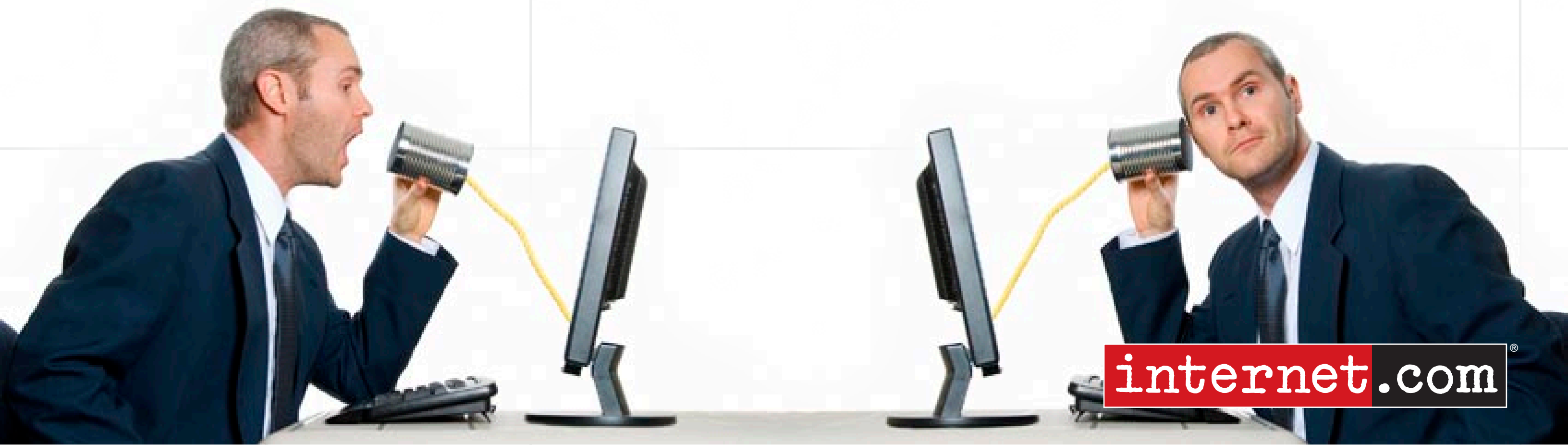
AVAYA

DEVELOPING EXCEPTIONAL MULTI-MODAL CUSTOMER EXPERIENCES



internet.com®

contents

DEVELOPING EXCEPTIONAL MULTI-MODAL CUSTOMER EXPERIENCES

Letter from the Editor...

By Michael Pastore

Quality communication between businesses and customers is an important part of conducting commerce, and has been from the beginning. Customer communication has evolved rapidly over the last several decades to include printed newsletters and catalogs, e-mails and Web sites, and online tools like Twitter.

For nearly 100 years the telephone has been a central part of the interaction between customers and business. The advent of mobile phones, high-speed networks, and accessories like hands-free devices not only ensures that the phone remains critical to communicating with customers, but it opens new doors as well.

Self-service telephone applications that use an approach called Interactive Voice Response (IVR) are relatively commonplace today. This eBook examines how businesses can take self-service applications to a wholly different level by adding speech and video capabilities to the equation, enabling Interactive Voice and Video Response (IVVR). By taking advantage of the ability to include speech recognition, video and advanced call routing capabilities in a self-service voice application, businesses can enhance the experience for users and get a leg up on the competition. IVVR can also help control costs by further streamlining the involvement of human contacts in customer support business processes.

Thanks to powerful development tools and standards like SMIL, VoiceXML, and CCXML, the creation of IVVR applications is easier than ever before. This eBook is going to take you on a tour of the development process used to create such multi-modal customer experiences. We'll start with a look at the business case for developing IVVR applications, written by Mark Miller, the author of several books on networking technologies.

Then we'll start to explore the nuts and bolts of building such applications with articles by Shari Gould and Steve Apiki that discuss Avaya Dialog Designer, an open-standards based Integrated Development Environment for voice self-service applications.

From there, we'll take a look at the structure of a CCXML application and get into writing some simple CCXML documents. We'll also examine some of the best practices for developing speech grammars, which are the key to more natural interaction and a more pleasant caller experience.

To help you out along your journey IVVR applications, we've assembled some resources you'll want to explore.

• More information on Avaya Dialog Designer, including free download of the software, is available to registered DevConnect members through the Avaya DevConnect Portal (www.avaya.com/devconnect – become a member and access the DevConnect portal).

• The Avaya DevConnect Portal offers additional featured articles from the Avaya DevConnect Center on DevX.com, including pointers to sample application code and additional materials.

• More information on Avaya Self-Service Solutions, including professional services that help you plan, design and integrate self-service applications into your contact center operations can be found at www.avaya.com - Enjoy and good luck! ■

The Business Case for Interactive Voice and Video Response

Unless you have been in a remote part of the Yukon and cut off from the rest of the business world for the last year or so, you know that the global economy has turned south, and that companies must work harder than ever these days. And harder can have several meanings.

by Mark A. Miller, P.E., President, DigiNet Corporation

Perhaps your company went through a corporate downsizing, and now there are fewer workers around to help shoulder the load. Or your competition is aggressively cutting its prices to stay afloat, and with that cutting into your share of the market. Or maybe you are searching for *the edge*; that distinguishing factor that gives your firm a leg up on the competition--be that speed of product delivery, improved customer service, reduced customer churn, or optimized cost structure--whatever might keep your firm from becoming a recessionary statistic.

But all business transactions have a common thread: communication with the customer; and if we can optimize that aspect of the business, we are likely to improve some of the other financial challenges as well. While this eBook focuses on broadly enabling self-service applications through technologies such as automated speech recognition (ASR), Call Control XML (CCXML) and VoiceXML, this article will introduce you to a new approach for communicating with your customers, called Interactive Voice and Video Response, or IVVR for short. IVVR can dramatically enhance the customer interaction by adding a new form of communication--video--to the already-proven technologies of voice response services. But that is getting ahead of the story. Let's begin by considering how an interactive response system can provide financial benefits during these challenging economic times.

The Value Proposition

Let's consider a basic assumption:

Automating the customer communication process can lower the cost of customer contact, which can lead to improvements in business financials.

In other words, the more customer enquiry calls that can be handled on a self-service basis, the lower the overall cost to the organization.

For example, let's assume that you need a solution to support change of address updates or PIN reset requests for your customers, estimated at 800,000 updates a year. And let's estimate that automating this with a voice response system has a first year cost of $200,000.00. Let's also assume that a call agent-assisted call costs your company $5, whereas a self-service call costs only $0.25 to implement; and further, let's assume that 80 percent of the time the self-service process is successful in handling the customer's enquiry.

A simple payback period would be calculated by:

$$\frac{\textit{First Cost}}{\textit{Number of Calls/month * cost savings per automated call * percentage of automated calls}}$$

Thus, if we assumed savings of $4.75 per call ($5.00 - 0.25), and the automated system could complete the call 80 percent of the time, we would see a payback period of around eight months:

$$\frac{200{,}000}{800{,}000 * 4.75 * 0.80}$$

This yields a payback period of 0.65 years, or about 8 months.

Not factored into this simple analysis are the side benefits to the customer, such as getting a quick answer to a simple question, or not having to wait on hold to speak with a live call agent. Thus, and perhaps unlike other business investments that might be considered, a voice response system can provide a positive net return in a very short amount of time.

Speech Self-Service Applications

In the last decade or so, we have seen significant improvements in speech self-service applications. The first category could be called informational, where a customer calls a response center needing a stock quote or airline flight status information. The customer enters a few numbers, such as a flight number, on the telephone, a database lookup ensues, and a voice response is returned.

A second category could be called transactional, where the customer wants to verify a credit card balance, buy some stock, or make a flight reservation. These actions are more complicated, requiring interactions with multiple business processes, such as confirmations of flight time, number of passengers, payment information, and so on.

A third category would be called problem solving, which is typical of technical support organizations, where the customer is prompted through a set of decision tree-based questions that attempts to diagnose the problem without engaging the more costly resources of a human. The design of the underlying decision tree makes these problem solving communication systems even more complex.

These three applications would typically use one of two types of customer input, either through touch-tone input using the keypad on the telephone ("Press 1 for your account balance, press 2 to apply for a loan," etc.), or though a short phrase of speech ("schedule a pickup," "track a package," etc.). But note the underlying technology of these existing systems: they are all based on a customer's data and/or speech entry that generate an audible (i.e. voice) response from the system, hence the term Interactive Voice Response, or IVR.

In recent years, the underlying software platforms that deliver voice response solutions have evolved to encompass capabilities for speech biometrics (for user authentication), interactions with back-office systems via web services, and call control capabilities to allow seamless transitions and interactions with live agents or other enterprise contacts. As a result, the IVR platform has given way to the Voice Portal, an application server-oriented solution that provides greater application flexibility and control that simple routing or pre-defined voice responses.

A New Paradigm: Interactive Voice and Video Response

Remember your first cell phone? Are you willing to swap your new Blackberry or iPhone for one of those old dinosaurs? Of course not–you have become accustomed to a full keyboard that allows you to send text messages, a color display that lets you see a GPS map to the your destination, or access information at 3G speeds, almost like you were in front of your office PC.

So the next logical question might be: given the availability of this technology, why should a customer response system be limited to just data and voice? Why not add a visual component, because after all, a picture is worth a thousand words? And if you recognize that Bluetooth allows you to use the headset to send and receive the audio rather than the phone body itself, it means you can now hold your mobile device in your hand and simultaneously look at the display while still exchanging audio information (instead of having to hold the phone, and its display, up to your ear). Used in this way, your system is now both voice and video enabled–hence the concept of Interactive Voice and Video Response.

To further your imagination, consider other ways that self-service solutions can be utilized, from in-store kiosks that allow customers to speak with remote customer service agents (or 'bots, as the case might be with an automated self-service application). While in-store personnel are busy serving other customers, a customer can interact with an IVVR-enabled self-service application that not only responds to their queries, but pushes visual information to a media-enabled IP telephone or kiosk display, such as maps directing them to the right store aisle for a specific replacement part, or allowing them to order items to be shipped from a central warehouse that may not be available in the store at the momen–and showing them appropriate product images to help them select color or style.

Avaya has created a demo of IVVR, using an example of an Internet-based florist called Bloomin' Blooms who has implemented a coordinated voice and video customer response system, which allows customers to see images of flower arrangements before they place their order, plus creates an upsell opportunity by offering the customer a beautiful vase to go with those flowers. Check out the YouTube link to see this demo in action.

Enabling Standards

The IVVR application is based upon two key technologies: the Voice Extensible Markup Language (VoiceXML) and the Synchronized Multimedia Integration Language, or SMIL (pronounced "smile"). Both of these technologies have been developed by the World Wide Web Consortium (W3C, or www.w3c.org), which develops software, protocols, and standards to promote the interoperability of Web applications.

The first standard, VoiceXML, specifies a format for the verbal interaction between a human and a computer. This architecture employs a voice browser that presents the information in audio format, such as a playback file, which is somewhat analogous to a Web browser in that it presents visual information to end users. The second standard, SMIL, defines an XML-based language for writing multimedia presentations that include streaming audio, streaming video, images, text, and so on. The language can be used to create training courses, product presentations, or multimedia events for Web distribution, and includes parameters to specify content, timing and synchronization, animations, visual transitions, and so on.

Putting It All Together

But you can't just take a neat idea, integrating voice and video for customer response, add in some standard technologies like VoiceXML, CCXML (for call control) and SMIL, and make it happen–you need a development platform that can integrate all these technologies to make them effective. That platform is Avaya Dialog Designer, a tool for streamlining VoiceXML speech and video self-service application development.

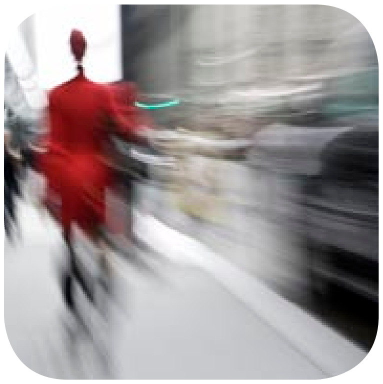
Avaya Dialog Designer was developed to meet several objectives. First, it utilizes a common tool framework that is powered by Eclipse, the open source development platform (see www.eclipse.org). Second, it supports lightweight development and deployment, meaning that developers can install it on their own PC with built-in simulation that allows testing without a platform. Third, it supports industry standards, including Java, J2EE (Java Platform Enterprise Edition), SOAP (Simple Object Oriented Access Protocol), VoiceXML, CCXML and others. Finally, it meets the needs of both an IVR developer, and an IT buyer or systems integrator; rapidly building applications using drag/drop tools.

Dialog Designer includes a five-step process for application development. First, the end user builds call flow, caller prompts, and grammars using an intuitive, graphical user interface. In the second step, those structures are sent to another process that generates the application code. Third, this code is then simulated using an embedded VoiceXML browser and evaluated to determine if the call processing application has been characterized correctly. Fourth, a deployment wizard packages the application for the voice portal platform, and moves the packaged application to a J2EE-compliant application server. Fifth, the VoiceXML code is processed on the customer's interactive response or voice portal, which runs the new application.

In summary, Avaya's latest release of Dialog Designer provides the industry's first full-cycle application development tool, which includes video simulation, mixing touchtone, pre-recorded, and dynamic text-to-speech, video, speech recognition, VoiceXML, CCXML, and SMIL, plus has the capability for connecting to back end systems and data sources via Structured Query Language (SQL) and web services. It also provides a simplified deployment model, which can be integrated with the Avaya Voice Portal. The move to a more application server oriented approach with Avaya Voice Portal also opens the door for outbound self service applications, including both informational and transactional services. Imagine how you could wow your customers by proactively reaching out to them with both voice and video--now you won't just confirm delivery of those flowers, you'll actually let them see the look of joy on the recipients face with a coordinated video clip!

Subsequent chapters in this eBook will explore these capabilities in greater detail. But for now, turn your imagination loose, and consider how your company might better weather this current economic storm by ratcheting up your customer contacts with an interactive voice and video response. ■

Mark A. Miller, P.E. is President of DigiNet Corporation®, a Denver-based consulting engineering firm. He is the author of many books on networking technologies, including Voice over IP Technologies, and Internet Technologies Handbook, both published by John Wiley & Sons.



Going Against the Flow: Avaya Makes Voice Application Development Look Easy

See how enterprise developers can look like they have been coding telephony-based, self-service applications for years—with tools and a few tricks of the trade.

by Shari L. Gould

Both novice and seasoned developers can create voice applications for their company's telephony platforms. Novice developers frequently rely on integrated development environments (IDEs) to create applications for a variety of industries and to run on one or more platforms, be they UNIX, Linux, Mac OS X, or Microsoft Windows, to name the top tier operating systems. More experienced developers sink their teeth into the code behind the IDE and can troubleshoot problems should an obscure error occur. Regardless of the level of expertise, enterprise developers without telephony experience can get ahead on the learning curve with Avaya Dialog Designer IDE.

Regardless of the level of expertise, enterprise developers without telephony experience can get ahead... with Avaya Dialog Designer IDE.

In addition to abstracting the details of connecting to a telephony interface the Dialog Designer provides support for open standards like VoiceXML, Web services, Speech Recognition Grammar Specification (SRGS), and Java 2 Enterprise Edition (J2EE). It also provides ease of designing Voice User Interface (VUI) through a graphical drag and drop interface.

Built on the extensible architecture of the Eclipse open source platform, Dialog Designer provides in-built connectors for supported speech recognition engines, making it simple to add speech to applications via a framework of development tools. Dialog Designer stays true to its object-oriented design by offering modules that allow for efficient reuse of code by other applications. Further, Dialog Designer's loose coupling separates call flows and application flows from language elements, allowing for easy localization of applications.

Getting Started

Dialog Designer's deployment environment requires your company to have a Web server platform running either Apache Tomcat 5.0 or higher, IBM WebSphere Express 6.0, or IBM WebSphere Application Server 5.1.1 Web server software; and a high-performance Web server architecture.

Here's what you need to get started:

- Dialog Designer from Avaya, Eclipse, and other components.
- A development machine with Windows 2000 or Windows XP operating system.
- An Interactive system, if you want interaction between the Dialog Designer application through a telephony interface.

Avaya's CD includes the Eclipse platform, Dialog Designer, and other critical design and development tools such as an embedded VoiceXML browser to allow for fast simulation and debugging of speech-enabled applications. Although it is not required to run Dialog Designer, Avaya's Voice Portal and Interactive Response telephony platforms share the VoiceXML browser for consistent and reliable deployment as well as dynamic generation of VoiceXML and speech and touch-tone grammars.

Using the Development Environment and Templates

Developers that are familiar with any modern IDE can generate voice applications with the simple drag and drop technique. Developers who are more experienced can modify your voice applications via Dialog Designer's Console window, as you can with many leading IDEs.

Wizards in the Development Environment allow you to quickly create and integrate Web services within dialog call flows. The Development Environment also allows you to test your call flows on the fly using its simulator with built-in VoiceXML browser.

Within the Development Environment, you can select, configure, and link application templates, building reusable components for automated voice services and call flows. You also can create multilingual voice applications because of Dialog Designer's separation of call and application flows from language elements.

You can choose a template to help design the call flow you want. Here are some basic template options within Dialog Designer:

- Announce: used to create an initial announcement the caller hears
- Prompt and Collect: the application prompts the caller and collects data from him or her
- Several transfer options
- Record: used to record caller input (For example, Record allows end users to record a message to be delivered within the company's voicemail system.)
- Disconnect: delivers a message to the caller prior to disconnecting the call

If you are a more daring and experienced developer, you can create your own call flow without the aid of a template using Dialog Designer's Application Items. Under the Application Items palette, you can select an option to create a main menu of choices for the caller, create a template from the Form option; and use the Data option to manage variables, application data, databases, and Web services interfaces for computer/telephony integration (CTI). Dialog Designer allows more advanced developers to add servlets, or to drop in Voice XML servlets to further extend the functionality.

Creating a Call Flow

Dialog Designer allows you to manage projects in a workspace. As with most mainstream IDEs, Dialog Designer's graphical use interface (GUI) uses standard expandable and collapsible trees on the left of the window in the Navigator Window. In the right side of the window, the palette displays the call flow as you create it. Using the call flow builder, you can create a speech project and collect input for a call flow, such as setting the prompt name and grammar names.

Here's what you need to know to get started with a basic voice application.

For a basic call flow design, you as a developer need to know in what order each question and response should go. Heads up, though—for more complicated applications like a call center application, a little telephony experience may come in handy when designing a call flow.

Let's start with creating a basic call flow with speech and touch-tone grammars. Let's say you want to create an application in which an automated system answers the telephone with a pleasant greeting when someone calls your company. The caller hears a menu of options from which he or she chooses and the call is routed automatically to the appropriate place. A series of questions from the automated system and responses from the caller will help with directing the call.

Use the Flow folder in the Navigator Window to create a main menu for the primary call flow, called main.flow by default. Next, set the prompt name in the Prompt Editor. When you create a prompt, you are telling the system what announcement it should deliver to callers.

The next step is to set grammar name, or the list of menu options from which the caller will choose. Grammars can be defined as static or dynamic (at run time), or your company can purchase lists of grammars from third-party sources if specific menu options are required.

Handlers need to be identified next, so the system will know how to respond if the user offers no input or response, or if the caller's response does not match any option in the menu. Dialog Designer uses a text-to-speech (TTS) engine to interpret what the caller says and repeats the caller's response back to him or her. For example, the automated system replies back to a caller, "You said, 'Yes'." With Dialog Designer's VoiceXML capabilities, the IDE supports the specifications for the language defined as the standards for voice interaction with end users. In much the same way a Web browser retrieves HTML from a server, the voice browser retrieves VoiceXML data from a voice server. The voice browser then works with the text-to-speech information using a speech recognition engine.

Finally, use the Connection tool to link the call flow. This tool connects the nodes in the call flow from the starting node to the exit node.

Simulating and Deploying Applications

When you have designed, developed, and saved your call flow, Dialog Designer automatically checks the syntax, providing built-in error detection. You can use the Simulator to test your call flow to ensure it responds as you designed and developed it to perform. The simulator allows you to test your applications in real time as you develop your application. The voice browser is built into Avaya IR and Voice Portal, and works with Dialog Designer in Simulation mode.

In Simulation mode, click the Start Call button. The Console window shows the VoiceXML that is generated and processed dynamically, and Java servlets are generated that run on application servers that host the servlets, creating VoiceXML rendering of Java.

When the voice application you created performs to your specifications, you can deploy it to your company's telecom platform, be it Avaya Voice Portal, Avaya IR, or another telephony platform. You will need to point the IVR system to your company's VoiceXML-compliant browser.

You will also need to deploy the Application Runtime Environment that Dialog Designer creates to your company's Apache Tomcat or IBM WebSphere Java Servlet environment.

Differentiating Factors

Dialog Designer sets itself apart from its competitors in several ways.

First, localization of voice applications is made easy, because Dialog Designer was architected with loose coupling between call and application flows and language elements. Also important, Dialog Designer dynamically generates VoiceXML.

Dialog Designer's modular design makes it easy to use regardless of your level of expertise. The ability to create Java servlets and incorporate Web services allows you to implement the latest solutions for gaining a competitive edge. It also allows developers to integrate voice applications with JDBC-compliant databases as well as support Web services.

Both novice and seasoned developers can create voice applications for their company's telephony platform. Having an IDE to create voice applications without having a telephony background gives organizations and their developers a competitive edge. Regardless of your level of telephony expertise, you as an enterprise developer can get ahead on the learning curve with Avaya Dialog Designer IDE. ■

Shari L. Gould has more than 16 years of journalism and technical writing experience. Shari has written for numerous leading publications throughout her career, most recently Software Development Times and its various publications, and had an article hand picked by Sun Microsystems for inclusion in its Solaris Developer Connection. She also has more than 10 years experience working with high-tech companies documenting everything from network designs and installations, through software design and APIs, to user interfaces. Shari currently is pursuing her Master's degree in Criminal Justice, specializing in Information Security.

Avaya Contact Centers

You have six incoming calls, one high-priority client on the line, and 20 seconds to make his day.

Contact Center Express

Connect the right customers to the right agents.

Home Agent

Work anywhere, anytime.

Self Service

Help customers help themselves.

AVAYA

INTELLIGENT COMMUNICATIONS

Visit **www.avaya.com/CC** to see how Avaya simplifies business communication.

Speech Sandbox: Application Simulation in Avaya Dialog Designer

Avaya Dialog Designer's built-in application simulator gives you the freedom to test, tweak, and innovate outside of a production self-service environment.

by Steve Apiki

Avaya Dialog Designer can take your VoiceXML (VXML) or Call Control XML (CCXML) application all the way from concept to fully-tested deployment—and it's every bit as strong in the later phases of the development cycle as it is in up-front design. The key to Avaya Dialog Designer's test and debug prowess is Avaya Application Simulator, a built-in voice browser and simulator that allows you to simulate calls from the Dialog Designer IDE. With the application simulator, you can test DTMF or speech-enabled applications without interfering with production Avaya Voice Portal or Avaya Interactive Response (IR) platforms. After testing, you can easily deploy completed applications over to production servers. Avaya Application Simulator fully integrates testing in the development process, not only encouraging thorough testing but also fostering iterative development.

Simulation is just one of the capabilities of Avaya Dialog Designer, an Eclipse-based IDE for creating speech-based self-service applications including those using VXML and CCXML. With Avaya Dialog Designer, you can build self-service applications such as reservation systems or account information services and later deploy them to either the Avaya Voice Portal or Avaya IR. Although we'll focus on test and simulation here, the IDE also includes a graphical flow editor (Figure 1) for designing user interactions and connection wizards for accessing web services, JDBC data sources, CTI using Avaya Application Enablement Services, and services offered by Avaya Interaction Center.

You can download a CD image that includes Avaya Dialog Designer, Eclipse, and all other prerequisites from Avaya DevConnect (www.avaya.com/devconnect; free registration is required). The CD includes a number of sample applications, one of which we'll use to illustrate the simulation features of Avaya Dialog Designer. The DevConnect site also includes full documentation for Avaya Dialog Designer, including a Getting Started document and a Developer's Guide.

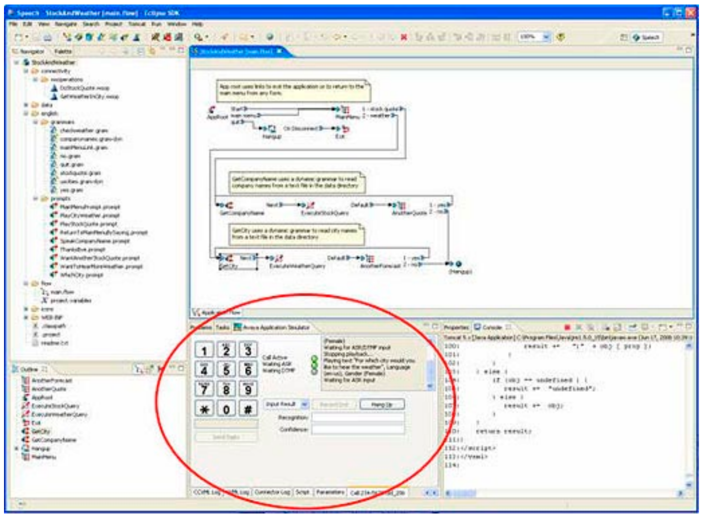


Figure 1. Avaya Dialog Designer is built on the Eclipse platform and includes specialized editors for building self-service applications. The application simulator pane is at the bottom, in the center (indicated by red circle).

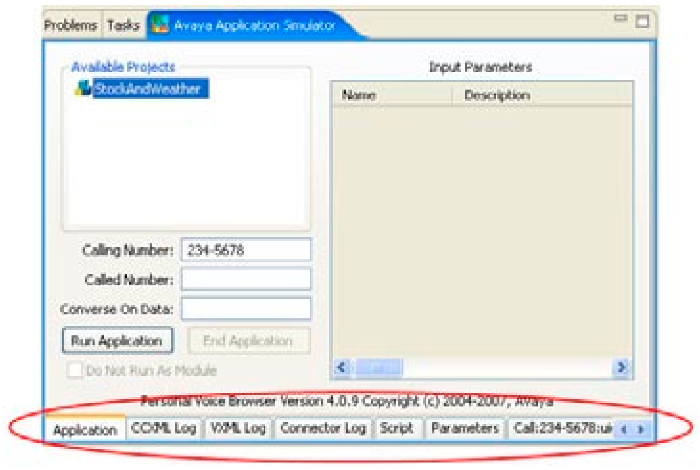
Focusing on Simulation

A Dialog Designer VXML or CCXML application is structured much like an HTML web application, with a web server/servlet container hosting Servlets and serving them up to a browser. In production, those roles might be filled by an IBM WebSphere, BEA Weblogic or Apache Tomcat server hosting Servlets to an Avaya Voice Portal voice browser. VXML is then rendered dynamically by the Avaya Voice Browser.

Avaya Dialog Designer replicates this stack in a self-contained development environment. In Avaya Dialog Designer, you use the integrated design tools to generate Java servlets which are then served by Tomcat (under the control of the IDE) to the integrated voice browser. The voice browser is common to Avaya Dialog Designer, Avaya Voice Portal, and Avaya IR. In simulation, you drive the integrated voice browser using simulated inputs, including DTMF tones and basic speech recognition through the Microsoft SAPI-based Automatic Speech Recognition (ASR) and Text-to-Speech (TTS) engines.

Figure 2 shows the Avaya Application Simulator interface. The screenshot shows the Application tab, where you can start a simulation run. There are a number of additional tabs arrayed across the bottom. Each tab may control a simulation feature or display simulation results. These are the functions of each of the tabs:

- Application: This is the starting tab. On the Application tab, select the application to run and set startup parameters. Startup parameters, which are optional, include a calling number (for ANI or automatic number identification), a called number (for DNIS or dialed number identification service), and simulated Converse On data (in production, this would come from a call center switch).
- CCXML Log: Displays CCXML log messages from the application simulator.
- VXML Log: Displays VXML log messages from the application simulator.
- Connector Log: Applications in AvayaDialog Designer can interact with external computer telephony integration (CTI) servers such as Avaya Application Enablement (AE) Services and with Avaya Interaction Center (IC). Applications interface with these systems through connectors in Avaya Dialog Designer. The connector log tab shows log messages from CTI and IC connectors.



Calling Number: 234-5678
Called Number:
Converse On Data:
Personal Voice Browser Version 4.0.9 Copyright (c) 2004-2007, Avaya

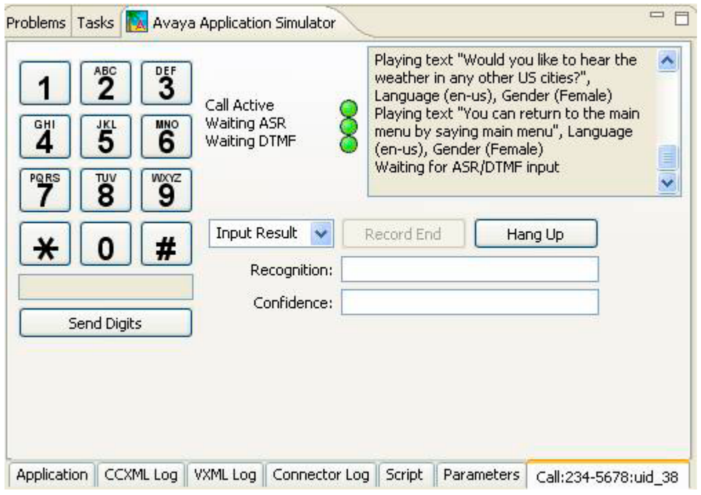
Figure 2. The Avaya Application Simulator view. Here, the Application tab is active. The Application tab is where you can set startup parameters and start and stop the application.

- Script: Caller responses (DTMF and speech recognition) may be scripted. You can use scripts for regression testing, or in debug mode to move a call along to an interesting position. Avaya Dialog Designer also supports a second type of script, used to simulate connector actions. With connector scripts, you can test CTI and IC connector applications without access to real servers. You write both types of scripts as external XML files. The script tab lows you to manage these scripts.
- Parameters: Allows you to specify parameters to be passed to the application under test in a number of categories, including call control and call classification. In a production system, these parameters would be set by Avaya Voice Portal or Avaya IR.
- Call: Avaya Application Simulator creates a call tab once the application is started. The Call tab includes a keypad for generating DTMF input, a Hang Up button, and call status information. This is the main interface you'll use while a simulation is active, and we'll discuss it in some detail in the following section.

Stocks and Weather

Stocks and Weather is one of the sample applications supplied with Avaya Dialog Designer (and even more are available via the DevConnect website). It's a VXML application that prompts the caller for DTMF or voice inputs, and then uses those inputs to look up stock or weather data through an external web service. We'll walk through a simulation using Stocks and Weather as an example.

To start the simulation, we select Stocks and Weather on the Application tab and click Run. This starts Tomcat if it's not already running and then launches the voice browser. As the call starts up, Avaya Application Simulator creates a Call tab (Figure 3). Since we specified a calling number on the Application page, that number is shown along with a call id number on the tab label. Stocks and Weather doesn't use the calling number, but another application could use the ANI data, for example, to help authenticate the caller.



Playing text "Would you like to hear the weather in any other US cities?", Language (en-us), Gender (Female)
Playing text "You can return to the main menu by saying main menu", Language (en-us), Gender (Female)
Waiting for ASR/DTMF input

Figure 3. The Call tab is used to control the simulation.

At the start of the call, Stocks and Weather asks the caller if he or she wants a stock quote or weather information. This prompt plays through the workstation speakers. On the Call tab, the progress display (upper right) shows the text that the voice browser just played, and the Call Active indicator glows green to show that a call is in progress. The Waiting ASR and Waiting DTMF indicators are also green, showing that the application is now ready to accept speech or DTMF input.

In the application flow, that starting prompt is attached to a menu node that will select the next node based on the caller's response. Avaya Dialog Designer highlights this node in the flow diagram. As the call proceeds, each active node is highlighted in turn, making it easy to follow the flow of a call in simulation.

To continue the call, we can click "2" on the keypad and then the Send Digits button to send DTMF input to the application. The 2 corresponds to the "weather" choice, so the call continues down that path, next prompting for the city name. There are several ways to supply a speech response. First, the simulator accepts voice input through the microphone, so you can just say the city name ("San Francisco"). Second, you can type the city name into the Recognition box in the call tab, optionally specifying recognition confidence in the Confidence box. Or finally, you can send No Match or No Input responses directly by selecting those conditions in the Input Result drop down.

Stocks and Weather uses the city name to look up the weather, sending a SOAP query to a remote web service. Avaya Dialog Designer supports connections to web services but doesn't simulate them, so you'll need to have a real connection to fetch data from a web service when you run the application in the simulator. Similarly, Avaya Dialog Designer supports connection to a JDBC data source, but this, too, is not simulated. You will either need access to a real external database or need to create a test database server on the development machine.

Stocks and Weather completes the query and then reads the response (the weather for San Francisco). At this point we can continue to navigate through the menus using the Call tab, eventually completing the call using the Hang Up button.

You can also run CCXML applications through Avaya Application Simulator. CCXML applications may create additional calls (for example, a find-me/follow-me application might dial several contact numbers from a list). The simulator creates a Call tab for each active call so you have control over all the calls in the simulation. (The UI for incoming calls looks slightly different, allowing you to choose to answer the phone).

Making it Real

When your Avaya Dialog Designer application has been fully tested, you can export it to a WAR or EAR file for deployment to the production server. Deployment is a one-way process—you don't round trip changes from production back into Avaya Dialog Designer. Instead, you continue to revise and make changes to the project in the IDE, and re-deploy as each round of revisions is complete.

Avaya Dialog Designer adds speech project (VXML) and call control project (CCXML) export types to Eclipse. To deploy an Avaya Dialog Designer application, you first choose one of these project types and then work your way through the export wizard, specifying options such as the target platform (e.g., Avaya Voice Portal) and the target web server (e.g., Tomcat). When you complete the export wizard, Avaya Dialog Designer packs the application into a WAR file (for Tomcat) or an EAR file (for WebSphere), ready for deployment as a live application.

On its own, Avaya Application Simulator would be an interesting tool. But it is hard to overstate the value that its tight integration with the rest of the Avaya Dialog Designer IDE brings. Integration means that you can build applications iteratively, instantly seeing the results of changes. It makes Avaya Dialog Designer a self-contained system that can be installed on a laptop and brought home, or brought on the road. And it gives you the freedom to try new ideas and new applications without touching a production server. ■

Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

CCXML: Powerful, Standards-Based Call Control

Take a high-level approach to call control with Call Control XML and bring new capabilities to self-service applications.

by Steve Apiki

CCXML can transform a self-service application from a passive provider of information to an active hub connecting callers to data, to applications, and—most interestingly—to other people. As the call-control framework for Voice XML (VXML) applications, CCXML (Call Control XML) is designed to support the "people-to-people" features that VXML lacks, including bridging, multi-party conferencing, and outcalling.

CCXML is a markup language that describes call control, just as VXML is a markup language that describes voice dialogs. The two languages are complementary, and can be used independently. Both are intended to be served up by web servers as documents to be interpreted by a voice browser such as Avaya Voice Portal. CCXML 1.0 is currently a Working Draft of the W3C, the body that publishes and maintains web-related standards.

When VXML and CCXML are used together, CCXML is typically at the front end, with VXML dialogs providing user interaction. The CCXML interpreter is responsible for handling calls. It takes action based on dialog responses, or asynchronously, in response to external events. Unlike VXML, CCXML applications can manage more than a single call leg. A CCXML application can deliver features like multi-call conferencing, transfer from IVR to live help or to a call recipient (and back), and whisper transfer, none of which is possible with VXML or easily accomplished with legacy IVR systems.

In this article we'll take a look at the structure of a CCXML application and get into writing some simple CCXML documents.

The Structure of a CCXML Application

The CCXML application model is similar to the familiar web programming model. In both cases there are two primary components, a web server and a browser. The web application server runs a server-side telephony web application that delivers CCXML documents to the voice browser in response to HTTP requests (Figure 1). A CCXML application is a set of related CCXML documents that are interpreted by the voice browser.

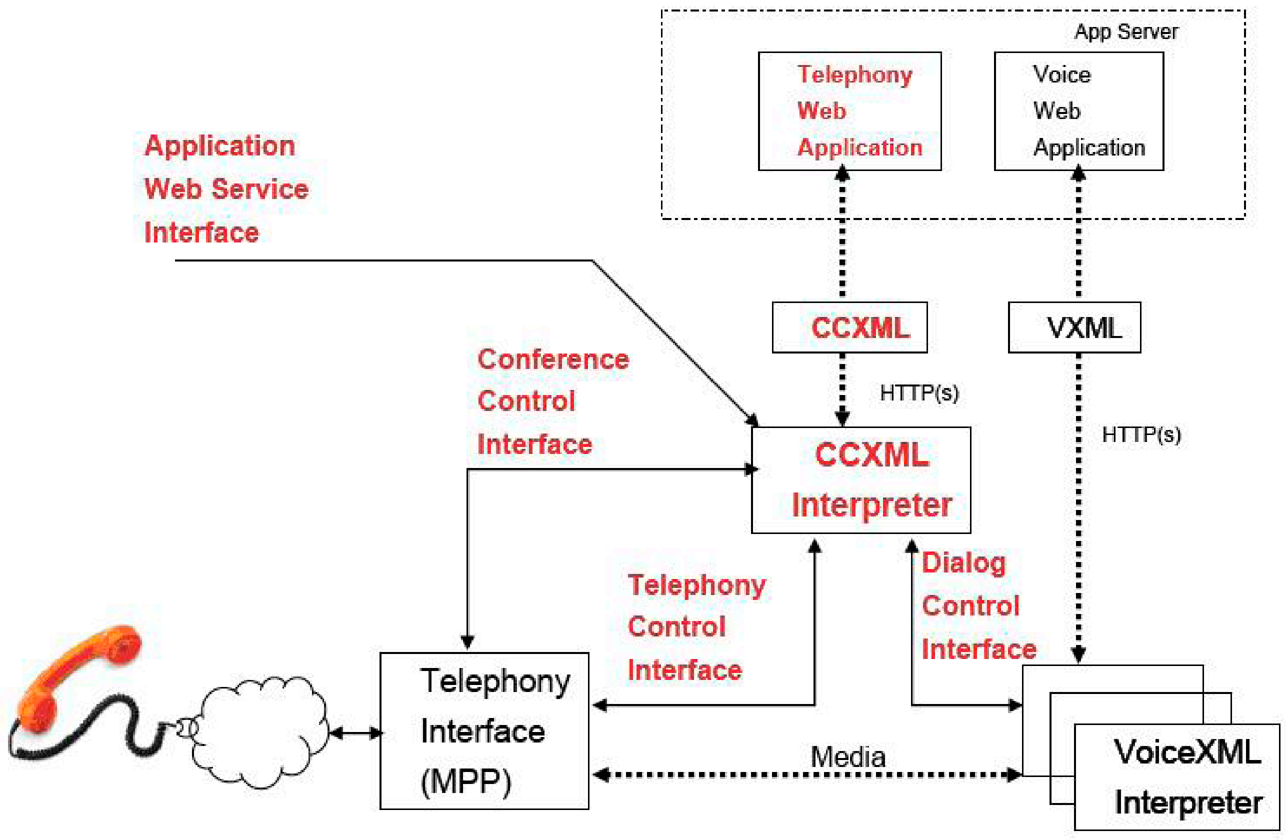

App Server
Telephony Web Application
Voice Web Application
Application Web Service Interface
CCXML
VXML
HTTP(s)
HTTP(s)
Conference Control Interface
CCXML Interpreter
Telephony Control Interface
Dialog Control Interface
Telephony Interface (MPP)
Media
VoiceXML Interpreter

Figure 1. CCXML architecture. The CCXML interpreter becomes the controller for the application, with one or more VXML interpreters optionally providing user dialogs. This is a logical model; Avaya Voice Portal would encompass the CCXML Interpreter, the VXML Interpreter and Telephony Interface blocks along with the control interfaces between them.

A CCXML application can be launched using one of several methods, but one common way is in response to an incoming phone call. When the voice browser receives the call, it determines the starting URL of the application and loads the initial CCXML document. From that point on, the CCXML application has control of the incoming call and can take actions with that call or respond to call-related or external events.

An incoming call is represented in CCXML by a Connection object. CCXML applications work primarily with three kinds of objects: Connections, Conferences, and Dialogs. A Connection often represents a call leg, but as an abstraction, a connection can be thought of as an object with a single audio input and a single audio output. A conference object can be thought of as having multiple audio inputs and a single audio output that mixes all of its inputs together. A dialog represents a voice dialog, most often driven by a VXML interpreter rendering a VXML document.

At the risk of oversimplification, most of the actions taken by a CCXML application revolve around making and breaking connections among these three types of objects. CCXML applications present a caller with a voice dialog by linking a Connection with a Dialog; bridging two calls requires joining the inputs and outputs of two Connections; and an application builds a conference by connecting multiple Connections to a single Conference object. These high-level abstractions are part of the power of CCXML, as they free developers from having to implement complex call control actions at the API level.

Events and States

CCXML was designed to handle the asynchronous events inherent in telephony. Like GUIs and many other modern programming environments, CCXML is event driven. The bulk of a CCXML document is a series of event handlers that the interpreter invokes depending on the type of event received by the CCXML session and (optionally) on the current state of the session. Virtually all of the work in a CCXML application takes place inside these event handlers.

Here is a CCXML Hello World:

```
<?xml version="1.0" encoding="UTF-8"?>
<ccxml version="1.0" xmlns="http://www.w3.org/2002/09/ccxml">
<eventprocessor>
<transition event="connection.alerting">
<log expr="'Hello World.'"/>
<exit/>
</transition>
</eventprocessor>
</ccxml>
```

When this document is processed in response to an incoming call, it logs the "Hello World" string and then exits. This simple (but complete) application includes an `<eventprocessor>` element with a single `<transition>` element. The eventprocessor element is the container for transition elements; each transition element represents a single event handler. In this case, the handler is called when the application receives a connection.alerting event (that is, when the incoming call is received). Non-trivial CCXML documents will contain a number of transition elements (event handlers) within the eventprocessor container.

CCXML applications may receive events generated by changes in connection state, by dialogs, in response to executing elements within an event handler, or from a variety of other conditions. In fact, CCXML sessions may define arbitrary events and send these to other sessions.

These are a few of the more common CCXML events:

- connection.alerting
  Sent when an incoming call is received.
- connection.connected
  Sent when a call is initially connected.
- connection.disconnected
  Sent when a call is disconnected either by the caller or by the application.
- dialog.exit Sent when a voice dialog terminates.
  You can retrieve values from the dialog from the fields of this event.

A large CCXML document with more than a handful of transitions can quickly become hard to manage. One way to better structure a large document is to consider the eventprocessor as a finite state machine, with each transition element representing a state transition. CCXML supports associating a state variable with the eventprocessor element which allows you to track the session's current state, as shown in the following CCXML fragment:

```
<var name="sessionstate" expr="'initial'" />
<eventprocessor statevariable="sessionstate">
<transition state="initial" event="connection.alerting">
<accept/>
</transition>
<transition state="initial" event="connection.connected">
<assign name="sessionstate" expr="'dialog _ running'" />
<dialogstart src="'dlg.vxml'" />
</transition>
```

This eventprocessor is associated with a state variable named sessionstate. When the document is initialized, sessionstate is assigned the name of the starting state ("initial"). When an event is received, the state variable is used as part of the match criteria to determine which transition should be selected. The transaction that handles the connection.connected event launches a VXML dialog and moves the session to the next state ("dialog_running"). Figure 2 shows the associated (partial) state diagram.

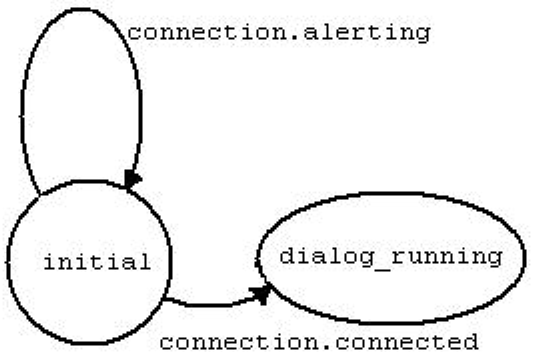


Figure 2. The state diagram for the state variable CCXML snippet.

Common Tasks

Let's expand the state variable example above to come up with a simple CCXML application that connects an incoming call to a voice dialog. When the dialog exits, the application terminates. The full state diagram is shown in figure 3.

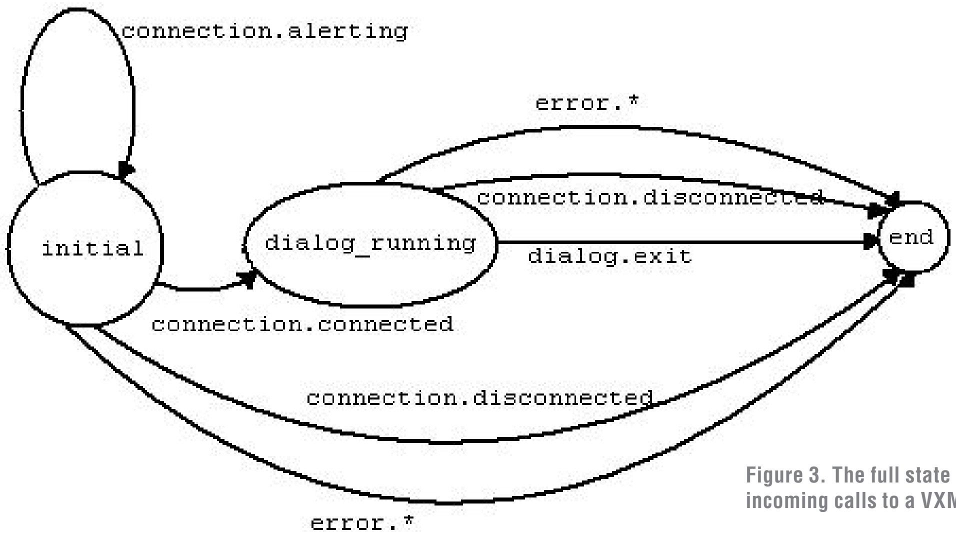


Figure 3. The full state diagram for a CCXML application that connects incoming calls to a VXML dialog.

To connect the caller to a VXML dialog, we use the dialogstart element, and give the URL of the VXML document as the src attribute:

```
<dialogstart src="'dlg.vxml'" />
```

Again, this is a simple example. There are a number of other attributes that can be used for finer control of the dialog's behavior. One important point here is that dialogstart is non-blocking; it starts a VXML interpreter on a new thread and uses the new interpreter to process the VXML document. This means that the CCXML interpreter is free to go on and to respond to other events.

As you can see from Figure 3, we need to add a few additional transitions to get from the states we already have defined to the end state. When the dialog exits, we want to read back the dialog information and exit, so we add this transition:

```
<transition state="dialog _ running" event="dialog.exit">
<log expr="'Caller said:' + event$.values.response" />
<exit />
</transition>
```

The dialog.exit event handler is where a CCXML application can read values back from a voice dialog. The event object visible inside the transition is an ECMAScript object with attributes that vary according to the event type. (ECMAScript is the scripting language that is used with CCXML, the standard language of which JavaScript is an implementation.) A dialog.exit event has a values attribute which is itself an ECMAScript object which can contain information to be passed from the dialog to CCXML.

If the caller hangs up, we also want the application to exit. We only expect this event to occur in the dialog_running state, but since we always want to exit on this event, we can write a transition that matches the event in any state by leaving out the state attribute.

```
<transition event="connection.disconnected">
<exit/>
</transition>
You can also set a transaction to match multiple events using wildcards. As shown in the state
diagram, we'll handle all error events by terminating the application:
<transition event="error.*" >
<exit/>
</transition>
```

As a final example of what you can do with CCXML, let's move beyond simple IVR and look at how you might place an outgoing call (this might be part of a find me forwarding service).

To place the outgoing call we use the createcall element:

```
<createcall dest="'tel:5551212'"
connectionid="out _ connectionid"/>
<assign name="sessionstate" expr="'calling'" />
```

`out _ connectionid` is a variable that receives the identifier for the new connection. Once we've started the call, we move into a calling state where we can respond to connection events (both successful and unsuccessful).

On a successful connection event, we can bridge the incoming call to the outgoing call using the CCXML join element:

```
<transition state="calling" event="connection.connected">
<join id1="in _ connectionid" id2="out _ connectionid" />
<assign name="sessionstate" expr="do _ join" />
</transition>
```

Using join in this way creates a bridge between the two call legs, essentially connecting the incoming caller with the outgoing callee. Although the connections are bridged, the CCXML application is not out of the picture. It still maintains full control over these connections and can respond to events from these connections by taking additional actions. In a full find me application, we would use these call control techniques along with VXML dialogs to guide the caller. We would also need a mechanism for looking up a list of numbers to call. This could be either a database lookup by the server or, because CCXML has the ability to route events to external services, from a web service. With either method, CCXML's high-level programming model naturally incorporates external data into making decisions related to call control.

As these simple examples illustrate, CCXML is a valuable abstraction from network protocols that gives developers the ability to add full telephony features to a web-based VXML application. In part two of this series we'll get down to the nuts and bolts of building CCXML applications in Avaya Dialog Designer. Avaya Dialog Designer provides a graphical, Eclipse-based environment that integrates VXML dialog design with CCXML control. Using Avaya Dialog Designer, we'll work our way through an auto attendant application and explore more advanced CCXML features. ■

Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Developing Speech Grammars That Rock, Part 1: Best Practices

Flexible grammars are key to more natural interaction and a more pleasant caller experience. In the first installment of this two-part series on speech grammars, we cover the basics of grammars and outline some grammar development best practices.

by Steve Apiki

Done right, flexible, efficient grammars can make speech applications feel more like real speech, and less like the verbal equivalent of a game of "Simon Says" ("Press 'one' for a list of sandwiches," "If you'd like a slice of cheese, say 'cheese'," and so forth). Creating a grammar is seemingly easy, but getting a complex grammar right—building a grammar that responds quickly, and in ways that callers expect—is an iterative process that has its share of pitfalls. In this article, we'll focus on the "pitfalls", introducing speech grammars and then describing some critical best practices that can help keep your grammar development on track. In part two we'll come back around to the notion of "iterative" grammar development, working our way through the tuning workflow for grammars, and covering tuning techniques.

Grammar Development

In a speech application, a grammar is a set of rules that define the universe of words or phrases that can be recognized when spoken by the caller (the full set of phrases that can match a grammar is said to be generated by that grammar). A separate grammar set is often associated with each input state in a dialog, although some grammars may be used in more than one context.

The application hands the grammar off to a speech recognition engine (a component of the voice browser) for processing. In effect, the application uses the grammar to tell the recognizer what words or patterns of words should be expected at a given point in a dialog. The recognizer chooses the best match (if possible) from entries in the grammar, returning the entry and a confidence score that describes the "closeness" of the match. The confidence score is determined algorithmically by the recognizer, as part of the process of matching the te plate models from the grammar to the caller's utterance.

A well-designed grammar is flexible enough to match most of the responses you might expect from a caller, but restrictive enough to give the recognizer a reasonably small and distinct set of options from which to select a match. A good grammar optimizes recognition accuracy and enables the recognizer to return matches with higher confidence scores. The challenge of grammar development is choosing the right balance between flexibility and restriction.

Listing 1 shows an example of a GRXML grammar. This example is in the XML form developed by the W3C as part of its SRGS recommendation (this form is commonly called GRXML). There are also a number of other speech grammar formats that may be supported by speech recognition engines, including ABNF (a non-XML format defined in SRGS) and GSL (a proprietary Nuance format).

Listing 1. An example GRXML grammar.

```
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns="http://www.w3.org/2001/06/grammar" xml:lang="en-us"
version="1.0" root="root" mode="voice" tag-format="swi-semantics/1.0">
<rule id="root" scope="public">
<one-of>
<item>
<one-of>
<item>website help</item>
<item>web help</item>
<item>web</item>
</one-of>
<tag>CHOICE='website_help'</tag>
</item>
<item>
<one-of>
<item>cancel</item>
<item>cancel service</item>
</one-of>
<tag>CHOICE='cancel'</tag>
</item>
</one-of>
<item repeat="0-1">
<ruleref uri="#postphrase"/>
</item>
</rule>
<rule id="postphrase">
<one-of>
<item>please</item>
</one-of>
</rule>
</grammar>
```

Listing 1 matches phrases that the caller might use to request website help or to cancel his or her service. Although it's simple, it shows two interesting features typical of speech grammars. First, several synonyms are mapped to a single return value (using the one-of element) for both options. To cancel service, the caller could say either "cancel" or "cancel service", yet "cancel" will be returned as the (semantic) response (or "slot") value in both cases.

Second, the postphrase rule defines an optional utterance (i.e."please") that may occur after the main entry. Because of the postphrase, "cancel please" would also match the cancel rule. The postphrase rule is an example of postfiller. Postfillers consist of utterances that come after keywords that may increase the range of possible responses but which are ignored by the speech recognizer in determining the match. Similarly, prefillers consist of utterances that may occur before the target (e.g., "I want to") but which don't affect the return value.

Some Best Practices

When approaching grammar development it is important to recognize that every ication is different, each with its own set of data rules and its own target audience. What's more, since callers are human, it's not possible to create hard and fast rules about the best way to build a grammar. Instead, we've compiled a short list of best practices to use in grammar design. This is not a comprehensive list, but applying these practices will go a long way toward creating efficient, responsive grammars.

Best Practice #1: Consider All Relevant Prompts

Carefully review the prompt or set of prompts that are associated with a grammar to come up with terms that a caller is likely to use. If a grammar is associated with more than one prompt—for example if the initial prompt and reprompts differ however slightly in what they instruct callers to say—review each of those prompts independently to come up with a set of appropriate target phrases and synonyms.

Callers tend to parrot back prompts, so you may hear more of the prompt in the response than just the expected keyword. For example, a caller might respond to a prompt that says, "Say 'main menu' to start over" with phrases such as "main menu to start over", "start over" or just "main menu." This requires that you expand the synonym list to include additional phrases.

Say you have the following prompts, both associated with the same grammar:

"To look up an order, just say "order"..."
"Remember: Just say 'order' to look up another order..."

Although "order" is the keyword you're looking for, the caller might also say "look up an order" in response to the first prompt and "look up another order" in response to the second. Both must be handled by the grammar.

Nevertheless, when building the initial grammar, start with expected answers (phrases a cooperative caller would say) and err on the side of excluding possible additional synonyms rather than including them. The more synonyms you have, the greater the chance of incorrect matches. The number and acoustic similarity between in-grammar phrases can also deflate confidence scores, thus potentially resulting in unnecessary reprompting. Additional synonyms can be added during application tuning, when you have evidence that a reasonable number of callers are speaking those phrases.

Best Practice #2: Avoid Overgeneration

Combining prefill, postfill, and a keyword with several synonyms can quickly lead to an explosion of combinations of valid responses. The number of combinations is given by:

(# prefillers +1) x (# entries) x (# postfillers +1)

With just two prefillers (e.g., "help with" and "help me with") and one postfillers ("please"), a grammar with four entries turns into 24 valid phrases. This can negatively impact recognition because, as mentioned above, the more in-grammar items there are, the greater the likelihood of a false or low-confidence match.

But overgeneration can also lead the application to accept nonsensical responses. Consider a grammar used to locate a hotel. The grammar includes the names of all fifty states, plus the additional keyword "international". It also includes a prefiller value of "the hotel is in." A user might reasonably say "the hotel is in Texas." But this grammar would also match on "the hotel is in international." Accepting nonsense responses makes the grammar needlessly complex, thereby negatively impacting recognition performance and potentially detracting from the caller experience. Essentially, if a native speaker wouldn't naturally produce it, don't include it in your grammar.

Overgeneration of this kind can be reduced by not automatically applying sub-rules with simple combination. In the example above, each state name might be given a synonym that includes "the hotel is in" phrase, and the prefiller could be removed. Alternatively, this prefiller rule could be applied only in the domains that it makes sense. There are also more sophisticated techniques, for example the use of JavaScript or a dynamic grammar, which can also help to combat overgeneration.

Best Practice #3: Re-Use, but Re-Use Carefully

As we said earlier, a grammar is a set of expected phrases for each individual input state. And, because a speech application often has dozens of different prompts, it's tempting to re-use grammars for what appear to be almost identical interactions. But even simple grammars may not be as generally applicable as they seem. Again, it's important to note the differences between candidate prompts where grammar-sharing is being considered. Take, for example, a simple "yes/no" grammar, which may include yes, no, and a set of general synonyms (eg, "OK"). Now let's look at two prompts that may seem on the surface to elicit similar responses:

> "That was Austin. Did I get that right?"
> "Would you like to place an order?"

In response to the first prompt, a caller may say "right," "that's right," "yes you did" or "correct" as well as the aforementioned generic entries; in response to the second, he or she may say "yes I would or "yes please". It may be tempting to salvage the reusability of this grammar by expanding it to include all of these potential entries. But doing so would introduce overgeneration given the domain and its associated recognition pitfalls.

Better candidates for re-use are grammar rules that can be used at prompts throughout the application, such as "help me out" or "operator." Other candidates for reusable grammars are common data formats, such as dates (but be sure that they really are the same kind of "date"— see Best Practice #4).

Best Practice #4: Know Your Data

You can exploit the differences among different types of data and among different data formats to build highly selective grammars. By restricting potential matches to those both allowed by data format rules and valid for a specific data type, you can greatly reduce the srecognition domain, thus boosting recognition accuracy and confidence.

Account numbers are often restricted to a specific format, such as a certain number of digits, or a requirement that the account starts or ends with a letter. Adding these restrictions to the account number grammar is an example of how you can take advantage of format restrictions to build better grammars. Taking advantage of known data formats is crucial when employing alphanumeric grammars and is recommended even for varying length digit strings when it's not known which length string the caller will use.

Be specific when identifying data types for each element. A reservation date (which must be in the future) and a birth date (which must be in the past), though both dates, belong to completely disjoint sets. You can reduce grammars for these data items by including past and future restrictions.

If you have enough domain knowledge, you can and should introduce additional type restrictions For example, suppose you are processing inquiries for a retirement community with a minimum age requirement. You can use the minimum age to further narrow the range of probable birth dates.

Including these restrictions up front in the grammar, creates additional benefits in addition to increasing recognition accuracy. It offloads some of the validation logic from that application to the recognizer, reducing the load on the application. It also lowers the risk of confirming an utterance in the dialog ("You said March 4th, 1921. Is that right?") only to have it later rejected by the application ("I'm sorry, that's not a valid date for travel"). Consistency errors such as these can quickly frustrate callers and result in a loss in caller confidence.

Starting Early

To take advantage of structures and patterns within the data that can enhance recognizer accuracy, it's critical that you start the grammar design process early, during the first stages of application design. By starting early, you can ensure that the data you need to restrict grammars appropriately can be made available by the host system, and that the recognition task required for a given input state is technically feasible. For example, you could load a grammar used for movie ticket purchase with a list of features that are currently playing, but you might need to build the web service to deliver that list as part of the application development effort.

Starting early also allows application developers to work closely with speech scientists to coordinate prompts and grammars with other parts of the application. As with any UI elements, speech application prompts and grammars can't be bolted on at the end of the process—they need to be designed and integrated into the flow of the application.

Avaya supplies both tools and professional services expertise in support of grammar development. Avaya Dialog Designer includes a built-in grammar editor for VXML applications that allows the user to create grammars as they work in a graphical dialog editor. The grammar editor in Avaya Dialog Designer creates list-based SRGS-compliant grammars for any of the speech recognition engines supported by Avaya Voice Portal. Avaya Professional Services also offers a wide range of consulting services, including grammar design and development support and application tuning.

Developing flexible and efficient grammars pays off in speech applications that respond to typical speaking patterns accurately the first time. But getting all the way there requires some additional work beyond initial deployment. In part 2 of this series, we'll talk about tuning, and how to analyze data from actual calls to further improve grammars. ■

Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.

Developing Speech Grammars That Rock, Part 2: Grammar Tuning

Grammar Tuning is the process of improving grammars based on analysis of actual caller interactions. Here we'll tell you what to look for in a grammar tuning analysis—and what to do with what you find.

by Steve Apiki

Grammar development doesn't end when an application goes live. It's only with a live application that you can collect the kind of information you need to create truly responsive grammars. In grammar development, tuning closes the loop between deployment and development, feeding actual call data back into the design process to drive progressively better recognition accuracy and a better caller experience. In part one of this series, we highlighted some best practices for speech grammars that you can apply during design and development. Here, we'll talk about what happens after deployment, when you can collect data from caller responses to tune the grammars you've built.

A speech grammar defines all the words and phrases that an application expects in response to a prompt, expressed as a set of rules. The aplication provides the grammar to the speech recognition (or ASR) engine, which returns the best matching phrase in the grammar along with a confidence score that measures the quality of the match. It's our goal to build applications that respond to the caller's words quickly and accurately the first time. To do that, we want grammars that include all of the phrases a cooperative caller is likely to use, and nothing else. The tuning process can help move us closer to these ideal grammars.

The Tuning Cycle

Grammar tuning is a specific effort directed at improving caller experience by optimizing recognition accuracy. It's an important post-deployment activity for speech applications that includes caller experience tuning, but distinct from QA testing, customer acceptance testing and load testing. Grammar tuning is a cyclical process that relies on continuous or at least repeated data collection to get to the best results.

The grammar tuning workflow has four stages:

Capture

Capturing tuning data begins by recording caller interactions in both audio and written form and then manually transcribing these recordings. For recording the utterances, you can use call recording capabilities on the Voice Portal. Manual transcription is a labor-intensive process and may limit the amount of data that you can collect. However, it's the human in the transcription loop that allows us to determine recognition accuracy in later stages.

Classification

Using the application logs and recognizer data, we next classify caller utterances based on whether they are in-grammar or out-of-grammar, and whether or not they were correctly handled by the recognizer (see the sidebar, "The Terminology of Tuning" for an explanation of common classifications). This classification data is then compiled into statistics that can be used for analysis.

Analysis
Here, we analyze the data to spot problems like frequently misrecognized utterances, or common caller responses that are out-of-coverage, and develop suggested changes to grammars to address these problems.

Implementation
Lastly, we close the loop by implementing the recommended grammar changes resulting from this analysis.

After implementation, the process starts again as you capture new data to measure the effect of your changes and suggest steps for further recognition optimizations.

Analyzing Tuning Data

Once you've got your data set classified, you can look at the histogram of classifications and start to see where you may need to make adjustments. But these adjustments aren't simple control knobs—none of the parameters in grammar tuning are completely orthogonal. Making changes to improve measures in one category will also affect other categories, sometimes in a negative way. See the sidebar to understand how these parameters are defined.

From a high level, there are two ways to approach tuning: you can optimize for in-grammar (that is, increasing CA-in and decreasing FA-in and FR-in), or you can optimize for out-of-grammar (increasing CR-out and decreasing FA-out). Steps that you may take to better in-grammar performance will generally degrade out-of-grammar performance, and vice-versa. Getting the balance right can be tricky. As a rule it's best to focus on in-grammar first to make sure that the recognizer is recognizing legitimate caller requests as intended. After that, you can move on to better screening of out-of-grammar items, understanding that any changes made to account for these could negatively impact recognition for cooperative callers.

When reviewing in-grammar behavior, a large number of utterances that are classified as FA-in may indicate that items in the grammar are too acoustically similar (e.g., "repeat"

The Terminology of Tuning

Part of the tuning process is to classify recorded caller utterances based on their relationships to the grammar and to the recognizer's response. These are some common terms used in classification.

IG In Grammar
Utterances that are included within the allowable words and phrases generated by the rules of the grammar.

OOG Out of Grammar
Utterances that are not covered by the grammar, including "um," "ah," and background events such as coughing and side speech. IG and OOG are mutually exclusive, and all utterances can be classified as either IG or OOG.

OOC Out of Coverage
Utterances that are OOG, but including direct user speech only, and excluding noise and background events. This is a subcategory of OOG. OOC utterances are especially interesting for tuning because they are intended responses by the user that are not included in the grammar.

IG, OOG, and OOC classifications are independent of the recognizer's response. The following classifications are all subcategories of the first three that also consider how the utterance was recognized.

CA-in Correctly Accepted In Grammar
An IG utterance that the recognition engine correctly recognized.

and "delete"). If any of the words in this list are unnecessary (say, due to overgeneration in the grammar), then they can simply be eliminated. Otherwise, the grammar entries should be made more acoustically distinct. Some ways to choose entries that are easily differentiated include choosing phrases of different length, or phrases that differ in vowel sounds, number of syllables, or other acoustic parameters. You may also get some improvement by changing settings on the recognizer or employing creative methodologies on the design side.

A high incidence of FR-ins may mean that the recognizer's confirmation levels are set too high. But it may also mean that there's a variance between the pronunciation that the recognizer expects and the pronunciations actually used by callers. One way to get around the pronunciation problem is to reference a custom pronunciation lexicon from the grammar. It's also possible that two acoustically similar and thus competing grammar items are resulting in artificially lowering the confidence scores returned.

Out-of-Grammar Tuning Data

When considering the out-of-grammar tuning data, the first classification to consider is OOC (out-of-coverage). These are direct caller utterances without any background events (such as noise or side speech) that simply aren't in the grammar. If there are a significant number of different instances where callers tried the same OOC utterance, that utterance should be considered a candidate for inclusion in the grammar. In part one of this series, we suggested a best practice of limiting the initial grammar as much as possible, and adding additional grammar as needed during tuning. It's from this category that you would identify these additional grammar items.

Second, take a look at all OOG (out-of-grammar) items. Try to determine if there is a consistent noise source among these items (some consistent noise sources may be line noise, or echo from the prompt) that may be wreaking havoc with recognition accuracy. If there is a repeatable noise source, it may make sense to modify confirmation thresholds so that items in this group are confirmed (e.g., "Did you say 'Balcony?'") rather than rejected outright or modify the endpointer sensitivity. If the noise source actually is prompt echo, that may be something you can control by lowering the outgoing prompt volume.

FA-in Falsely Accepted In Grammar
An IG utterance that was recognized as a different item in the grammar.

FR-in Falsely Rejected In Grammar
An IG utterance that was rejected and therefore not recognized.

CR-out Correctly Rejected Out of Grammar
An OOG utterance that was correctly rejected.

CR-OOC Correctly Rejected Out of Coverage
An OOC utterance that was correctly rejected.

FA-out Falsely Accepted Out of Grammar
An OOG utterance that was falsely accepted as an in-grammar item.

FA-OOC Falsely Accepted Out of Coverage
An OOC utterance that was falsely accepted as an in-grammar item.

When reviewing recognizer behavior when presented with out-of-grammar utterances, a high number of FA-outs may indicate that confirmation levels are set too low. In this case, you should increase the high confirmation threshold to increase the number of confirmations versus accepts. FA-outs may also occur when a caller uses a variation on an in-grammar phrase. For example, the in-grammar phrase may be "main menu," but the caller may say "main menu please." This is technically out-of-grammar, but it may be accepted by the system. In these cases, the system is probably already doing what the caller expects, but it may be appropriate to make the grammar adjustments (such as adding postfiller) to change these items to CA-in.

In every case, optimizing a grammar requires really looking into the details of each specific recognition state. Determine the condition you need to fix, then look at all the possible contributors, including the grammar, confirmation thresholds, ASR parameters, the prompt, and the quality of the line. As an example, say that you are dealing with a number of FA-ins. These could be due to the common suggestions given above, or they could be something more exotic, like ASR end pointer sensitivity, or prompt echo resulting in premature barge-in. Be sure to test proposed solutions to make sure they solve the underlying problem.

Finding the Balance

Developing a flexible, responsive grammar is an iterative process that depends heavily on collecting production data and feeding that data back into grammar development with tuning. Although grammar tuning isn't easy, it pays off in better caller experience. The grammar tuning process is a balancing act, all about finding the balance between IG and OOG items. And balancing is never a one-time effort, it requires constant course correction as new functionality is added, caller demographics change, and caller's become expert users. That's what grammar tuning provides.

Proper tuning requires both tools for data collection and analysis, and expertise to interpret results. As it does for stages throughout grammar development, Avaya provides both tools and professional service expertise in grammar (and complete application) tuning through Avaya Professional Services.

As we said in part one, the most important advantage you can get in grammar development is an early start. By starting early, you can involve developers and speech scientists in the process to ensure that prompts and grammars are structured properly and that your application has access to all the data it needs for grammar processing. ■

Steve Apiki is senior developer at Appropriate Solutions, Inc., a Peterborough, NH consulting firm that builds server-based software solutions for a wide variety of platforms using an equally wide variety of tools. Steve has been writing about software and technology for over 15 years.