the *ConnectionId* event parameter, as shown following.

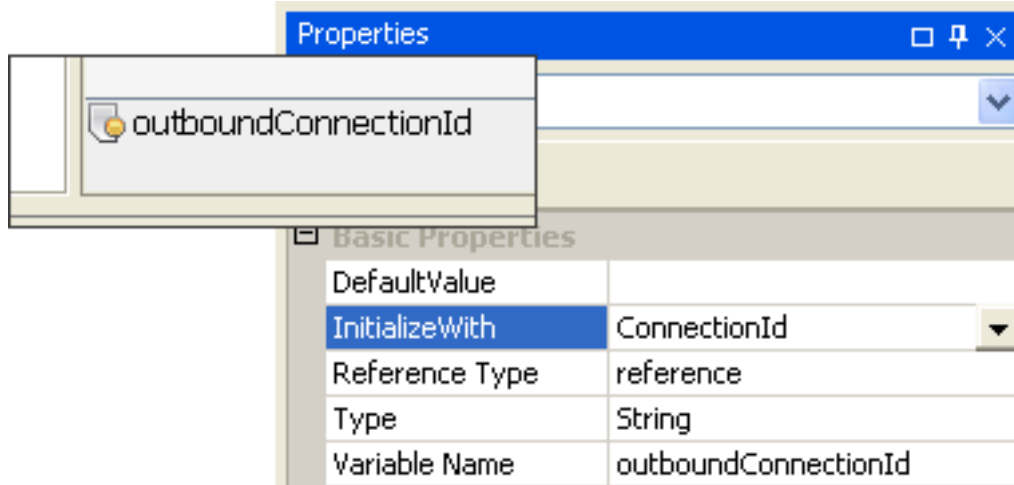


Figure 22: Storing the ConnectionId event parameter to a local variable

In order for the two calls to be able to hear each other, both Media Engine connections will need to be placed into a Media Engine conference. First, we will have to create the conference, as shown in Figure 23. A conference on the Media Engine must always have one participant, so for this application we have chose one of our two connections to place into the conference.

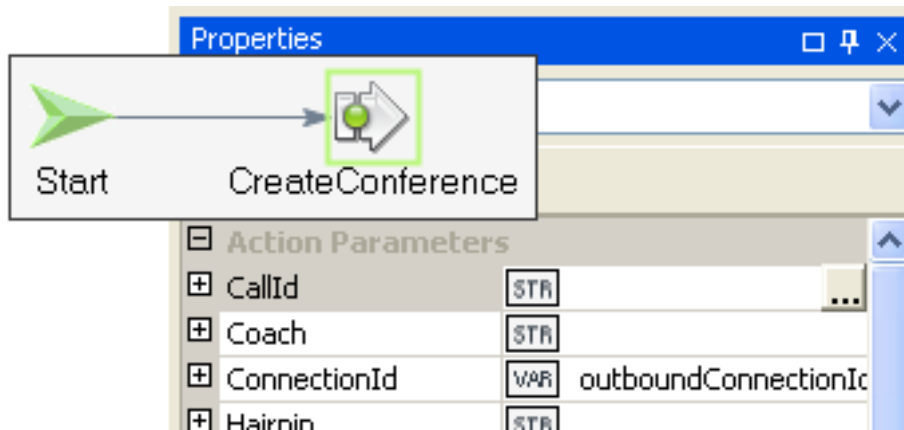


Figure 23: Creating a conference with one participant

Once the conference is created, the *CreateConference* action will return a unique identifier to the conference itself, so that further media operations can be performed upon it.

NOTE: Every Media Engine conference is given a unique identifier. Actions that manipulate Media Engine conferences use this identifier to indicate which conference it will operate on.

As with *callIds* and *connectionIds*, the most useful location to store a *conferenceId* is in a global variable, as shown following.

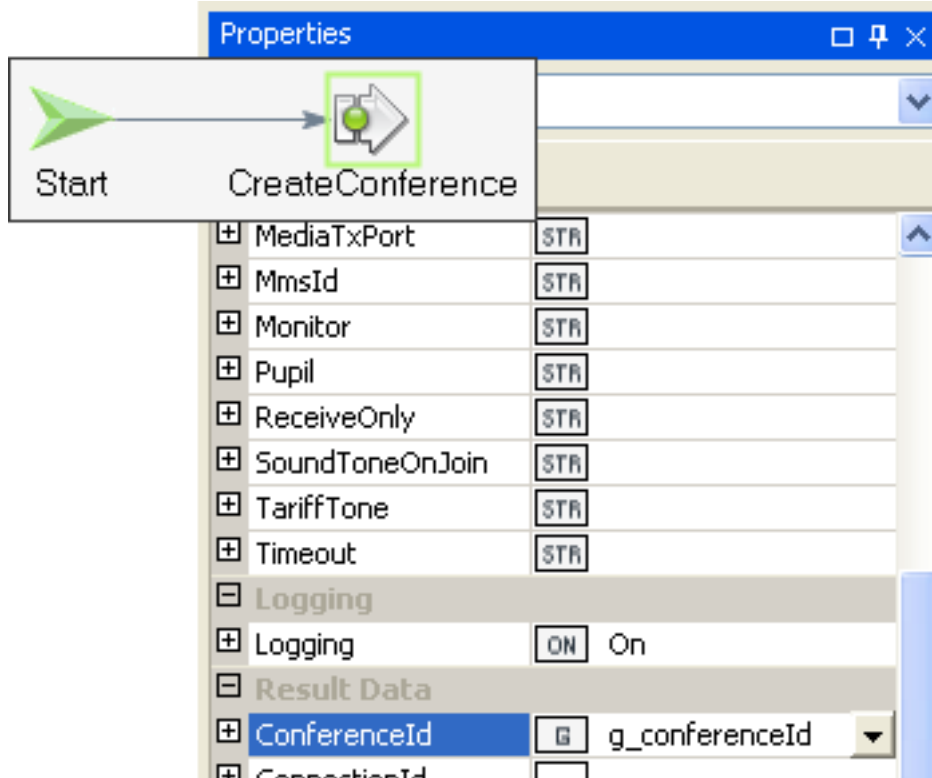


Figure 24: Storing the ConferenceId

Placing the other participant in the conference is done by using a *JoinConference* action. By specifying a *ConnectionId* and *ConferenceId* as properties of this action, the *JoinConference* will cause a new connection to be placed in the conference. Figure 25 shows this process.

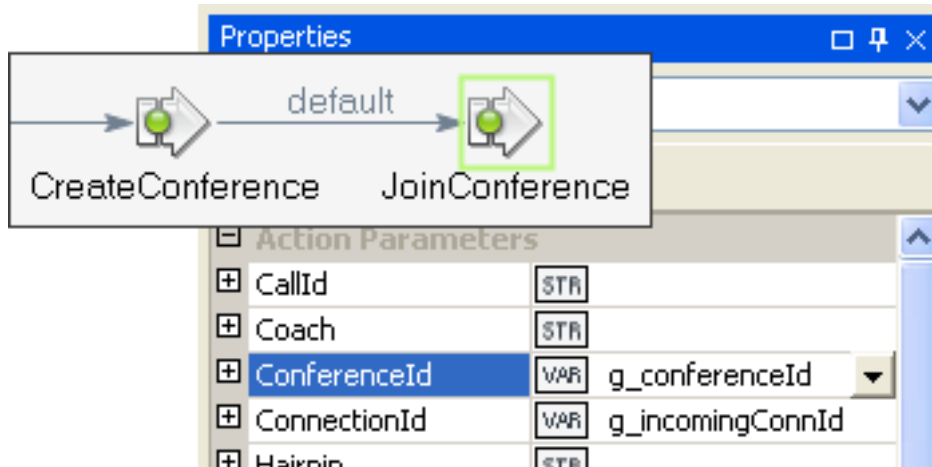


Figure 25: Adding a new connection to the conference

Once the *JoinConference* command completes, both parties should be able to hear each other.

With nothing else to do until one party or the other hangs up, invoking an *EndFunction* allows our script to enter a wait state, waiting for the *OnRemoteHangup* event.

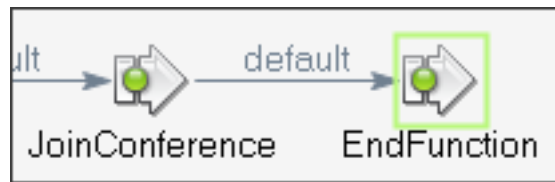


Figure 26: Entering wait state after conferencing

Before applying logic to the *OnRemoteHangup* event handler, let us take a moment to define what should happen when either caller hangs up. In most two-party calls, each party expects that when the other party hangs up, their own line will become disconnected. Following this as a model for how we will define the logic flow of the *OnRemoteHangup* event handler will give our script standard and expected behavior.

The *CallId* event parameter accompanies all *RemoteHangup* events, so that the application developer can distinguish which call leg hung up. By using the *Compare* action, as shown in Figure 27, we can determine if the call that hung up was the incoming call. Figure 28 gives an example.

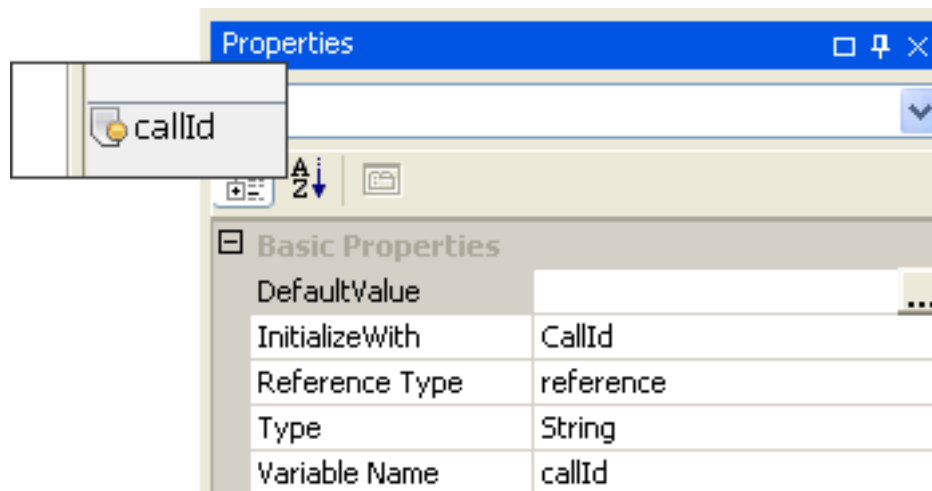


Figure 27: Initializing the CallId specifying the hung up call

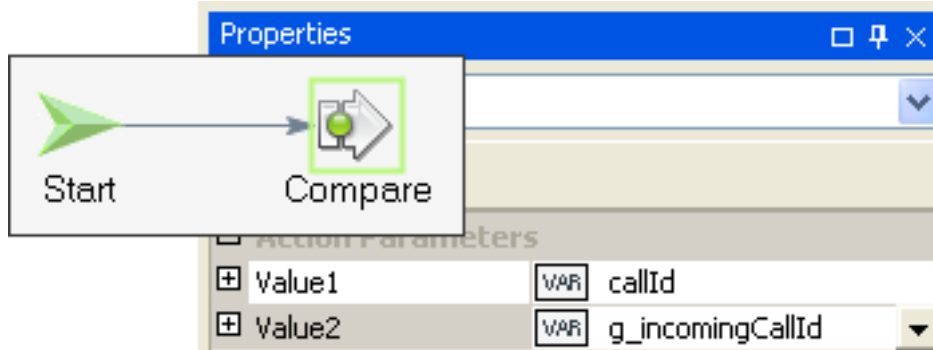


Figure 28: Determining if the hung up call is the incoming call

For the first time in this tutorial, we will need to leverage the result path returned by a native action. In this case, we are interested in the result of the Compare action. As you might expect, the *Compare* action returns one of two values: equal and unequal. Note that *default*, as shown in Figure 29, is a special path that indicates that it should be taken in the absence of a more explicit path.

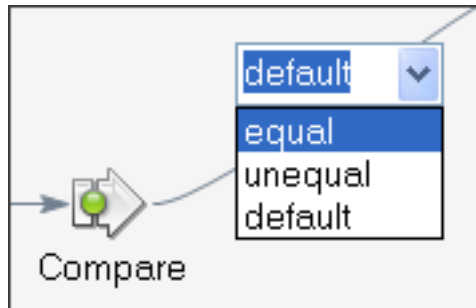


Figure 29: Utilizing specific result paths

In the case that the *incomingCallId* and *callId* variable compare equal, we know that our incoming call leg has been hung up. Therefore we should explicitly hang up the other call, the outbound leg, to create the expected behavior described earlier. Figure 30 shows how to do this.

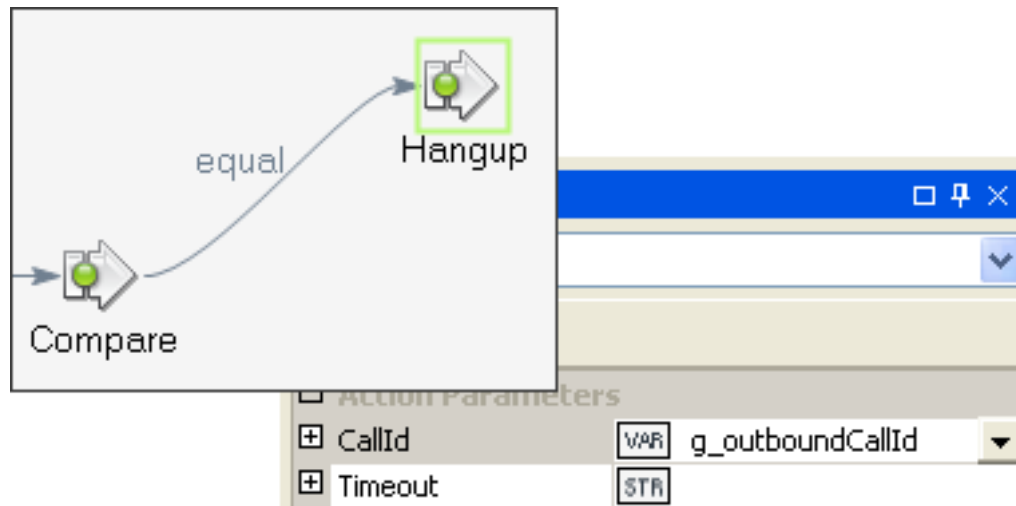


Figure 30: Hanging up the outbound call

In the other case, we would instead hang up the incoming call leg as shown in Figure 31.

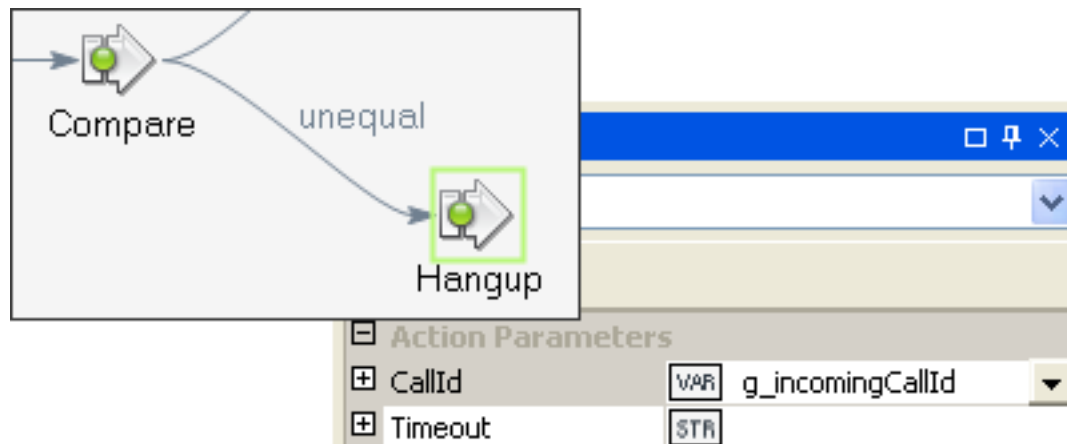


Figure 31: Hanging up the inbound call

In either case, after the *Hangup* action, both calls have been cleared, and the only task left is to destroy our script so as to clean up all resources associated with it. The code to do so is shown in Figure 32.

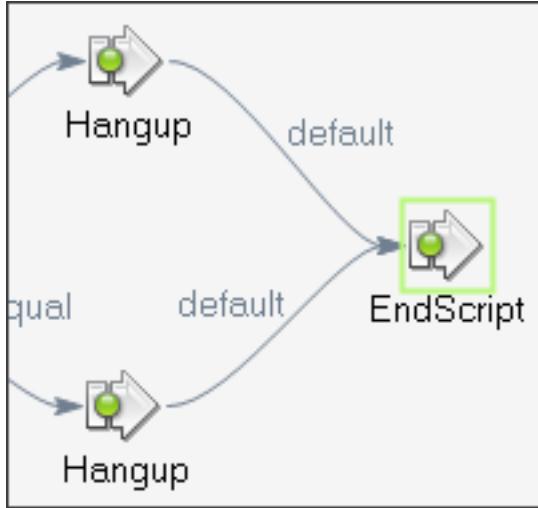


Figure 32: Ending the script after final hangup

You may have noticed that although we explicitly created a conference for our media connections, we did not destroy that conference. To understand why this is acceptable, it is important to note that the Media Engine will automatically destroy any Media Engine conference when the number of participants in that conference becomes zero. Also, when a *Hangup* is issued, or when a *RemoteHangup* event fires, the associated Media Engine connection for that call is destroyed. So, our conference will be destroyed automatically in this scenario.

NOTE: The Application Environment creates Media Engine connections automatically when answering or making calls. Accordingly, Media Engine connections are destroyed as calls are hung up.

To conclude this particular call flow, we need to provide a sequence of actions for the case that the *OnMakeCall_Failed* event handler fires. To keep the script simple (although not necessarily as user-friendly as possible), we will hang up the incoming call that we have already answered and exit the script, as shown following.

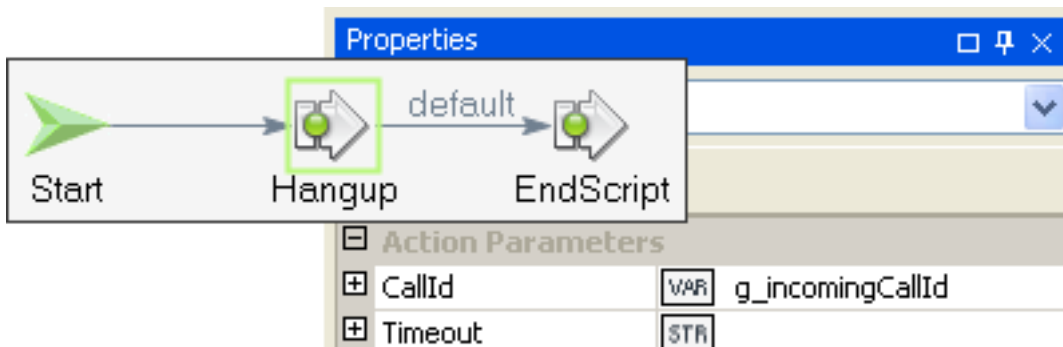


Figure 33: Hanging up the inbound call after outbound call fails

At this point, the application should be able to be built and deployed. Ensure that the **To** parameter value on the *MakeCall* action routes to a real endpoint, so you can test the application.

5. OPTIMIZING CONFERENCING

The previous section showed how to create a conference and add participants to the conference after both connections are created. So, there is some inherent delay added before both parties can hear each other due to the way we place each participant into conference.

From a capabilities standpoint, the Cisco Unified Media Engine can simultaneously create a connection while placing it into a conference. However, because the platform creates connections automatically when making and placing calls, the platform itself would have to leverage this feature of the Media Engine; otherwise we would always have to create conferences as we did in the previous section!

Fortunately, the platform does indeed have this capability: it is simply a matter of telling it to use it. Using the script as the previous example as contrast as well as a starting point for development, this section will show how to optimize conferencing such that it occurs faster and with less overall impact on the system. Note that the sample code for this feature may be found in the *OptimizingBridgingCalls* script contained in the accompanying *ReceivingCalls.max* project.

Let us begin by removing the *CreateConference* action, *JoinConference* action, and *outboundConnectionId* local variable, from the *OnMakeCall_Complete* event handler — we will no longer need them.

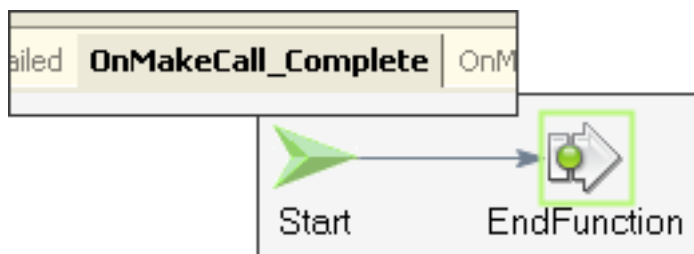


Figure 34: Removing explicit conferencing actions

Navigating back to the *OnIncomingCall* handler, we need to indicate on the *AnswerCall* action that we would like to create a conference while creating the connection; placing that connection into the conference in the process. The **Conference** parameter is the means by which we do just that. **Conference** accepts boolean values, so we specify a value of **true**. See Figure 35 for the example code.

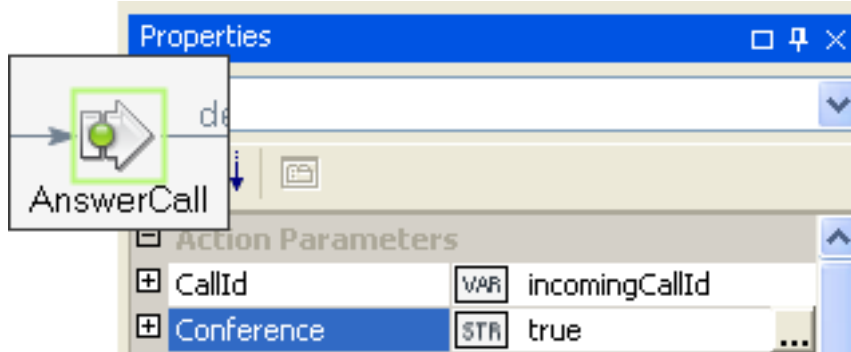


Figure 35: Requesting a conference when answering

Now, when the *AnswerCall* action completes, it will return a newly created **ConferenceId**, returned by the Media Engine. We will need that later, so let us save it to the same global variable we created for this purpose in the previous section, *g_conferenceId*. Figure 36 shows how.

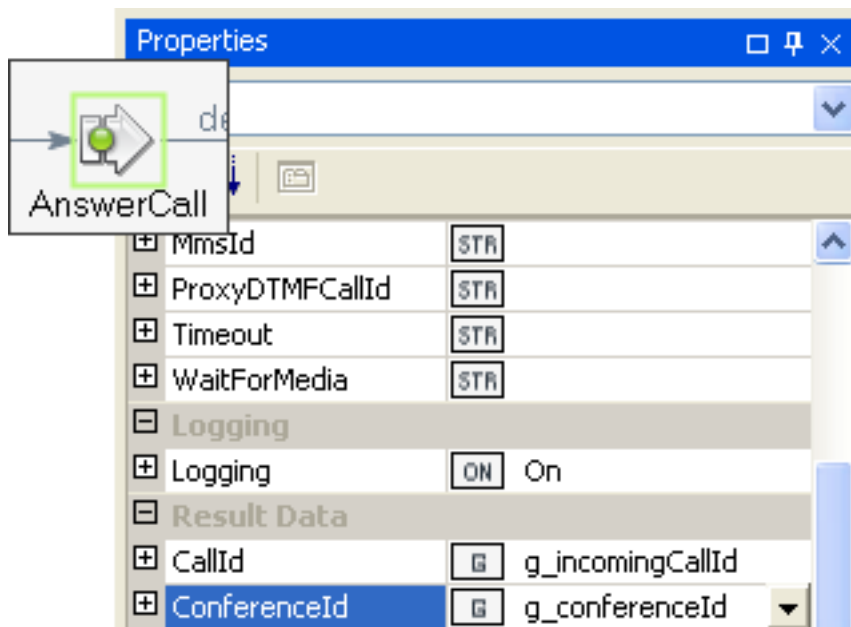


Figure 36: Storing the ConferenceId result parameter

Navigating now to the *OnPlay_Complete* event handler, we now not only specify **true** for the **Conference** action, but we also pass in the already-created **ConferenceId** as shown in Figure 37. In contrast to what occurred when we answered the inbound call, at which time a new conference was created, the *MakeCall* action will negotiate with the Media Engine such that the connection made for the outbound call will be added to the specified conference.

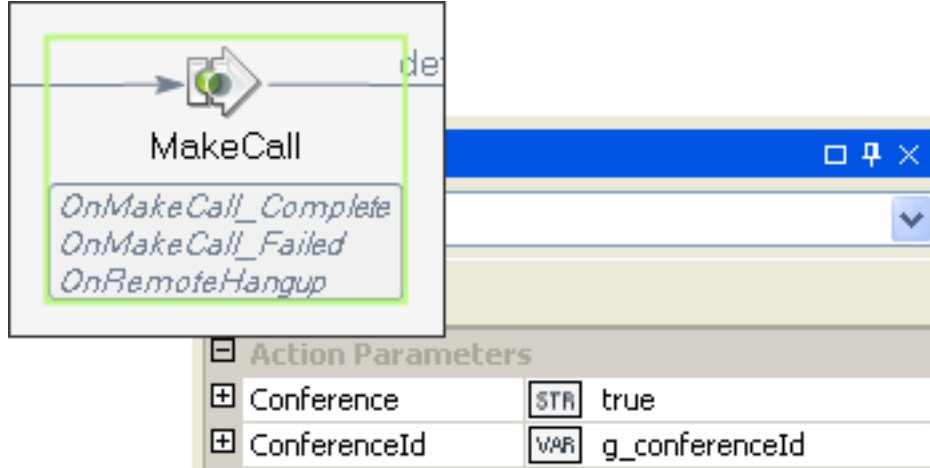


Figure 37: Adding the outbound call to the conference

Now we have the same end result as the previous example, but with less overall negotiation with the Cisco Unified Media Engine, and less initial delay in setting up the conference, resulting in the participants experiencing two-way audio sooner.

6. IN SUMMARY

The purpose of these three examples was to give you a basic understanding of how you might use the *CallControl* and *MediaControl* APIs in your own applications. There is certainly much more to both these APIs; however, the concepts in this document will be instrumental in becoming increasingly familiar with making UAE applications.