METREOS

IP Communications Applications for Business Advantage

DEVELOPER GUIDE

Metreos Communications Environment 2.2

www.metreos.com

Part Number M-2400-DG-01
10/2005

Metreos Corporation
4401 Westgate Blvd.
Suite 308
Austin, Texas 78745

METREOS

Contents

About This Guide vii
Target Audience vii
Organization vii
Notational Conventions vii
Notes, Cautions, and Warnings vii
Typographical Conventions viii
License/Certification Information viii
Chapter 1 Introduction 1
Metreos Communication Environment 1
The MCE Application Server 1
The Metreos Media Engine 2
Metreos Visual Designer 3
Metreos Management Console 3
Example Applications 4
Chapter 2 MCE System Architecture 5
The Metreos Application Runtime Environment Architecture 5
Provider Framework 6
Router 6
Telephony Manager 7
Application Manager 7
Management Interface 7
Application Runtime Environment 8
The Application Lifecycle 8
Development Phase 9
Build Phase 9
Deployment Phase 10
Execution Phase 10
Application Components 10
Scripts 10
Installers 10
Databases 11
Media Resources 11
The Event-Action Model 11
Events 11
Event Signatures 11
Types of Events 12
Triggering Events 12
Unsolicited Events 12
Asynchronous Events 12
Hybrid Events 13

Actions 13
Provider Actions. 13
Synchronous Provider Actions. 14
Asynchronous Provider Actions. 14
Native Actions 15
Core Actions 15
Scope. 15
Application-Level Scope 15
Script-Level Scope. 16
Function-Level Scope 16
Application Resources 16
Installers and Configuration Parameters 16
Database Management 19
Chapter 3 Inside MCE Applications 21
Application Architecture 21
XML-Based Implementation. 22
Execution Model. 23
Application Script Elements 24
Application Script Triggers. 24
Functions 25
Variables 26
Actions 28
Chapter 4 Metreos Visual Designer 31
Metreos Visual Designer Tour 32
Explorer Window (1). 32
Application Canvas (2) 33
Overview Window (3) 33
Toolbox (4) 33
Property Grid (5). 33
Output Window (6). 36
Projects and Files. 36
Using the Metreos Visual Designer 36
Creating Projects 37
Creating a Script 37
Additional Scripts and Script Navigation 38
Event Handlers and Triggering Events 38
Manipulating Elements on the Application Canvas 38
Manipulating Execution Paths on the Application Canvas. . 40
Elements That Create Handlers 40
Special Elements: Variables and Other Application Components
40
Loops 40
Comments 40
Variables 41
Installer Manager 42
Chapter 5 Sample Applications 47

Weather Application 47
Nodes and Functions 52
Using Loops 55
IEnumerator: 57
IDictionaryEnumerator: 57
Script Execution 58
Announcement Application. 61
Getting Started. 61
Assigning Flow Control Labels 67
Variables 67
Other Action Properties 68
Additional Media Resources 69
Asynchronous Events 70
OnPlay_Complete 71
OnPlay_Failed 73
OnGatherDigits_Complete and OnGatherDigits_Failed . . . 74
OnRemoteHangup 76
Adding Media 77
Script Execution 79
Chapter 6 Native Actions and Native Types 81
Developing Native Actions 81
LogWriter Property 83
Action Attribute 83
ReturnValue 85
Adding a Native Action to a Metreos Visual Designer Project 86
Developing Native Types 87

Appendix A
API Reference. 89
Call Control. 89
Actions 89
Events 103
Media Control 110
Actions 110
Events 133
HTTP 143
Actions 144
Events 145
Types 147
Application Control 148
Actions 148
Conditionals 157
Database 160
Actions 160
Cisco IP Phone. 163

Actions . . . 164
Types . . . 183
Timer Facility . . . 194
Actions . . . 194
Events . . . 197
Cisco DeviceListX . . . 198
Actions . . . 198
Asynchronous Callback Events . . . 202
LDAP . . . 204
Cisco Extension Mobility . . . 208
Cisco AXL SOAP . . . 214
Types . . . 246
Dial Plan . . . 259
Actions . . . 259
Mail . . . 261
Actions . . . 261
Call Control (Deprecated): Actions . . . 263
Asynchronous Callback Events . . . 273
Media Control (Deprecated) . . . 278
Actions . . . 279
Standard Types . . . 298

Appendix B
Attributes . . . 299
Appendix B: Attributes . . . 299
Metreos.PackageGeneratorCore.ActionAttribute . . . 299
Metreos.PackageGeneratorCore.ActionParamFieldAttribute . . . 300
Metreos.PackageGeneratorCore.PackageDeclAttribute . . . 301
Metreos.PackageGeneratorCore.ResultDataFieldAttribute . . . 302
Metreos.PackageGeneratorCore.ResultValueAttribute . . . 303
Metreos.PackageGeneratorCore.TypeInputAttribute . . . 304

Index . . . 305

METREOS

About This Guide

This Metreos Communications Environment 2.2 Developer Guide explains how to design, develop, and deploy applications for the Metreos Communications Environment (MCE). This guide also describes the architecture of the environment and MCE applications.

For information about basic telephony and IP telephony, Metreos recommends the following Internet Web sites:

http://en.wikipedia.org/wiki/Voip

http://www.voip-info.org/tiki-index.php

http://www.packetizer.com/voip/

Target Audience

This Metreos Communications Environment 2.2 Developer Guide is intended for developers who are planning to build IP telephony applications for the MCE using the Metreos Visual Designer. This guide does not provide general information about IP telephony or application development for an IP telephony environment.

Organization

This guide is organized into the following sections:

- About This Guide
- Chapter 1: Introduction — Overview of the MCE
- Chapter 2: MCE System Architecture — Components, MCE Application Architecture, and the Action-Event Model
- Chapter 3: Inside MCE Applications — MCE Application Architecture
- Chapter 4: The Metreos Visual Designer — Visual Designer
- Chapter 5: Sample Applications — How to Develop an Application
- Chapter 6: Native Actions and Native Types — How to Develop Native Actions and Use Native Types
- Appendix A: API Reference — Metreos APIs
- Appendix B: Attributes — Metreos Attributes
- Index

Notational Conventions

The following section summarizes the notational conventions used in this Metreos guide.

Notes, Cautions, and Warnings

NOTE: *A Note provides important information, helpful suggestions, or reference material.*

CAUTION: *A Caution indicates a potential risk for damage to hardware or loss of data, and describes how to avoid the problem.*

WARNING: *A Warning indicates a potential hazardous risk that could result in serious bodily harm or death.*

Typographical Conventions

This section defines the general typographical conventions followed in this Metreos guide.

- **Bold** typeface — Represents literal information:
  - 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
- *Italics* typeface — Represents:
  - New concepts
  - A variable element such as *filename*.**mca**
- `Courier` typeface — Represents code or code fragements or text that you enter. For example, type `xxxxx`.
- ...(elllipsis) — Represents omitted content in code fragments.
- <UPPERCASE> — Typeface enclosed in angle brackets represents keys and keystroke combinations that type. For example, <CTRL + ALT + DEL>.

License/Certification Information

Metreos Corporation

End User License

Important - Use of this Software is subject to license restrictions. Carefully read this license agreement before using the software.

This End User License Agreement (the "Agreement") is a legal agreement between you, either individually or as an authorized representative of the company or organization acquiring the license, and Metreos Corporation ("Metreos"). USE OF SOFTWARE INDICATES YOUR COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. If you do not agree to these terms and conditions, promptly return or, if electronically received, certify destruction of the Software and all accompanying items within five days after receipt of Software and you will receive a full refund of the applicable license fees paid.

1. License Grant

a. The software programs you are installing, downloading, or have acquired with this Agreement, including any related equipment or hardware, documentation, updates, upgrades, modifications, revisions, copies and design data ("Software") are copyrighted, trade secret and confidential information of Metreos and its licensors who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Metreos grants to you, subject to payment of appropriate license fees, a non-exclusive, non-transferable, internal-use only, term license to use the Software owned or distributed by Metreos in machine readable, object-code form on the computer hardware or at the site(s) for which an applicable license fee has been paid, as authorized by Metreos.

b. To the extent that you design applications that operate on the Metreos Communications Environment (the "MCE"), Metreos grants you a non-exclusive, non-transferable, internal-use only, term license to use the Software, including the MCE, for the purposes of operating such programs, subject to payment of appropriate license fees.

2. Restrictions and Intellectual Property Ownership

a. You may not (i) remove or modify any notice of Metreos' proprietary rights, (ii) re-license, rent, lease, timeshare, or act as a service bureau or provide subscription services for the Software, (iii) use the Software to provide third-party training, except for training agents and contractors authorized under this Agreement; (iv) assign this Agreement or give the Software or an interest in the Software to another individual or entity; (v) cause or permit reverse engineering or decompilation of the Software; (vi) disclose results of any Software benchwork tests without Metreos' prior written consent; or (vii) modify the Software or any portions thereof without Metreos' prior written consent.

b. The Software, which is copyrighted, and any modifications, upgrades, or updates thereto, is the sole and exclusive property of Metreos and is a valuable asset and trade secret of Metreos. Metreos retains all ownership and intellectual property rights to the Software and to any modifications, upgrades, or updates thereto. Except for the rights granted in herein above, you shall have no right, title, or interest of any kind in or to the Software.

c. Metreos may audit your use of the Software. If Metreos gives you or your organization reasonable advance written notice, you agree to cooperate with the audit, and to provide reasonable assistance and access to information. You agree to immediately remit any underpaid license and technical support fees determined as the result of such audit.

3. Term and Termination

a. This Agreement remains effective until expiration or termination. This Agreement will immediately terminate upon notice if you exceed the scope of the license granted or otherwise fail to comply with the provisions in sections 1 and 2 above. For any other material breach of the Agreement, Metreos may terminate this Agreement if you are in breach and fail to cure the breach within thirty (30) days of written notification. If Software is provided for a limited term use, this Agreement will automatically expire at the end of the authorized term.

b. Upon termination of this Agreement for any reason, you shall within ten (10) business days return to Metreos all Software. Additionally, you agree to delete from any permanent machine storage (i.e., hard disk) previously loaded copies of the Software in all forms. Upon request of Metreos, you shall certify in writing that all copies of the Software and associated documentation have been destroyed or returned to Metreos. The indemnity and limitation of liability obligations hereunder, as well as your obligations with respect to confidential treatment of the Software and Metreos' trade secrets, other intellectual property, and proprietary information, shall survive the termination of this Agreement.

4. Limited Warranty

a. Metreos warrants that the Software will substantially operate as described in the applicable Software documentation for ninety (90) days after Metreos delivers it to you. **THE WARRANTY HEREIN IS EXCLUSIVE AND IN LIEU OF ALL OTHER WARRANTIES, WHETHER EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. METREOS MAKES NO WARRANTY THAT ANY SOFTWARE WILL PERFORM ERROR-FREE OR UNINTERRUPTED, OR THAT ALL ERRORS THEREIN CAN OR WILL BE CORRECTED. METREOS FURTHER DISCLAIMS ANY IMPLIED WARRANTIES ARISING FROM COURSE OF PERFORMANCE, COURSE OF DEALING OR USAGE OF TRADE.**

b. For any breach of the above warranty, Metreos' entire liability and your exclusive remedy shall be, at Metreos' option, either (1) refund of the fees paid upon return of the Software to Metreos, or (2) correction or replacement of the Software that does not meet this limited warranty, provided you have complied with the terms of this Agreement.

5. Indemnity

a. Metreos will defend and indemnify you against a claim that any Software, infringes a patent or copyright, provided that: (i) you notify Metreos in writing within thirty (30) days of the claim; (ii) Metreos has sole control of the defense and all related settlement negotiations; and (iii) you provide Metreos with the assistance, information, and authority reasonably necessary to perform the above; reasonable out-of-pocket expenses incurred by you in providing such assistance will be reimbursed by Metreos.

b. Metreos shall have no liability for any claim of infringement resulting from: (i) your use of a superseded or altered release of the Software if infringement would have been avoided by the use of a subsequent unaltered release of the Software which Metreos provides to you; or (ii) any information, design, specification, instruction, software, data, or material not furnished by Metreos or (iii) any combination of the Software with other hardware, software or processes that, but for the combination, the Software would not be infringing.

c. In the event that some or all of the Software is held or is believed by Metreos to infringe, Metreos shall have the option, at its expense: (i) to modify the Software to be non-infringing; or (ii) to obtain for you a license to continue using the Software. If it is not commercially feasible to perform either of the above options, then Metreos may require from you return of the infringing Software and all rights thereto. Upon return of the infringing Software to Metreos, you may terminate the Agreement with ten (10) days' written notice and you shall be entitled to a pro-rata refund of the fees paid for the infringing Software. This subsection sets forth Metreos' entire liability and exclusive remedy for infringement.

d. You will defend and indemnify Metreos and its licensors against any claim incurred by, borne by or asserted against Metreos or its licensors that relates to or results from (i) your use of the Software, (ii) any intentional or willful conduct or negligence by you or (iii) any breach of an applicable representation, covenant or warranty contained herein.

e. Should the party seeking indemnification ("Indemnitee") reasonably determine that the party from whom indemnity is sought ("Indemnitor") has failed to assume the defense of any claim referenced herein, Indemnitee shall have the right to assume such defense and have all expense and cost of the defense reimbursed by Indemnitor, including reasonable attorneys fees.

6. Confidentiality

The Software contains proprietary and confidential information of Metreos as well as trade secrets owned by Metreos. You agree to hold the Software in strict confidence and not to disclose the Software in any way except as expressly permitted hereunder. You agree to protect the Software at least to the same extent that you protect your similar confidential information, but in no event less than reasonable care. You further agree that you will not, directly or indirectly, copy the structure, sequence, or organization of the Software, nor will you copy any portion of the Software or related documentation to produce software programs that are substantially similar to the Software.

7. LIMITATION OF LIABILITY

EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, IN NO EVENT WILL METREOS BE LIABLE FOR ANY INDIRECT, INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES, OR DAMAGES FOR LOSS OF PROFITS, REVENUE, DATA, OR USE, INCURRED BY YOU OR ANY THIRD PARTY, WHETHER IN AN ACTION IN CONTRACT OR TORT, EVEN IF METREOS OR ANY OTHER PERSON HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. METREOS' LIABILITY FOR DAMAGES HEREUNDER SHALL IN NO EVENT EXCEED THE AMOUNT OF FEES PAID BY YOU FOR THE SOFTWARE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, METREOS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER.

8. Assignment; Jurisdiction

This Agreement will be binding upon, and will inure to the benefit of, the permitted successors and assigns of each party hereto. You may not assign, delegate, transfer, or otherwise convey this Agreement, or any of its rights hereunder, to any entity without the prior written consent of Metreos, and any attempted assignment or delegation without such consent shall be void. This Agreement, and all matters arising out of or relating to this Agreement, shall be governed by the laws of the State of Texas, United States of America. Any legal action or proceeding relating to this Agreement shall be instituted in any state or federal court in Travis or Dallas County, Texas, United States of America. Metreos and you agree to submit to the jurisdiction of, and agree that venue is proper in, the aforesaid courts in any such legal action or proceeding.

9. Severability; Waiver

In the event any provision of this Agreement is held to be invalid or unenforceable, the remaining provisions of this Agreement will remain in full force. The waiver by either party of any default or breach of this Agreement shall not constitute a waiver of any other or subsequent default or breach. Except for actions for nonpayment or breach of either party's intellectual property rights, no action, regardless of form, arising out of this Agreement may be brought by either party more than two years after the cause of action has accrued. The headings appearing in this Agreement are inserted for convenience only, and will not be used to define, limit or enlarge the scope of this Agreement or any of the obligations herein.

10. Restricted Rights Notice

The Software is commercial in nature and developed solely at private expense. The Software is delivered as "Commercial Computer Software" as defined in DFARS 252.227-7014 (June 1995) or as a commercial item as defined in FAR 2.101(a) and as provided with only such rights as are provided in this Agreement, which is Metreos' standard commercial license for the Software. Technical data is provided with limited rights only as provided in DFAR 252.227-7015 (Nov. 1995) or FAR 52.227-14 (June 1987), whichever is applicable.

11. Payment; Interpretation; Compliance

You will pay all amounts invoiced, in the currency specified on the invoice, within 30 days from the date of such invoice. This Agreement constitutes the complete agreement between the Parties and supersedes all previous and contemporaneous agreements, proposals, or representations, written or oral, concerning the subject matter of this Agreement. This Agreement may not be modified or amended except in a writing signed by a you and Metreos; no other act, document, usage, or custom shall be deemed to amend or modify this Agreement. It is expressly agreed that the terms and conditions of this Agreement supersede the terms of any purchase order. Each party agrees to comply with all relevant export laws and regulations of the United States and the country or territory in which the Services are provided ("Export Laws") to assure that neither any deliverable, if any, nor any direct product thereof is (a) exported, directly or indirectly, in violation of Export Laws or (b) intended to be used for any purposes prohibited by the Export Laws, including without limitation nuclear, chemical, or biological weapons proliferation. Each party agrees to comply with all federal, state and local laws and regulations applicable to this Agreement. Each party represents and warrants that it is qualified to do business in the geographies in which it will perform its obligations under this Agreement, and will obtain all necessary licenses and permits, and satisfy any other legal, regulatory and administrative requirements, necessary to its performance hereunder.

METREOS

Introduction 1

The Metreos Communication Environment (MCE) is the industry's first and only complete communications application environment. The platform allows rapid development and automated management of telephony applications to streamline business processes for a competitive advantage.

The MCE is a feature-rich platform that includes a management console (Metreos Management Console) and a development environment (Metreos Visual Designer). The Metreos Management Console allows management of Metreos telephony applications through a Web browser.

The Metreos Visual Designer abstracts coding details through a Graphical User Interface (GUI), allowing you to focus on application behavior rather than coding grammar and syntax. Using a conventional drag-and-drop technique, you can easily build complex telephony applications.

Metreos Communication Environment

Figure 1 shows a simplified view the MCE Architecture.

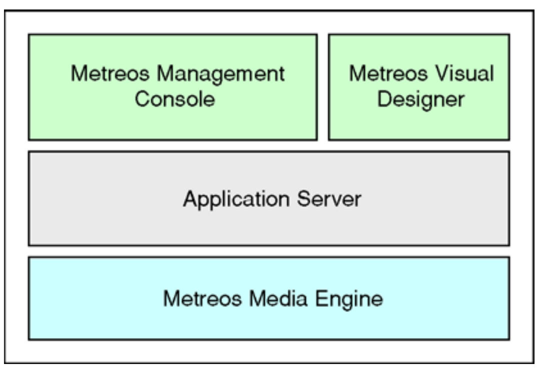

Figure 1: MCE Architecture

The Application Server is central to the MCE. It controls media and external resources under the direction of custom telephony applications. The Metreos Management Console and the Metreos Visual Designer are the user interfaces to the Application Runtime Environment.

The MCE also includes a software-based media engine that processes, mixes, analyzes and routes digital audio data.

The MCE Application Server

The MCE application server is a virtual machine that provides an application runtime environment and ships with a variety of providers shown in Figure 2.

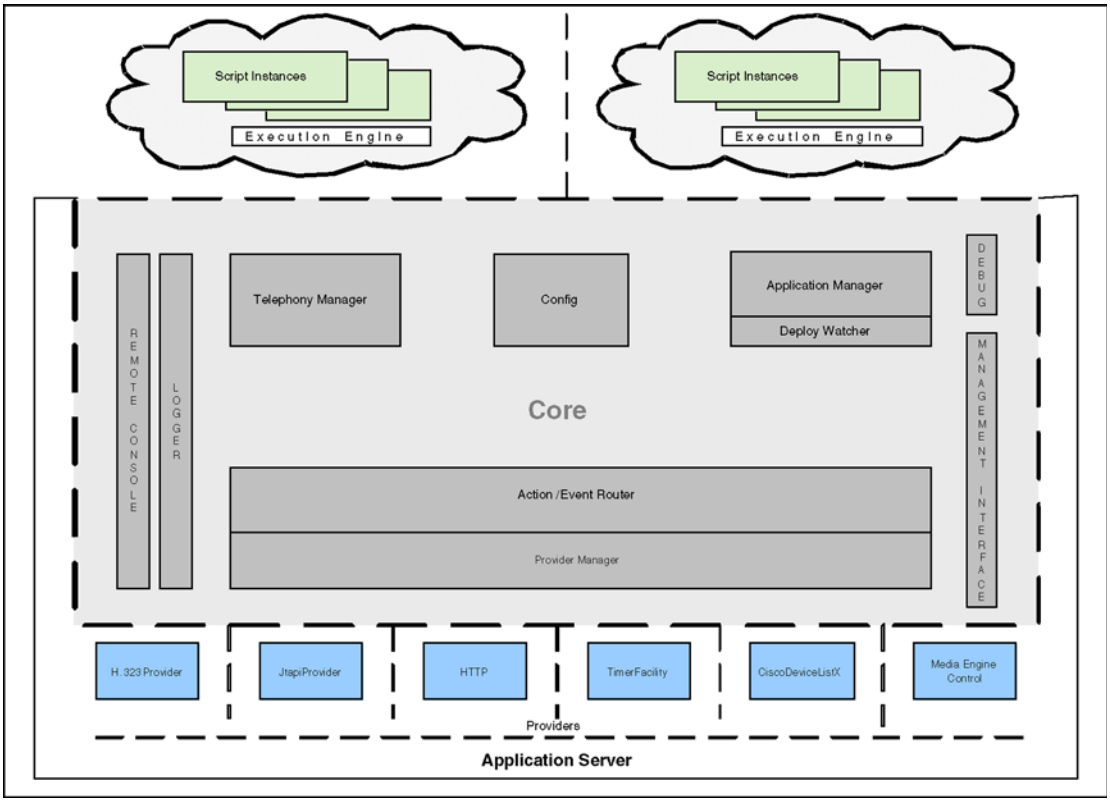

Figure 2: Application Runtime Environment

Applications communicate with other components of the MCE and third-party systems through providers. The application server ships with the following providers:

- H.323Provider — For first party call control
- JtapiProvider — For first party and third party call control
- HTTP — For network communication
- FacilityTimer — For creating event-driven delays
- CiscoDeviceListX — For caching CallManager device information
- Media Engine Control — For communication with the Metreos Media Engine

Metreos releases additional providers as needed to integrate the MCE with other enterprise systems. Refer to The Metreos "Application Runtime Environment" on page 8 for details about the Application Runtime Environment architecture.

The Metreos Media Engine

The task of managing media over an IP PBX has traditionally been reserved for expensive, high performance hardware switches. The Metreos Media Engine provides a software-only implementation to manage media that replaces the expensive hardware-driven approach to management.

The software-only design of the Media Engine provides the ability to keep pace with inevitable advancements in processor speed and capabilities and also helps to ensure interoperability with standard telephony and networking protocols, such as Real-Time Transport Protocol (RTP).

Each media server installation supports up to 240 simultaneous, bi-directional RTP connections that can provide dozens of simultaneous user sessions across multiple applications. Up to eight media servers can be connected to one application server.

The Metreos Media Engine includes a variety of powerful features including support for media streaming, Dual-Tone Multi-Frequency (DTMF) interpretation, multi-party conferencing, and recording.

Metreos Visual Designer

The Metreos Visual Designer simplifies the process of developing and deploying applications through the Metreos Visual Designer. The Visual Designer presents a GUI that allows the creation and connection of application components as shown in Figure 3.

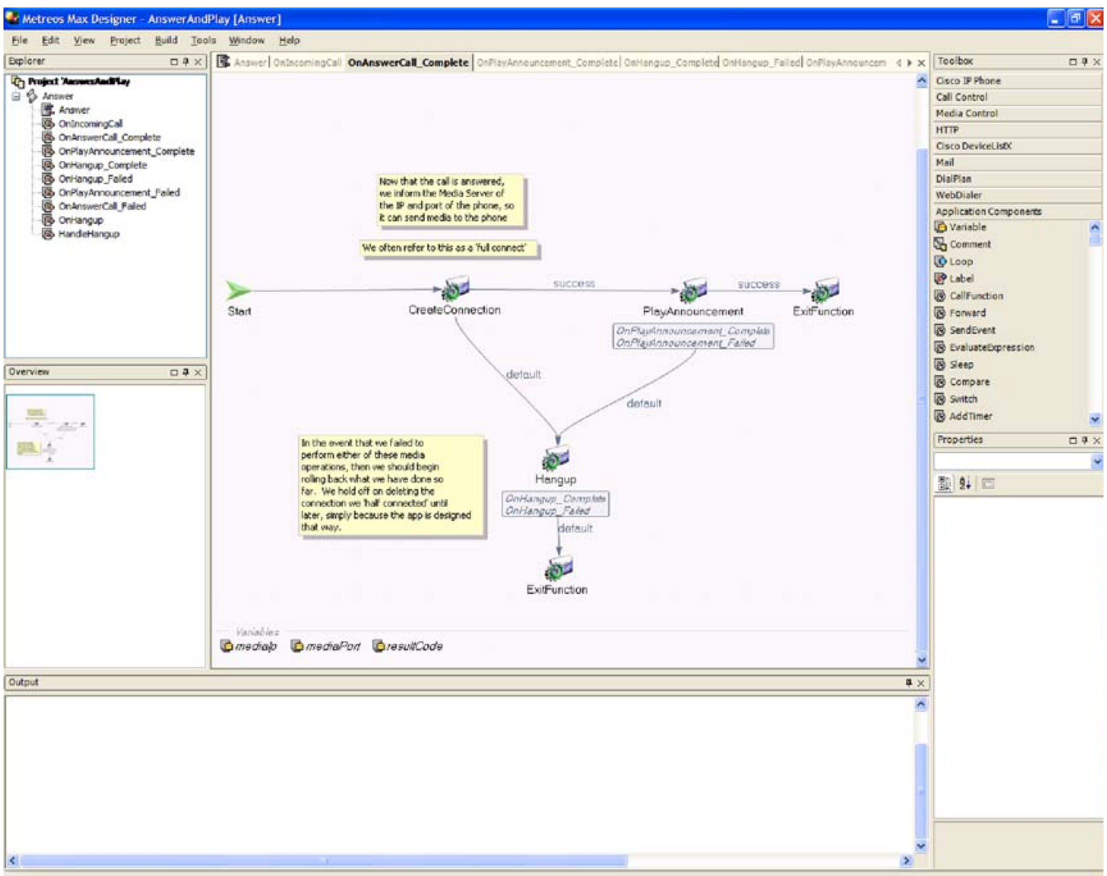

Figure 3: The Metreos Visual Designer

You can deploy finished applications to the MCE from the Visual Designer menu, or you can upload applications using the Metreos Management Console.

Metreos Management Console

MCE Administrators use the Metreos Management Console to manage the system through a Web-based interface that provides the following configuration and management capabilities:

- The application runtime environment

- All associated media servers
- User and license management
- Providers
- Applications
- Service control
- Backup/Restore
- System updates
- Viewing system logs
- System configuration

Figure 4 shows the Metreos Management Console Login Page.

Figure 4: The Metreos Management Console Login Page

Example Applications

The flexibility and power of the Metreos Visual Designer helps you build IP telephony applications quickly.Some examples are:

- Microsoft Exchange™ Integration — Provides alerts for meeting events by calling users on their mobile phones.
- Instant Messaging and Collaboration Integration — Starts collaborative conference calls from instant messaging sessions.
- Voicemail — Provides customized, flexible voicemail services to meet individual needs.
- Conferencing — Takes advantage of the rich features in the Metreos Media Engine to support instant recordable conferencing with participant mute and kick.
- Click-To-Talk — Extends a desktop PIM client, such as Microsoft Outlook™, to allow one-click calling between parties in your address book.
- Location-Based Forwarding — Integrates telephony and enterprise IT authentication systems to allow automatic forwarding of incoming calls based on your system login trail to your home phone, mobile phone and desk phone.
- Systems Management Alert Integration — Connects to Tivoli or HP OpenView and automatically starts conference calls.

Applications such as these can be built quickly and easily due to the the flexibility and power of the MCE.

MCE System Architecture 2

The flexible architecture of the MCE allows the production of powerful IP telephony applications with minimal effort. An understanding of this architecture is crucial for successful application development and deployment.

The Metreos Application Runtime Environment Architecture

The Metreos Application Runtime Environment provides the following functionality:

- Manages all component communication
- Processes user input
- Controls the Metreos Media Engine
- Provides interfaces for the Metreos Management Console and the Metreos Visual Designer

The architecture comprises several core components to provide these services as shown in Figure 5.

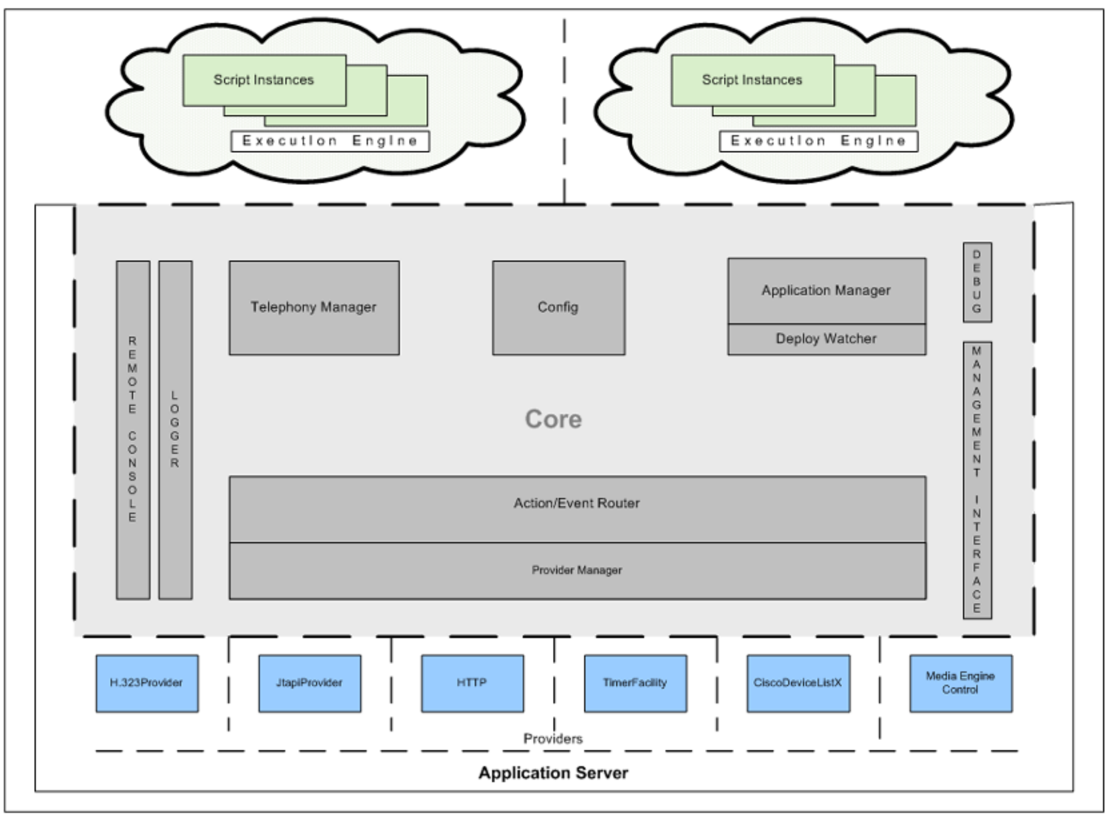

Figure 5: Metreos Application Runtime Environment Architecture

The MCE architecture separates the applications and providers from core components. Isolating the core prevents external components from adversely affecting the system in the event of system failure.

The components in the core are accessed through the management console. Each core component manages some aspect of the IP telephony system, and the management console enables the configuration of the core components.

The core components managed by the console include the following:

- Application Runtime Environment — Provides the runtime environment in which IP telephony applications are executed. It also serves as the bootstrap loader for the other core elements.
- Provider Manager — Manages loading and unloading of providers
- Router — Routes actions and events between providers and applications
- Telephony Manager — Manages the details of establishing and terminating a call
- Application Manager — Manages the installation and uninstallation of applications
- Management Interface — Notifies the core engine of changes to the Application Runtime Environment configuration
- Logger — Manages log messages

The following sections provide details about some of the core components.

Provider Framework

IP telephony applications within the MCE interact with the outside world through providers. Providers offer a family of services typically associated with a communication protocol. For example, the HTTP provider enables the acceptance of incoming Web traffic and responds accordingly.

Providers resemble a UNIX daemon process or a Windows service. They execute within their own virtual process space and facilitate communication between the Application Runtime Environment and external systems. Providers play a critical role in the operation and execution of the MCE. They are the sole means by which applications may execute actions and are the only sources for unsolicited events.

The Provider’s two primary functions are:

- To handle data received from external services and generate events to be handled by applications in the Application Runtime Environment.
- To respond to actions received from applications executing in the Application Runtime Environment.

The Provider Framework is a well-defined API, enabling third-party developers to build extensions to the MCE. By implementing the interfaces defined in the Provider Framework, developers can extend the MCE to any external system or protocol.

Router

In the Metreos system, applications specify a set of triggering criteria. After an instance of an application is triggered, the router retains the state of that instance and routes subsequent events until the instance terminates.

Telephony Manager

IP telephony applications use complex protocols to:

- Establish a connection between devices
- Exchange messages among devices
- Disconnect devices when appropriate

The Telephony Manager abstracts details of the telephony protocols so that you need be concerned with protocol-level details only if you want to be. The Telephony Manager provides a feature called *sandboxing*, which is a system-level, failsafe capability for ensuring that system resources do not remain in use after a script ends.

If a script prematurely stops, the script may not terminate outstanding calls and the media resources for outstanding calls could remain in use. In such an unlikely event, sandboxing permits the Telephony Manager to release the media resources.

Sandboxing should not be used for applications in which control of the call is transferred from one script to another. In this case, when the original script terminates, the Telephony Manager detects that the originating script is no longer active and the media has not been released. The Telephony Manager then releases the media on behalf of the original script and prematurely terminates the call.

Sandboxing is globally disabled by default, but can be enabled on the Telephony Manager Configuration page of the Metreos Management Console.

Application Manager

The Application Manager manages applications as they progress through the application lifecycle. The primary responsibility of the Application Manager is to unpackage applications and to create the application runtime for the application. The Application Manager also routes debugger commands to the appropriate application.

Each application instance has one or more partitions associated with it. A partition contains a set of configuration data to be applied to an application. All application-specific information is contained in the application partition. Application-specific information examples include triggering events, thresholds, and call control settings, as well as media settings and required IP addresses.

A partition is a template that determines the behavior of an application instance. Multiple partitions can be created for each application. Multiple users can execute concurrent instances of a given application, each running in a uniquely configured partition. The system creates a default partition during installation. The default partition can be used if multiple partitions are not required and can be edited through the Metreos Management Console.

Management Interface

The Management Interface manages communication with the Metreos Management Console and the Metreos Visual Designer. Configuration values and manual actions—managed through the Management Console—provide access to the Application Runtime Environment through the Management Interface.

Application Runtime Environment

The Application Runtime Environment is event-driven, in that MCE applications are comprised of scripts that execute in response to events. Refer to "The Event-Action Model" on page 11 for details about events and action scripts.

Within the constraints of system resources, any number of scripts can be executed concurrently and each script can have any number of instances. Multiple applications can run in parallel and multiple users can access any specific application simultaneously.

The virtual machine attempts to ensure system stability by segmeting applications from one another, such that an unstable application cannot adversely affect other applications. Should the virtual machine determine that an application instance is in an unstable state, the instance is terminated and unloaded.

The Application Lifecycle

Applications progress through a lifecycle from inception to retirement. Applications are usually initiated in an integrated development environment (IDE). As an application evolves, developers iteratively test the program by compiling and executing the application. The MCE facilitates the application lifecycle that consists of the following four phases:

1. Development — An application is developed using the Metreos Visual Designer development environment.

2. Build — The Metreos Visual Designer prepares the application for deployment by converting diagrams into a proprietary XML-based intermediate language. The code is then compiled and additional application data (such as media resources, an installer and any required databases) are combined into a single `filename.mca` archive file.

3. Deployment — The `filename.mca` archive file is uploaded to the Application Runtime Environment using the Management Console or the Visual Designer.

4. Execution — As events occur, the Application Runtime Environment executes scripts corresponding to those events.

Figure 6 depicts the application lifecycle:

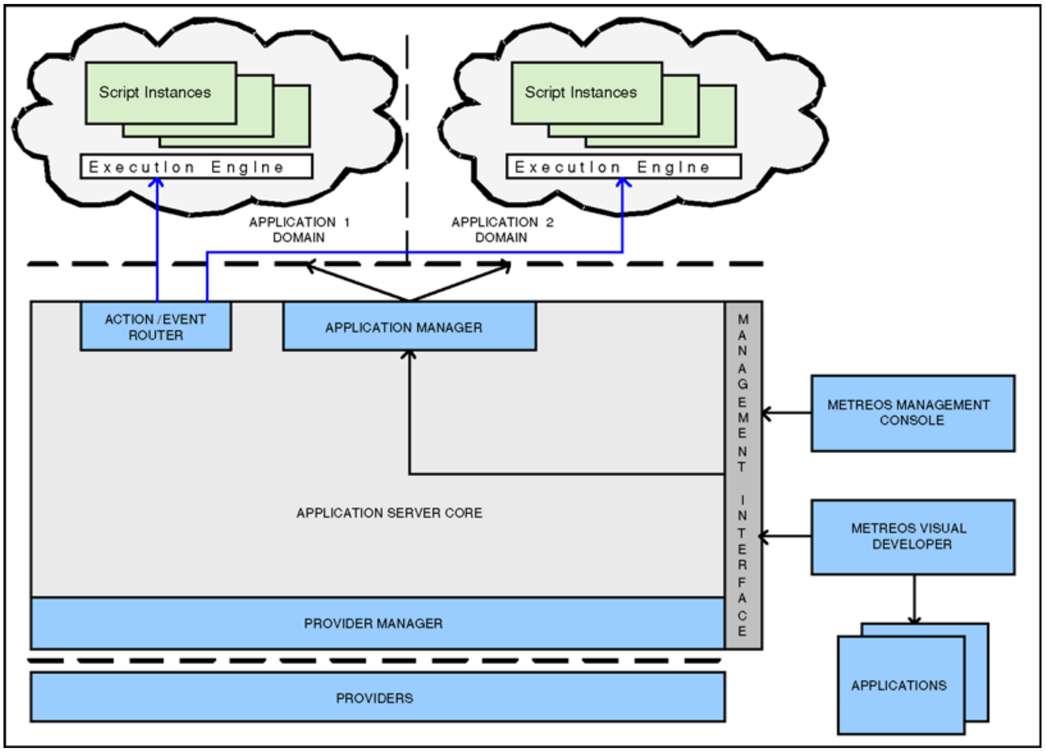

Figure 6: Application Lifecycle

Development Phase

Applications are developed using the Metreos Visual Designer. The Visual Designer automatically applies rules to ensure applications are well-formed. For example, dropping an asynchronous action, such as a Play, on the Visual Designer canvas causes functions to be generated for you which handle the action's asychronous events, such as Play Complete. This design feature assists in the development of applications conforming to the MCE architecture.

Build Phase

You can build or compile the application at any time during development. The build converts the logical representation of the application from symbols, such as boxes and arrows, to a proprietary XML-based intermediate language.

The build compiler offers typical compiler features such as detection of coding errors. Functions lacking proper exits, improperly initialized variables, and improperly specified execution paths are typical errors detected during a build. Using this iterative process, you can discover application-level errors prior to application deployment and testing.

The simplicity of the build process eases debugging. By attempting to compile the generated code, you can discover application-level errors before you commit the application to runtime for testing.

Once an application has compiled without error, the compiler packages all application components into a single unified .mca archive file. Packaged components include the application, its databases, media resources, installer, custom code, and versioning information.

Deployment Phase

Packaged archives are deployed to the Application Runtime Environment via either the Metreos Visual Designer or the Metreos Management Console. The Application Runtime Environment unpackages the application components and routes each to the appropriate destination.

The Application Runtime Environment determines whether databases for current installations will conflict with the impending installation. If this is the case, it will update schemas and data as defined by the new application.

Finally the Application Manager installs the application and its media resources. Any resulting application syntax, installer, or media server errors are reported via the Management Interface at this time.

Upon successful assembly, each application script is registered with the action/event router so that incoming events can trigger script execution.

Execution Phase

Whenever the Application Runtime Environment detects an event, an HTTP or call control event for example, it checks the triggering event signatures of each script. If it finds a match with the current event, the corresponding script is executed within its application domain. Each such instance of an application maintains its own local memory and access to resources.

Application Components

Each application consists of up to four component types, which are:

- One or more scripts
- An installer
- Databases
- Media resources

Scripts

All logic occurs within scripts or is directed by scripts. Scripts allow the programmer to specify:

- Actions to be executed
- Events to be handled
- Control structures to direct execution paths
- Variables to store data, such as databases or user input, and external resources to be accessed

Installers

Metreos applications must be explicitly installed before they can be used. If needed, you have the option of including special instructions to the MCE regarding the desired setup and deployment of the application services. Such services may include requirements such as dependencies on external providers and configuration settings.

An application has a single installer. The installer is represented as an XML configuration file automatically prepared by the Metreos Visual Designer. For more information on installers, refer to "Installers and Configuration Parameters" on page 16.

Databases

Telephony applications must often retain data beyond the execution of an individual script. MCE Applications can include any number of associated SQL databases enabling you to store relational data directly in the application. Information stored in databases persists until explicitly removed. This persistence makes MCE databases useful for storing long-term data such as call records, access permissions and general storage items.

If a database schema is defined by the application, the Application Runtime Environment executes the application's database script against its internal MySQL database server. State information to be shared with other scripts should be stored in such internal databases.

Applications that produce persistent data or that require access to an existing external data store, can connect to external databases using actions available in the Visual Designer. MySQL, Oracle, and Microsoft SQL Server databases are supported natively, but others can be defined easily.

Media Resources

Most useful telephony applications include audio prompts, such as pre-recorded greetings, error messages, and status messages. The MCE allows the packaging of any number of such announcements with the application. Supported formats are wav and vox audio.

NOTE: *Static prompts, when used in conjunction with NeoSpeech™ text to speech, must be 16 bit, 8 kHz. Refer to Additional Media Resources on page 83 for information on NeoSpeech.*

The Event-Action Model

The structure of the Application Runtime Environment is based on the Event-Action model, in which application instances respond to events associated with actions.

Events

The event architecture automatically executes code associated with each event. The MCE monitors a variety of externally occurring events, such as telephone calls and HTTP messages. When the MCE detects one of these external events, it generates a corresponding MCE event, such as CallControl.IncomingCall, or Providers.Http.GotRequest. MCE events originate in MCE providers to trigger MCE applications.

The MCE creates a new application script instance each time that script's triggering event is detected. Each such script instance operates independently, isolating the script's data and state from other such script instances. Subsequent events are routed to their corresponding script instance.

Event Signatures

MCE applications are collections of one or more scripts responding to events. Every programmatic task within the MCE must occur as a result of an event. Furthermore, an exact range of event parameters to which a task will respond can be easily defined. For example, a script could be executed every time a call is received from a specific phone number. Incoming call events could then be interrogated to determine the originating number of the incoming call and notify the script if there is a match. Such parameters constitute the event signature.

An event signature is a unique set of parameters used to determine which script the Application Runtime Environment should execute. Each application has an associated event triggering signature. Every time an event occurs, Application Runtime Environment functions, called event handlers, compare the event's signature to those associated with each script. When the event handler finds a match, it triggers a script instance associated with the event. This architectural characteristic serves two important purposes:

- Provides a programmatic way for event routing to occur within the architecture instead of within applications. This approach enables the MCE to handle the load balancing, failover, and performance requirements of a distributed deployment.
- Developers are freed from writing detailed analysis code to monitor all incoming events for the appropriate data and to discard unwanted events.

Types of Events

The MCE manages the following four types of events:

- Triggering — Events that trigger a script
- Unsolicited — Events that can occur during the execution of a script but there is no guarantee of their occurrence
- Asynchronous — Events that occur after a script has executed as a result of an asynchronous action in a script
- Hybrid — Events that are handled as either a triggering event or an unsolicited event based on event data

Although each event type occurs under different circumstances, the development requirements are similar for each type of event.

Triggering Events

Each script must specify a unique trigger that causes the script to execute. Each script has exactly one triggering event, and thus exactly one entry point.

Unsolicited Events

Unsolicited events can occur at any time during script execution and usually indicate an action taken by a user of the application. For example, you could develop a conferencing application in which pressing the # key allows the original conference owner to drop out of the call without terminating the conference. In order to implement such a feature, the script is required to handle a CallControl.GotDigits although there is no guarantee that owner will press the # key.

Asynchronous Events

Asynchronous events (sometimes referred to as asynchronous callbacks) are similar to unsolicited events. Unsolicited events may occur at any time, whereas asynchronous events occur only in response to an asynchronous action.

For example, executing the action MediaControl.Play plays an audio stream to a specified destination. When MediaControl.Play begins execution, a provisional response is generated. See “Asynchronous Provider Actions” on page 14 for more information on provisional responses.

If **MediaControl.Play** receives a provisional response, one of the asynchronous events **MediaControl.Play_Complete** or **MediaControl.Play_Failed** is guaranteed to occur. **MediaControl.Play_Complete** occurs if the play was successful. **MediaControl.Play_Failed** occurs if the play was not successful. This guarantee is what distinguishes asychronous events from unsolicited events—an unsolicited event may or may not occur.

When working with asynchronous events, you may wish to utilize the event's UserData parameter. UserData can be set to some unique value in order that an asynchronous event can be associated with its particular generating action. The default value of the parameter is the string *none* but any non-empty string is valid.

The Application Runtime Environment automatically propagates UserData to the event handler, allowing you to programmatically associate a particular event instance with its causal action. In the previous **MediaControl.Play** example, it may be necessary to play multiple announcements. If so, distinguishing the different announcements may also be required.

When the **MediaControl.Play_Complete** or **MediaControl.Play_Failed** asynchronous event handler executes, the associated UserData parameter value can be assigned to a variable. The UserData parameter value is then used to identify the specific action that was responsible for the asynchronous **callback.each MediaControl.Play** action.

Hybrid Events

A hybrid event can be handled as either a triggering event or an unsolicited event. In the MCE, HTTP requests are managed as triggering events when they contain no routing data. However, HTTP requests are handled as unsolicited events when they contain valid routing data. MCE HTTP routing data can take any of the following forms:

- A metreosSessionID query parameter in the URL
- A cookie named metreosSessionID
- A HTTP header named metreosSessionID in the request

Since an initial request contains no routing data, it will always be treated as a triggering event. A script must assign routing data with any subsequent and related HTTP actions, in order that the HTTP session can continue to be identified.

Actions

Application scripts are constructed by linking together two or more *actions* using conditional logic. Actions are individual commands executing on behalf of the application script. Actions can be divided into three types:

- Provider — Actions performed as a request to a provider
- Native — Actions developed to extend the capability of the Application Runtime Environment
- Core — Actions performed within and handled by the Application Runtime Environment

Provider Actions

When a synchronous provider action is invoked, script execution is blocked until the final and only response for the action is returned. This differs from an asynchronous action, which is non-blocking. An asynchronous action returns an immediate provisional response, with the final response occurring later in the form of an asynchronous event.

Synchronous Provider Actions

Synchronous actions are simple request-response transactions within the Application Runtime Environment depicted in Figure 7:

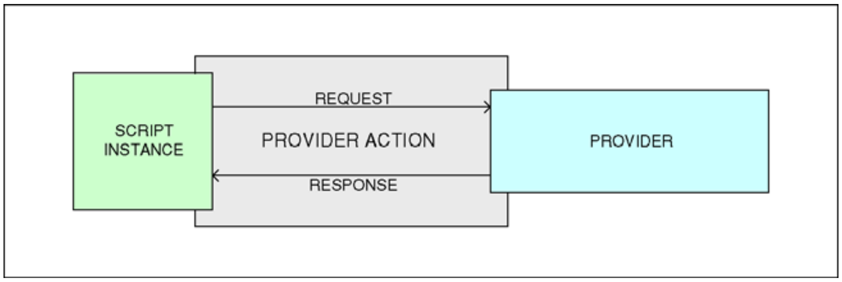

Figure 7: Synchronous Provider Action

The script first invokes a request. When the provider is finished processing the request or when an error occurs, an action response message is returned to the script instance and script processing continues.

From the time the original action request is sent until the time a response is received, the script instance is in a wait state pending a response to the action. This model works well for the majority of actions because the time required to process the request is typically short. For some types of actions an asynchronous model is more appropriate.

Asynchronous Provider Actions

As shown in the following diagram, asynchronous actions allow for a provisional response to be sent by the protocol provider before the final response is sent to the script instance.

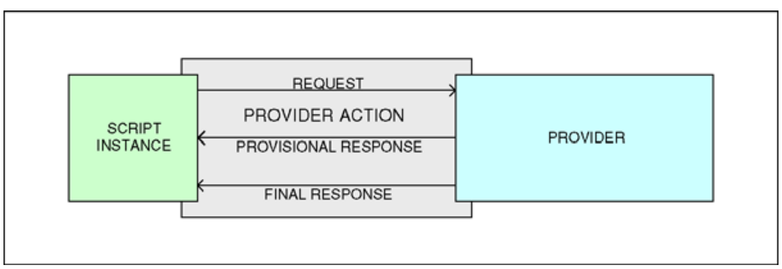

Figure 8: Asynchronous Provider Action

A script placing a call is an example of an asynchronous request requirement. The Application Runtime Environment does not wait for the call to be answered. Instead, the act of answering the call is handled as an asynchronous event elsewhere in the application.

A provisional response tells the client application that the provider is processing the request and notifies the application when the request has completed. A provisional response may also indicate that the request could not be serviced, in which case the provisional response is the final response.

After the provisional response is received the script instance continues executing normally. Blocking does not occur and execution continues at the next linked action. In this example, the provider has confirmed to the application that the call has been placed, the request is proceeding and the script can continue.

When the provider has finished processing the asynchronous action request, it sends an asynchronous event as its final response. When the application receives the final response the provider action is complete.

In contrast to an unsolicited event, an asynchronous callback is guaranteed if the provisional response for the action did not indicate error. Asynchronous events carry information allowing the virtual machine to map the event to the appropriate script instance and event handler.:

Native Actions

Native actions allow you to supply custom logic that executes within the process space of a script instance. This technique lets you define custom actions using more traditional programming methods. Native actions are always synchronous, and are best used for building application logic which:

- Has no need to persist beyond the lifetime of the script instance; and
- Does not monitor external network services.

Unlike provider actions, native actions cause the virtual machine to execute the action logic within the context of the script instance. For information on developing native actions refer to "Developing Native Actions" on page 81.

Core Actions

Core actions are handled internally by the Application Runtime Environment. All of the actions within the Metreos.ApplicationControl namespace are core actions. Core actions include services such as calling functions, exiting the current function and ending the application. All core actions are synchronous actions and a developer cannot add new core actions.

Scope

Scope refers to the accessibility of variables and code segments in different functional blocks. The Metreos MCE implements a robust scope mechanism designed to encourage a consistent yet flexible style of coding. Scope is best understood by reviewing its implications at each layer of the MCE development architecture.

Application-Level Scope

At the highest level, applications contain scripts and optionally include an installer, databases, and media resources. Each of these components has different scope requirements as described in the following information:

- Scripts — All scripts within an application are independent and invisible to one another. There is no way for a script to execute code, access variables or interrogate the elements of another script. Communication between scripts must occur through:
  - A database
  - A **Metreos.ApplicationControl.SendEvent** action (refer to the "Application Control" on page 148 of Appendix A for details)

- A **Metreos.ApplicationControl.Forward** action (refer to the "Application Control" on page 148 of Appendix A for details)

- Installer — The configuration settings defined by the installer are globally available from any part of the application but cannot be changed by the application itself. Instead, using the Metreos Management Console, an administrator can modify configuration settings for an application.
- Databases — Every database can be accessed from any script within the application. This capability permits scripts to communicate with one another or to share persistent data after all script instances have completed.
- Media Resources — All media resources can be accessed and played through any script within the application. Multiple scripts can play the same media resource at the same time.

Communication between applications must occur through an external service, such as a protocol provider or an external database.

Script-Level Scope

Scripts contain variables and functions. Script-level variables and functions can be accessed by any function within the same script, but cannot be accessed by other scripts.

Function-Level Scope

Functions comprise the lowest level of the MCE, and therefore have access to both script-level and application resources. Functions can also contain variables that can be set or modified by function elements but cannot be set by other functions, even those within the same script. Functions that must share information with each other must use script-level variables or function parameters and return values.

Application Resources

This section provides details about application resources mentioned in the previous section, Application-level Scope.

Installers and Configuration Parameters

Systems administrators must often set configuration parameters specific to a particular deployment. The MCE allows developers to define the configuration parameters, their types and default values by using application installers.

An application installer is an XML file similar to the following code:

```
<?xml version="1.0" encoding="utf-8" ?>
<install xmlns="http://metreos.com/AppInstaller.xsd">
   <configuration>
      <configValue name="CM_Address" description="CallManager ad-
dress"
                format="string" displayName="CallManager Address"
                   required="true" defaultValue="192.168.1.250"
                   readOnly="false" />

     <configValue name="DialPlan" description="A sample hashtable"
                   format="hashtable" displayName="Dial Plan"
                   required="true" readOnly="false" />

      <configValue name="CM_LDAP_Port" displayName="CM LDAP Port"
                   description="LDAP port"
                   format="integer" defaultValue="8404"
                   required="false" />
   </configuration>
</install>
```

Code listing 1. A Sample Installer

Application installers contain one or more configValue entries describing the various configurable elements of an application. A configuration entry can specify a default value for string, integer and boolean types. For example, the default value for the **CM_Address** configuration entry in the code fragment is **192.168.1.250**.

Most entry types in the installer can be represented as a string. However, hashtable and arraylist types cannot be represented as a string. Furthermore, default values cannot be specified for hashtable and arraylist data types. Values for these entries must be specified directly through the management console at install time.

Application installers are processed during the installation phase of the application lifecycle. The Application Runtime Environment creates corresponding database entries for each of the configuration values in the installer. The Application Runtime Environment then links installer configuration entries to the application being installed. Upon completion, all the configuration values specified in the installer are visible and editable using the management console.

Figure 9 depicts the application installer XML schema:

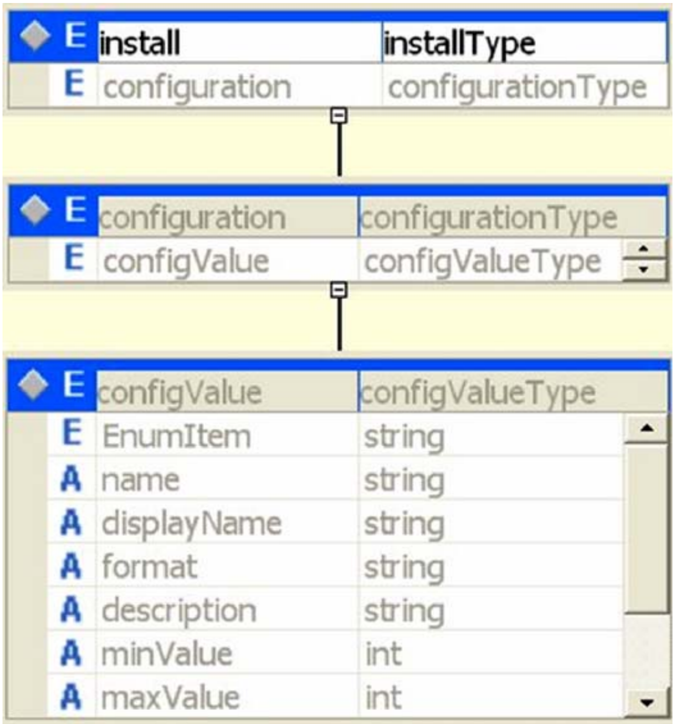

Figure 9: The Application Installer XML Schema

Each configuration entry must specify the name and format attributes. All other attributes are optional. The format attribute specifies the data type of the configuration entry. Valid format attributes include:

- String
- Bool
- Number
- DateTime
- IP_Address
- Array
- HashTable
- DataTable
- Password
- TraceLevel

You can also define a set of values called *enumerable format types*. Enumerable format types allow you to define a specific set of values. For example, use the **String** format attribute if any value is to be valid for a particular string attribute. However, if only specific values are valid in a particular string attribute (such as officePhone, homePhone and remotePhone), create an enumerable type and use that type.

You can define your own enumerable types, name them and then use them to supplement the types in the previous list. Valid enumerable format types include:

- **String**
- **Integer**
- **Hashtable**
- **Stringdictionary**
- **Boolean**
- **Password**

NOTE: ***Password*** *behaves in a similar fashion to a string, but the data is obscured in the Management Console and is encrypted for storage.*

Database Management

The MCE ships with a MySQL 4.1.5 database. To instantiate a database for use with your application, a schema creation script must be included in the application package. When creating a SQL script, you can code anything you might normally enter at the MySQL command line.

Your script will execute within the scope of a database that has already been created. For example, including **CREATE DATABASE [dbName];** and **USE [dbName];** in your SQL script is unnecessary because the Application Runtime Environment includes the **CREATE DATABASE [dbName];** and **USE [dbName];** commands before executing your script.

Inside MCE Applications 3

Because MCE applications are developed using a visual design tool, they can be thought of as a collection of concrete elements such as Web pages, actions, and custom features rather than code. This abstraction purposely masks the details of application structure to help you focus on application behavior rather than on grammar and syntax.

Application Architecture

Applications consist of four main components: an installer, scripts, databases, and media resources. Scripts contain variables and functions. The following diagram depicts the MCE application architecture:

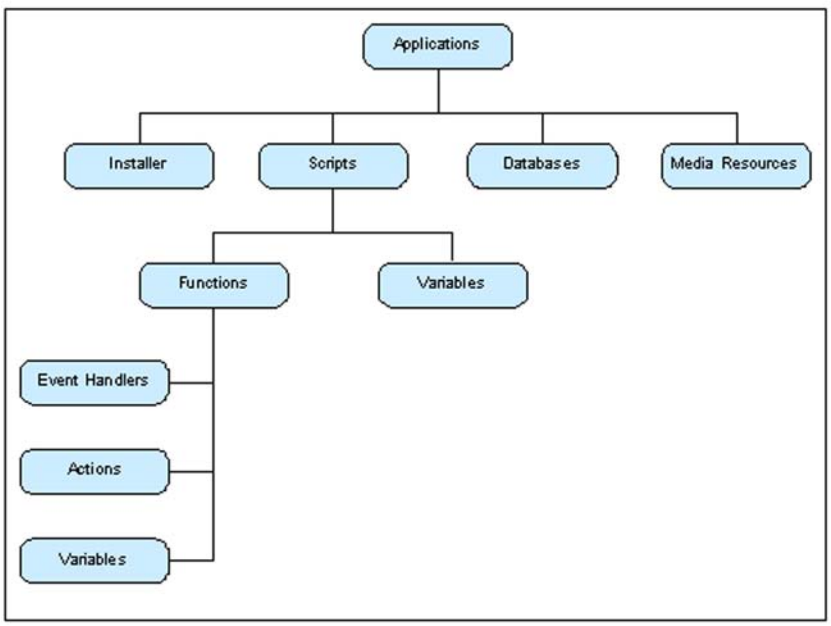

Figure 10: Application Architecture

When you develop an application using the Metreos Visual Designer, the system creates the components as depicted in Figure 10 based on the following information:

- The graphically represented nodes and variables you drag on to the canvas
- The way you connect the nodes
- The properties and values you assign to the nodes and variables

An application is defined by selecting **File → New Project** and then selecting **File → Add Script** from the Visual Designer user interface. A triggering event must be defined for each script. At the time the triggering event is specified, a function to handle the event is created for you. As for all functions, this function is initially empty and must be populated with actions, variables, and application logic from the toolbox.

Asynchronous actions, such as MakeCall, require a handler function for each of the asynchronous events that might be launched by the action. Stub handler functions for each such event are automatically created for you at the time you drop such an action onto a canvas.

You can add additional scripts as needed to complete your application. When the application is complete, or when you are ready to test some part of it, select **Build** → **Build Project** on the Visual Designer user interface. The Visual Designer then compiles the application, displays any error messages, and creates the application package. If the application is complete and the compiler writes no error messages, you can deploy the application by selecting **Build** → **Deploy**. Refer to "Using the Metreos Visual Designer" on page 36 for details.

XML-Based Implementation

The internal language of MCE applications is derived from XML. The Metreos Visual Designer provides a drag-and-drop application canvas. To build a function within an application script, drag and drop nodes from the toolbox on to the canvas, and then connect them with arrows as shown in Figure 11. The visual designer generates the XML when you select Build Project on the Build menu.

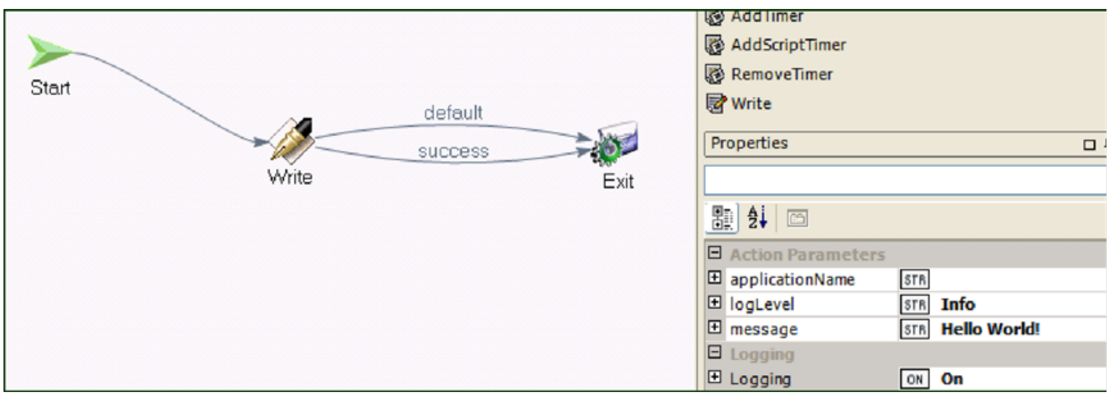

Figure 11: "Hello World" Application Script

When the Visual Designer compiles the application from the visual source code, it generates internal XML code corresponding to the Metreos Application Script XML schema. This XML can be thought of as the intermediate code used by the Application Runtime Environment for execution. Code Listing 2 shows the intermediate code for a simple "Hello World" application script for the MCE.

```
<?xml version="1.0" encoding="utf-8" ?>
<serviceApp name="Hello World" type="master" instanceType="multiIn-
stance"
   xmlns="http://metreos.com/ServiceApp.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

   <globalVariables>
  <variable name="myGlobalVariable" type="Metreos.Types.String" />
   <globalVariables>

   <function id="PrintLogMessage" firstAction="1">
    <variable name="myLocalVariable" type="Metreos.Types.String" />
     <event type="triggering">Metreos.Providers.Http.GotRe-
quest</event>

     <action id="1" type="native">
 <name>Metreos.Native.Log.Write</name>
       <param name="Message">Hello World!</param>
       <param name="LogLevel">Info</param>
       <nextAction returnValue="success">2</nextAction>
       <nextAction returnValue="default">2</nextAction>
     </action>

     <action id="2">
       <name>Metreos.ApplicationControl.EndScript</name>
     </action>
   </function>
</serviceApp>
```

Code listing 2. "Hello World" Application Script Intermediate Code

Because applications are internally represented as XML, they can be stored on almost any computing platform and transported across virtually any network infrastructure. The assembler converts this XML into streamlined object code while checking and rejecting malformed applications.

Execution Model

The building block of an MCE application is an application script. Each script represents a potential thread of execution for the MCE application. Scripts begin execution when the Application Runtime Environment receives a triggering event containing parameters matching the triggering criteria of an application script. The Application Runtime Environment then creates and executes a new instance of the script.

A script resides in memory until triggered. Upon receipt of the triggering event, the script is forwarded to the scheduler for execution. New script instances are executed by the Application Runtime Environment, and each script instance executes in a separate thread. Each time a triggering event is received, a thread from the MCE virtual thread pool is assigned to the new instance of the script. When a script finishes executing, it is removed from the scheduler, reset, and returned to the repository for potential reuse.

When script instances are executed, the process is managed as a state machine. Figure 12 shows a state machine diagram that models how application script instances behave during execution.

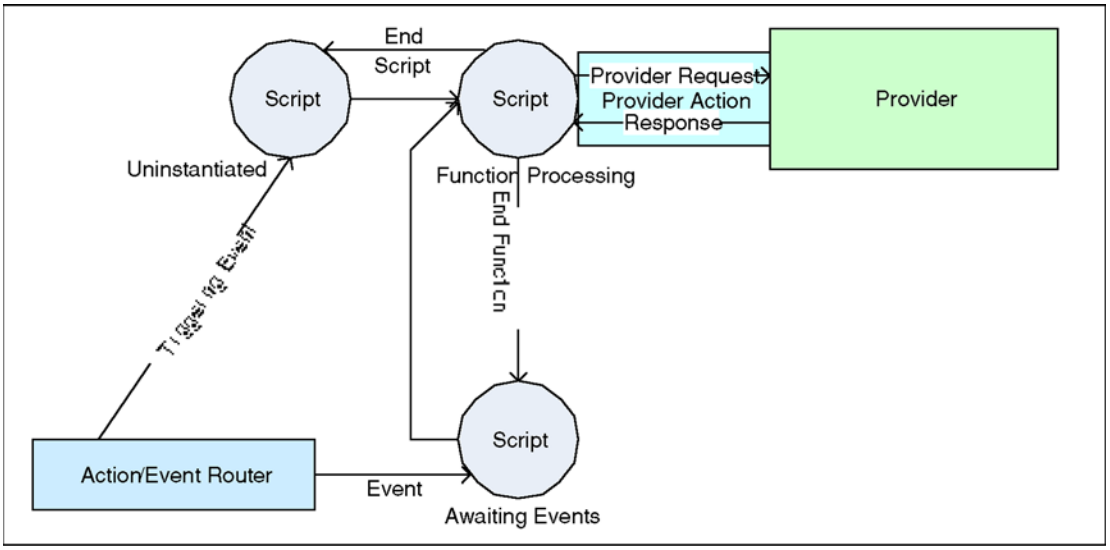

Figure 12: The State Machine for Application Script Instances

A script in the uninstantiated state is waiting for a triggering event. After the triggering event is received, an instance of the script is executed and a unique system identifier (the Routing GUID) is assigned to it.

When the script instance is executed, the instance enters the *function processing* state. In this state, events are placed in a queue and processed in the order that they are received until the script instance encounters an **EndFunction** action. The script instance then enters the *awaiting events* state.

While the instance is awaiting events, subsequent events can be routed to the script instance. When an event is received, the function transitions back to the function processing state. For a script to terminate, an **EndScript** action must be executed for it. For this reason, an **EndScript** action should be present in every script.

Once the **EndScript** action is encountered, the instance is destroyed and the script returns to the uninstantiated state.

Application Script Elements

Script processing occurs inside units of code called functions. As in most programming languages, all logic occurs within functions or is controlled by functions. However, variables, configuration and initialization parameters may exist outside the functions.

While a script may contain many functions, only one of those functions serves as the event handler for the triggering event of the application script. A new script instance is started each time a triggering event is received. After it is received, the triggering event handler function is started, which marks the beginning of execution for the new script instance.

Application Script Triggers

As described in “Triggering Events” on page 12, a triggering event is an event containing a signature matching the specified signature of an existing script. Figure 13 depicts the trigger used to start the example Hello World script:

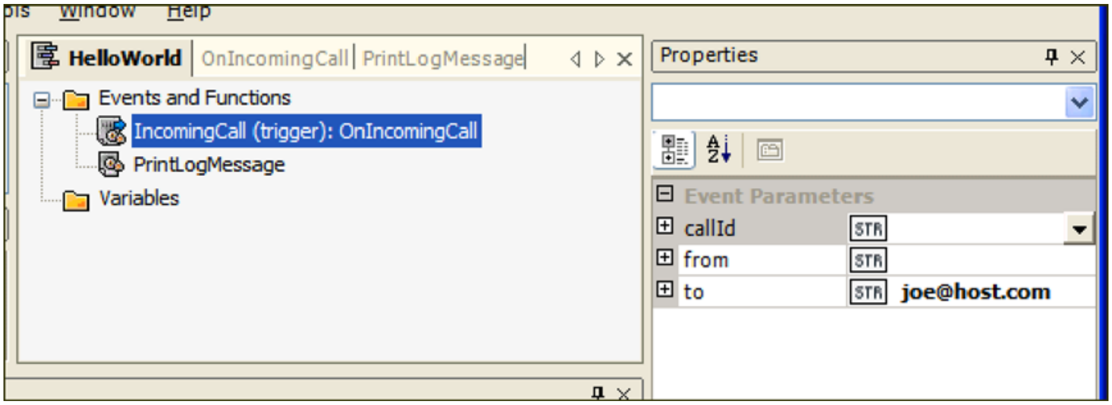

Figure 13: Application Script Trigger Screenshot

In this example, a new script instance begins when an event of type **Metreos.CallControl.IncomingCall** containing a *to* parameter with a value of **joe@host.com** is received. Multiple script types may trigger on the same event type and the same event parameters; however, the matching criteria for those event parameters must be unique. The unique parameters constitute the triggering event signature.

The triggering event signature must contain at least one event type. Even though it is not required by the MCE, Metreos recommends the specification of at least one additional parameter so that the script will have a a unique triggering signature. If two scripts have identical triggers, one of the scripts will be triggered by the event, but you cannot predict which script will be triggered. Therefore, use one or more additional parameters to create a unique trigger signature.

Functions

As with most programming languages, the MCE allows you to group application script logic into functions. The Metreos Visual Designer allows application developers to use *event handler* and *standalone* types of functions. An event handler is associated with a particular MCE event and is assigned to handle that event. A standalone function is explicitly called by the application script.

Create a standalone function by dragging a **CallFunction** node from the Application Components tab of the Toolbox as shown in Figure 14.

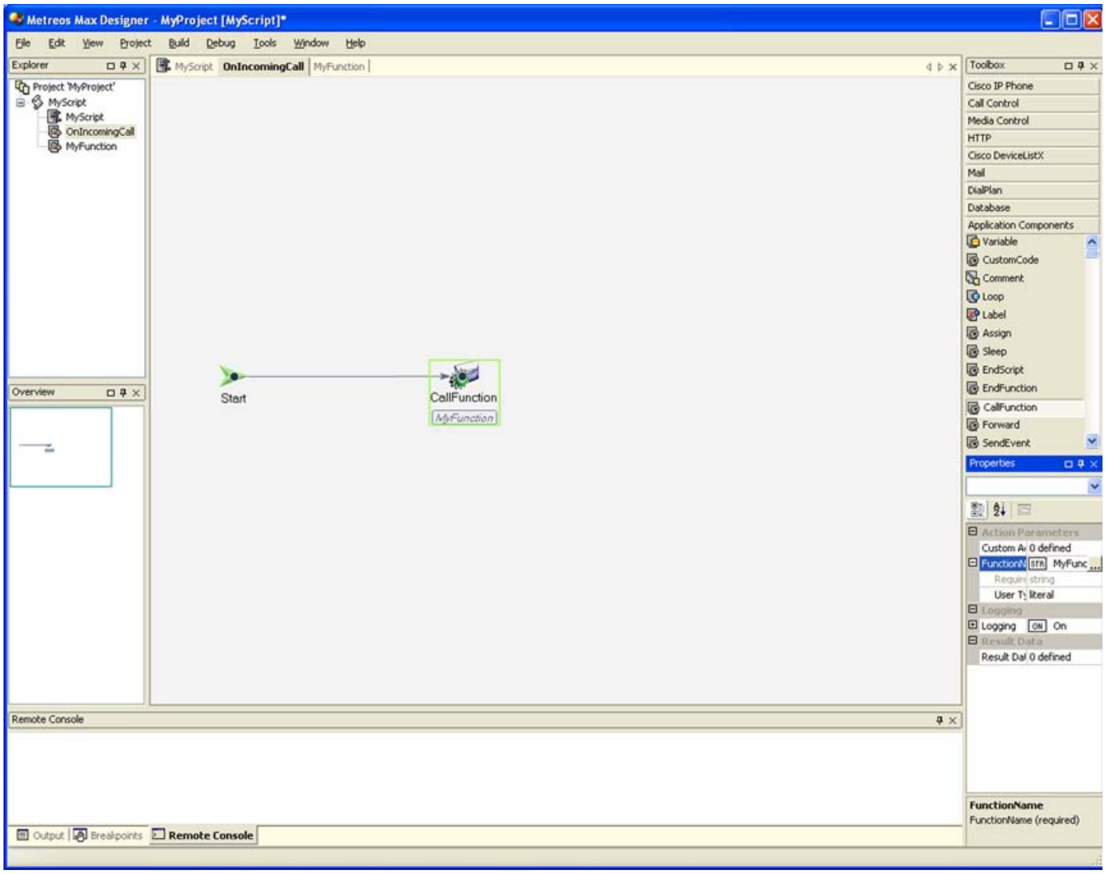

Figure 14: Creating a Standalone Function

As shown in Figure 14, the Visual Designer creates a new function for you with the name you entered as the **CallFunction** label if a function of that name does not already exist. In the example diagram, the triggering script will invoke **MyFunction** when the triggering call is received.

Each function has a signature that is used to call that function. Function signatures, much like the triggering event signature, must be unique within the script. The primary difference between function signatures and triggering event signatures is that of scope. Triggering event signatures must be unique to the application script compared to all other application scripts. Function signatures must be unique only within the application script to which they belong.

Variables

MCE variables allow you to store and manipulate data within application scripts. Variables may be initialized either with a constant value or from parameters arriving with events. Variables may be assigned script (global) or function (local) scope. A script variable may be used in all functions within a script. A function variable can be used only in the function in which it is defined.

Both script and function variables can be initialized with a value as shown in Figure 15.

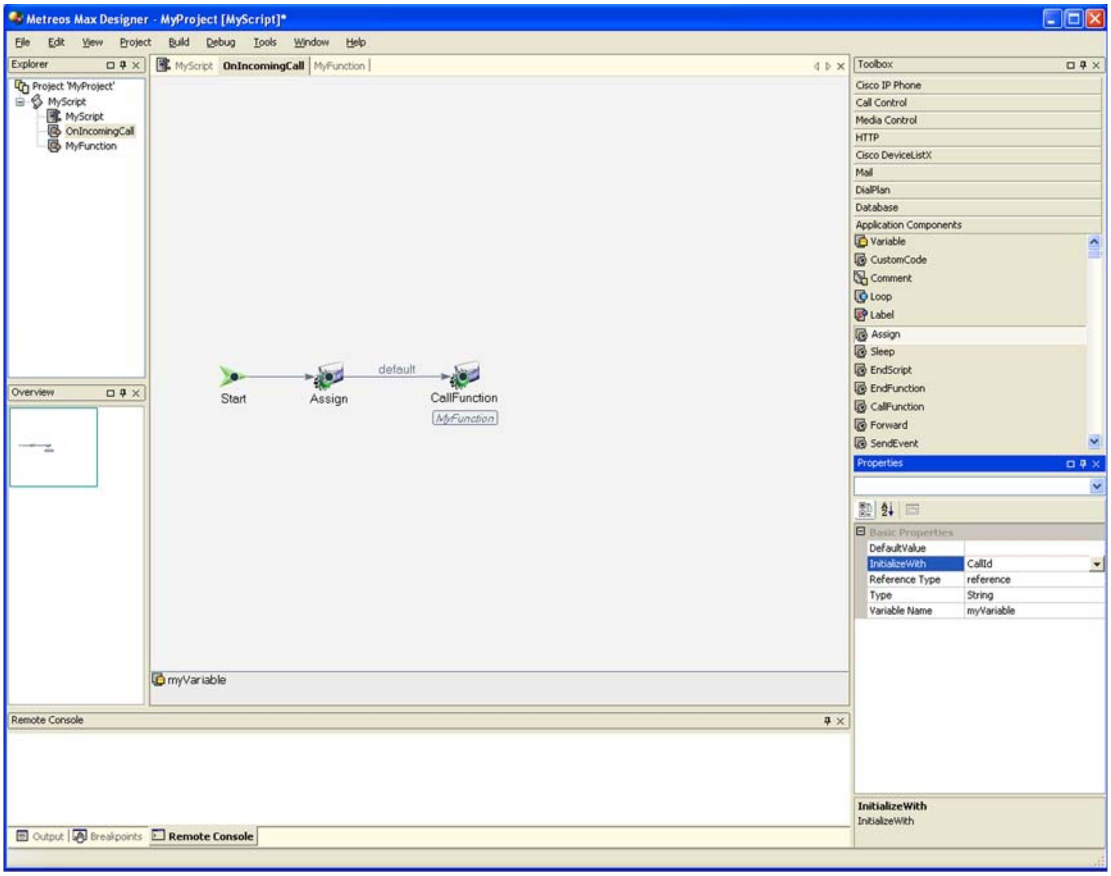

Figure 15: Adding a Function Variable Initialized with CallId

When a new instance of this script starts execution, the Application Runtime Environment initializes the function variable (myVariable) with the value of the **CallId** parameter. Script variables can be initialized only with values from the configuration database using the application's installer.

Script variables can also be assigned the value of a function variable as shown Figure 16.

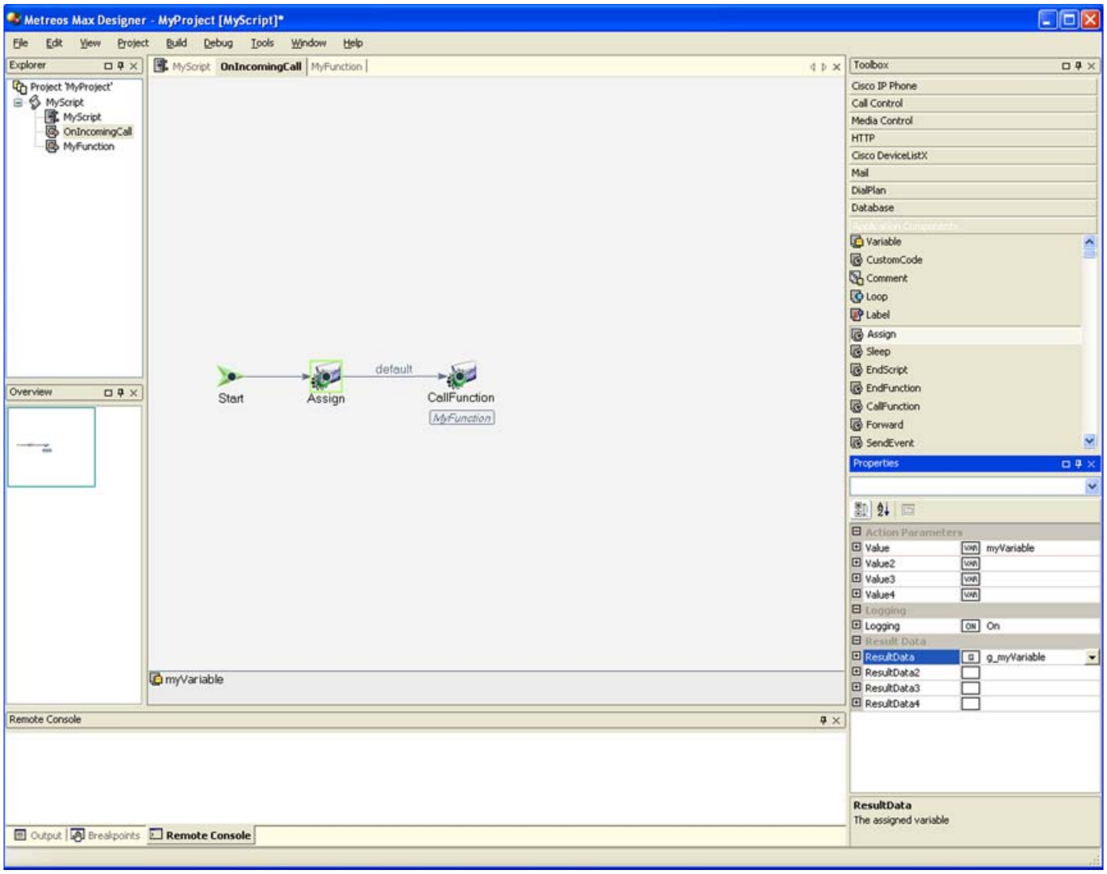

Figure 16: Assigning the value of a Function Variable to a Script Variable

In this example, a script variable (**g_myVariable)** is assigned the value of **myVariable: CallId**. Because **g_myVariable** is a script variable, the value of **CallId** is now available to other functions in the script, such as **MyFunction**.

Actions

The logic flow of a script is described by the linking together of actions on a canvas. In Figure 16, the project depicts the following logic.

1. A call is received.
2. The **CallId** value for the call is stored in a function variable (**myVariable**).
3. The value of **myVariable** is assigned to a script variable (**g_myVariable**).
4. The **MyFunction** function is invoked.
5. The call is answered (**AnswerCall** action).
6. The script ends.

Figure 17 depicts the use of an **AnswerCall** action in **MyFunction**.

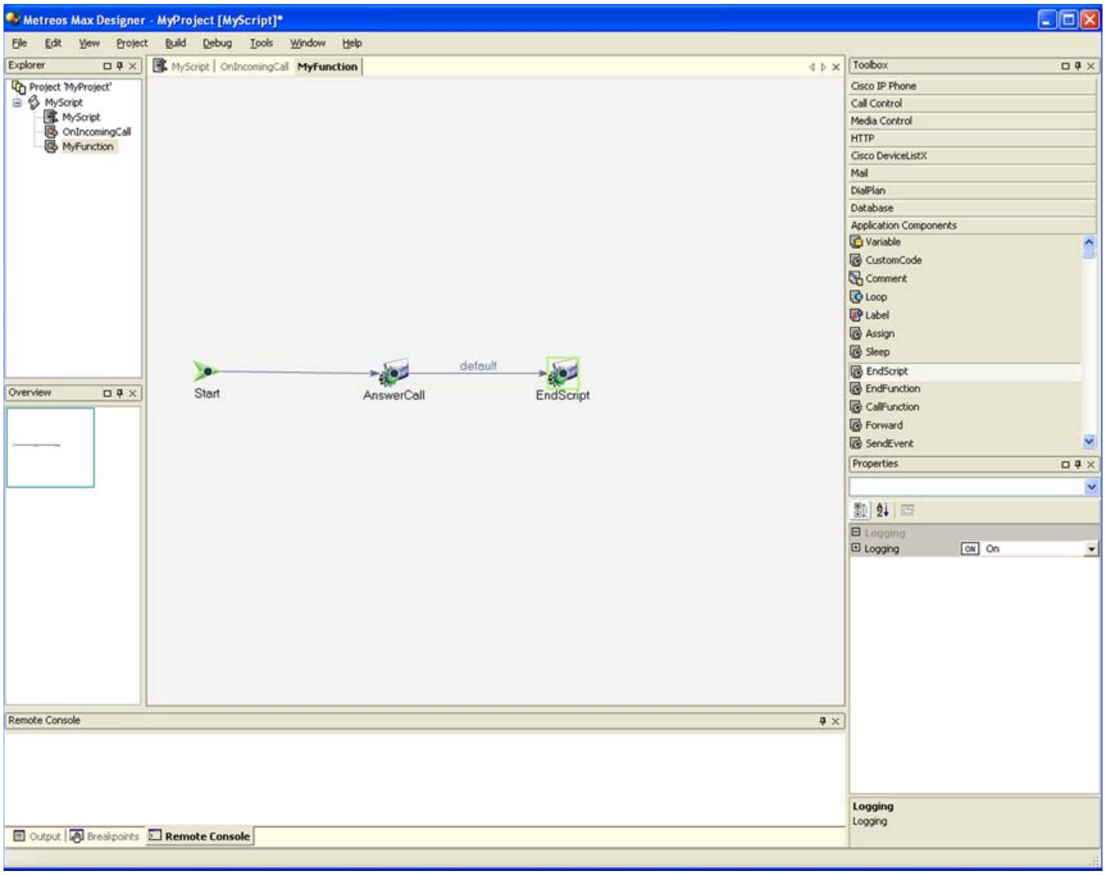

Figure 17: AnswerCall Action

Metreos Visual Designer 4

The Metreos Visual Designer minimizes the need to write code during application development. The Visual Designer displays one or more function canvases on which graphical representations of actions may be dropped to build script functions as shown in Figure 18.

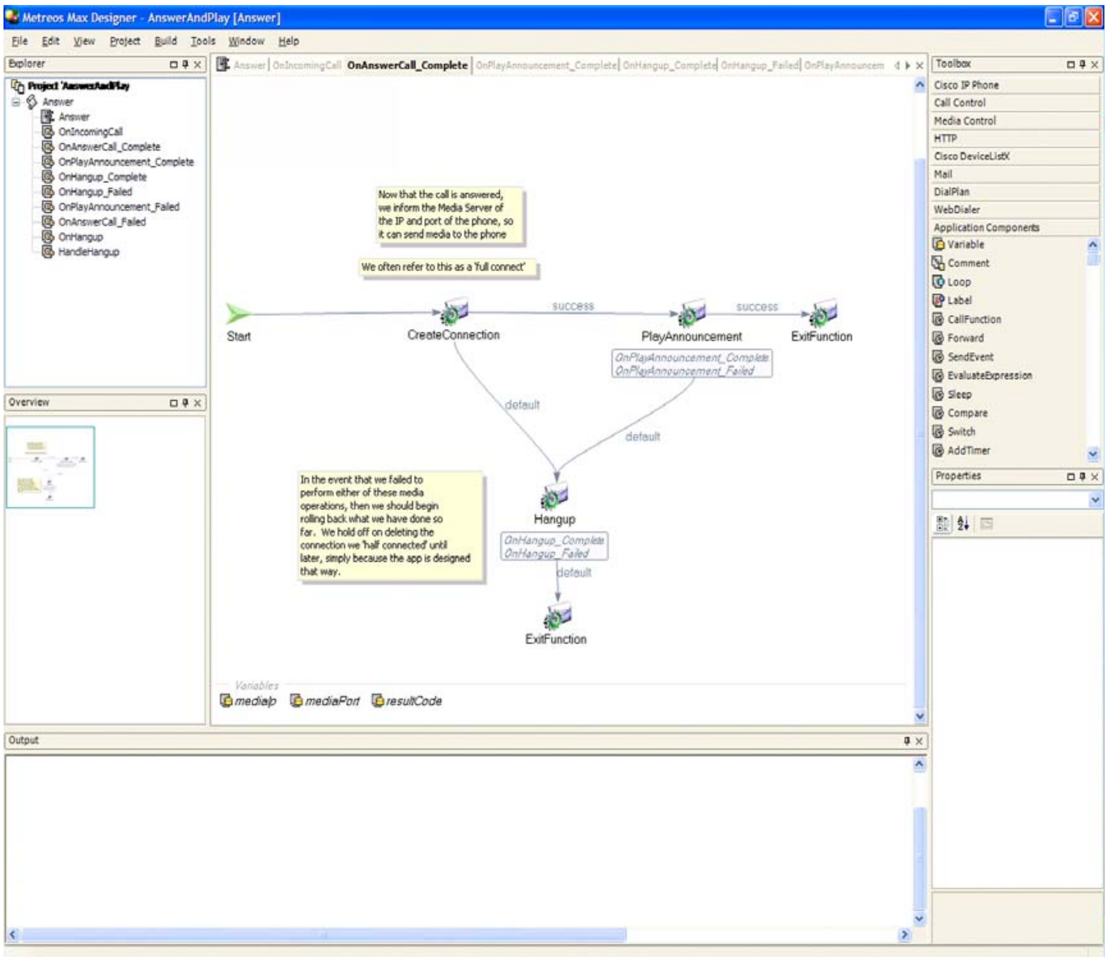

Figure 18: The Metreos Visual Designer

Elements of the telephony application are displayed as nodes connected with routing arrows. Each arrow represents a conditional path in the flow of programmatic execution.

A label on each arrow represents the condition under which the program proceeds to the action to which it is connected. For example, the diagram shows that the program proceeds from **CreateConnection** to **PlayAnnouncement** only if **CreateConnection** succeeds.

The success path is indicated by the label on the arrow connecting **CreateConnection** and **PlayAnnouncement**. If **CreateConnection** does not succeed, the program proceeds to **HangUp** using the default branch, indicated by the **Default** label on the arrow connecting **CreateConnection** and **PlayAnnouncement**.

For each function in the script there exists a canvas. When a function is created, a new canvas tab is added to the tab strip located above the canvas area. Display function canvases one at a time by clicking the tab labled with the function name.

Metreos Visual Designer Tour

The Metreos Visual Designer, as depicted in Figure 19, serves as the integrated development environment for MCE telephony application.

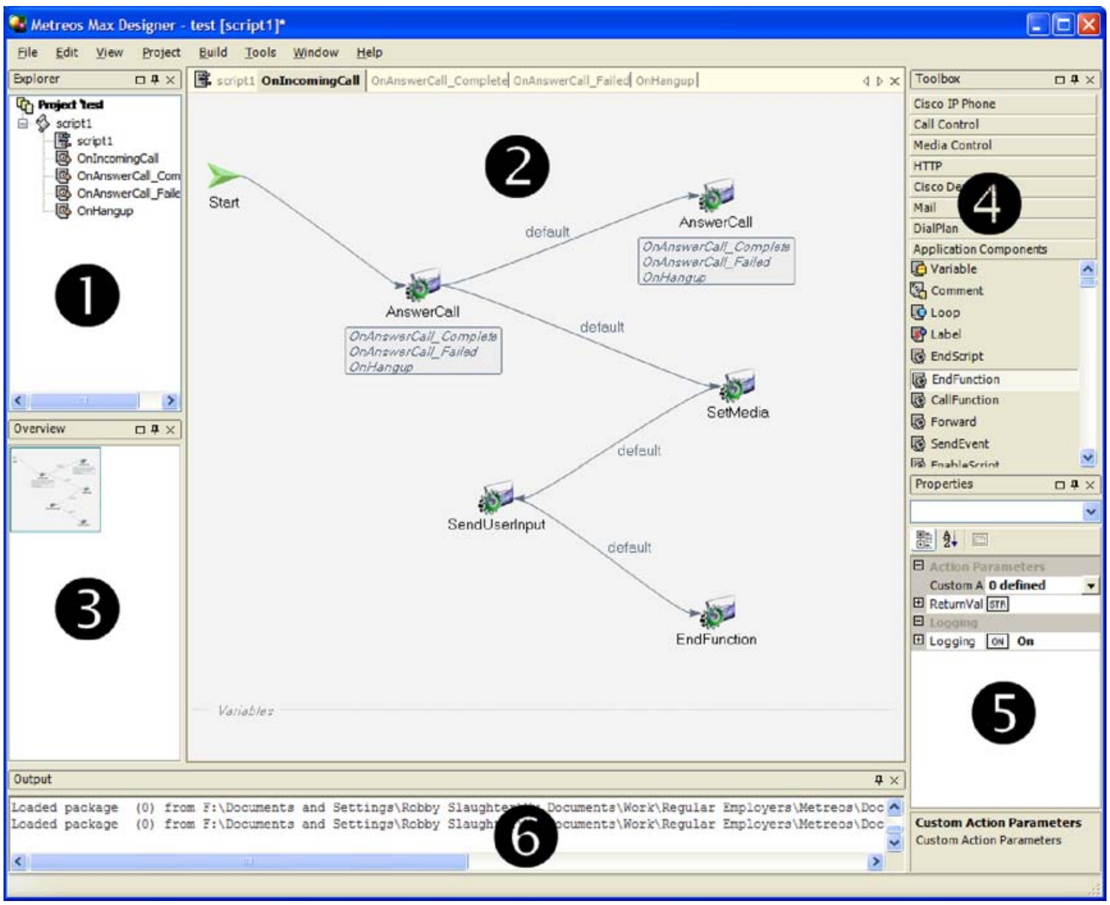

Figure 19: Windows in the Metreos Visual Designer

Explorer Window (1)

The Explorer Window lists all chief elements of the currently loaded application, grouped by category. The highest category that contains all others is the **Project**, which represents the application. The Components can include scripts, installers, SQL scripts, library references, and media resources. For details on these elements, refer to the "Application Architecture" on page 21. The Explorer Window is the main navigation system for the Visual Designer. It allows you to easily switch between each part of the application.

Application Canvas (2)

The Application Canvas is the heart of the Metreos Visual Designer where you can create diagrams on the canvas using application elements such as actions, control structures and variables. As you add elements to a function, the screen may not be large enough to simultaneously display all the nodes the function contains. In this case, the canvas automatically expands to accommodate additional elements displaying horizontal and vertical scroll bars that can be used to navigate the canvas.

Overview Window (3)

The overview window displays a miniaturized representation of the current application canvas. The blue rectangle overlaid on the window indicates the area of the canvas currently visible on screen. The Overview Window is useful in navigating the application canvas when the function contains too many components to be viewed in its entirety.

Toolbox (4)

Application elements reside in the Toolbox, which contains several tabs with each tab labeled with the name of a category. You can view the list of nodes for a specific category by clicking on the tab with that category label. Each one of the displayed nodes can be dragged from the Toolbox onto the canvas. Third-party elements can be added to the Toolbox by developing new providers or native actions. Refer to "Native Actions and Native Types" on page 81 for more information.

Property Grid (5)

Almost all application elements require configuration. You can view and modify the properties of an element through the Property Grid, which displays the properties of the node when the node is selected. The properties are grouped by category or alphabetically. To switch between these two options, click the grouping button or the alphabetical button.

To edit properties, click on the property you want to modify and change the value. Some properties allow only a range of values. In these cases the values are displayed in a dropdown menu. Complex properties contain a small plus sign to indicate that they can be expanded.

The Property Grid also allows access to the following two editors to assist you in providing coding logic and values to parameters:

- Code Editor
- Literal Editor

Figure 20 depicts the Code Editor.

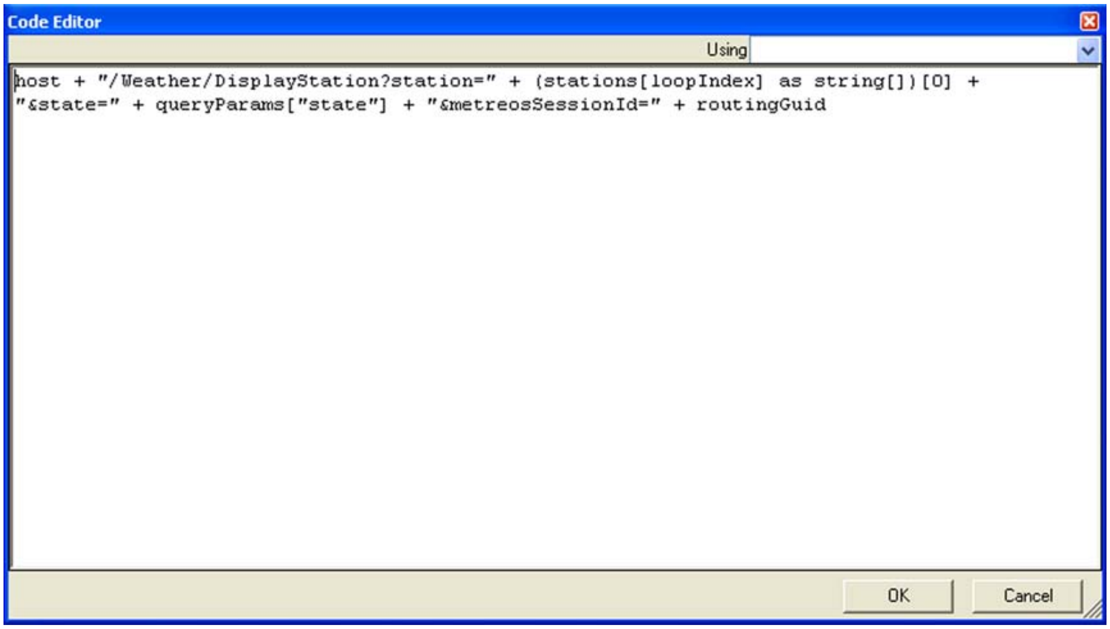

Figure 20: Code Editor

The Code Editor is used to modify element parameter values of types C# and Literal. Using the Code Editor to modify parameter values is optional. You also have the option of modifying parameter values directly in the **Parameter Value** in the Property Grid. However, it is useful to use the Code Editor to modify parameters containing lengthy values.

Access the Code Editor by clicking on the action parameter you want to modify in the Property Grid. If the parameter type has been set to C# or Literal, the Visual Designer displays a square containing an ellipsis (…) to the right of the element as shown in Figure 21.

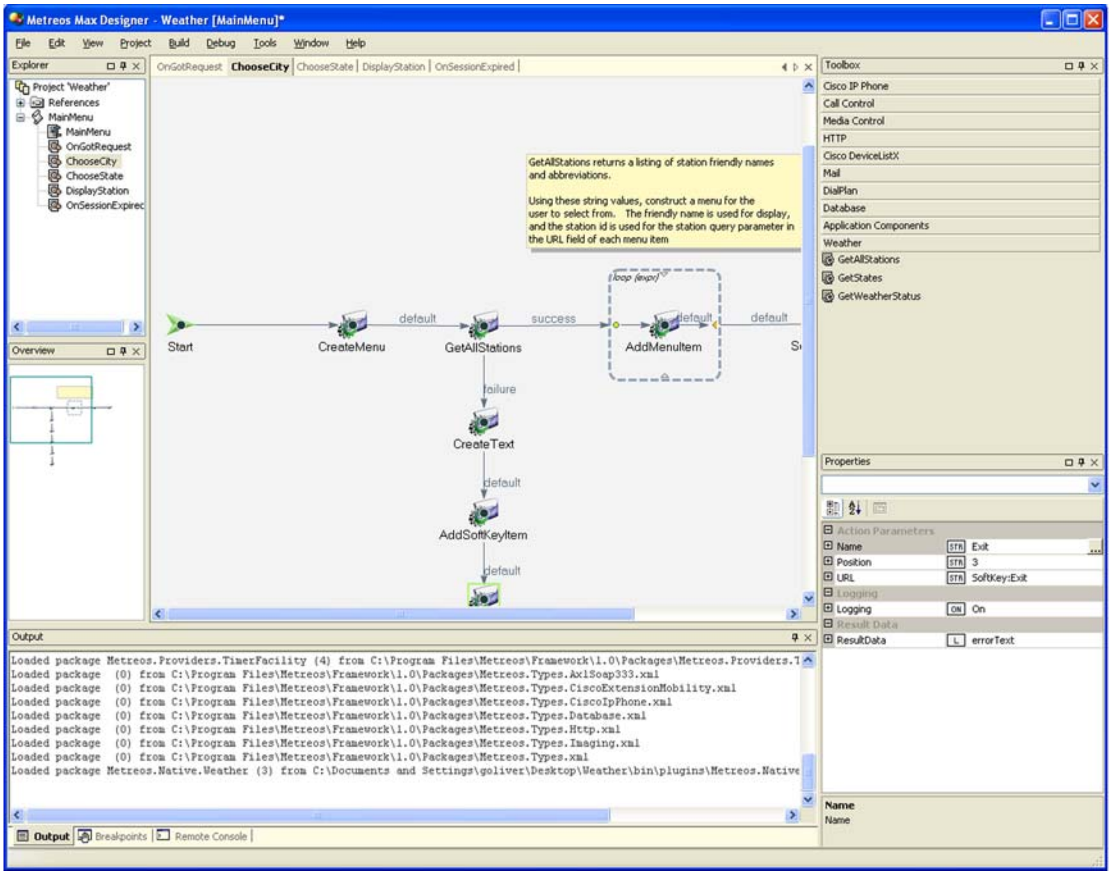

Figure 21: Accessing the Code Editor

Clicking the ellipsis displays the code editor.

The Literal Editor, shown in Figure 22, is used to edit literal values and is visually and functionally similar to the Code Editor.

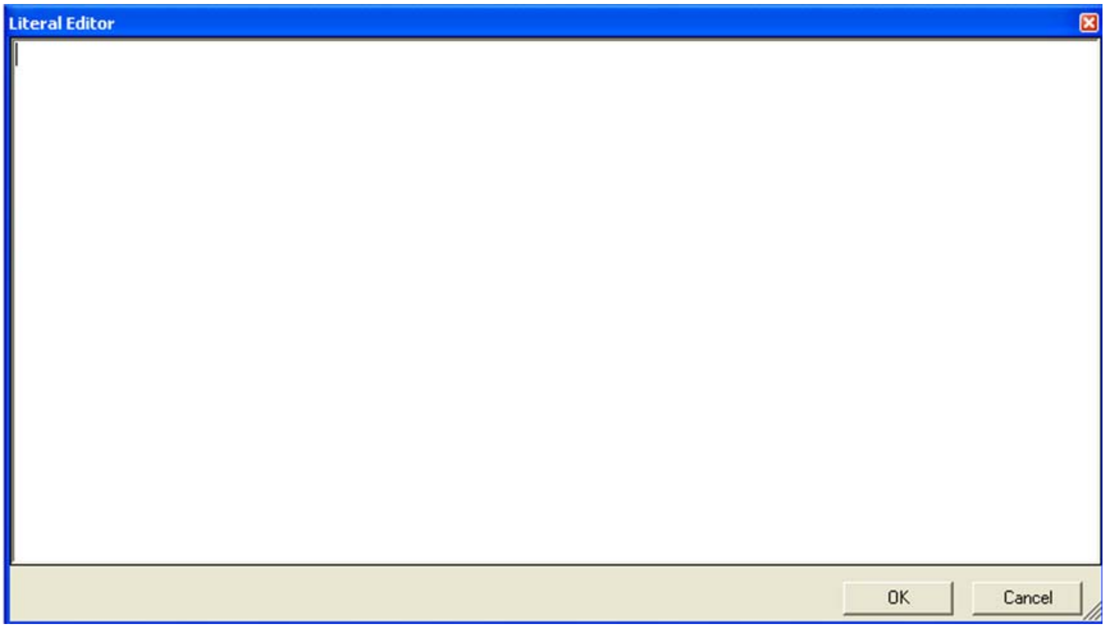

Figure 22: Literal Editor

Access to the Literal editor is similar to access to the Code Editor. Literals in the Property Grid contain a value field marked with an ellipsis (…). Clicking the ellipsis launches the Literal Editor.

Output Window (6)

When you compile an application, the compiler generates informational, warning, and error messages that are displayed in the Output Window.

Projects and Files

The Metreos Visual Designer organizes each application as a project. When you create a new project, the Visual Designer creates a folder with the new project name. The Metreos Visual Designer stores all the application component files in the project folder.

WARNING: The Visual Designer does not prevent you from manually adding files to the application package, such as notes, documentation or other components. However, Metreos does not recommend or support the practice of using tools other than the Metreos Visual Designer. Doing so can produce unexpected results.

Using the Metreos Visual Designer

Before creating your first application with the Metreos Visual Designer, learn to operate the Visual Designer interface.

Creating Projects

Creating a new application begins with the creation of a new project. Select **File → New Project** to display the **New Project** dialog. Enter the name of the new application, and select a location for storing the project folder on your computer or an available network drive. If necessary, you can move the project to a new location at a later time by relocating this folder.

After creating a new project, an icon is presented in the Explorer Window adjacent to the name of your new project. Right click this node and select **Properties** to display project-wide properties in the Properties Window.

Creating a Script

The script is the largest and most important grouping within an application. Every application must have one or more scripts. To create your first script, select **Project → Add Script → New Script** from the main menu. The Visual Designer presents a New Application Script dialog as shown in Figure 23.

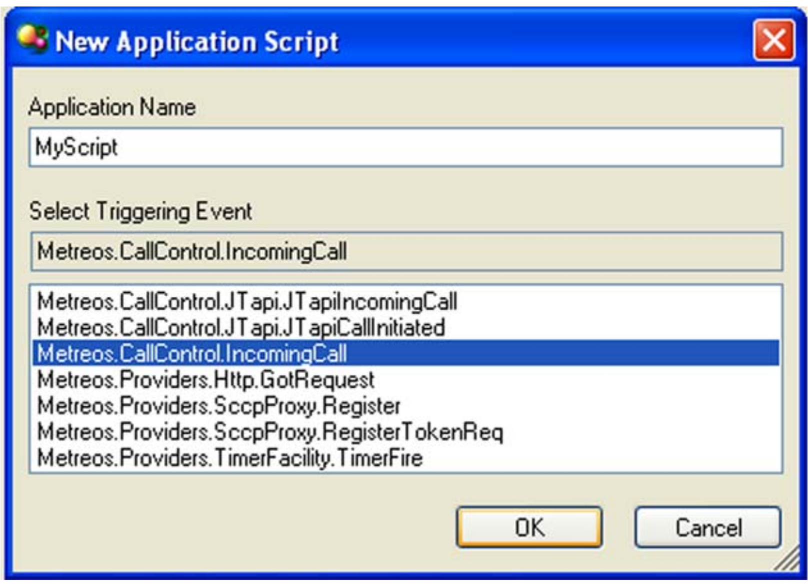

Figure 23: Creating a New Script

Scripts require a unique name, and must be triggered by some external event. Enter a name in the text field and select a triggering event from the **Select Triggering Event** list. Click **OK** to continue.

The Visual Designer creates the new script using the name provided. The canvas is updated with a new tab representing the script and a new tab that represents the triggering event handling function as shown in Figure 24.

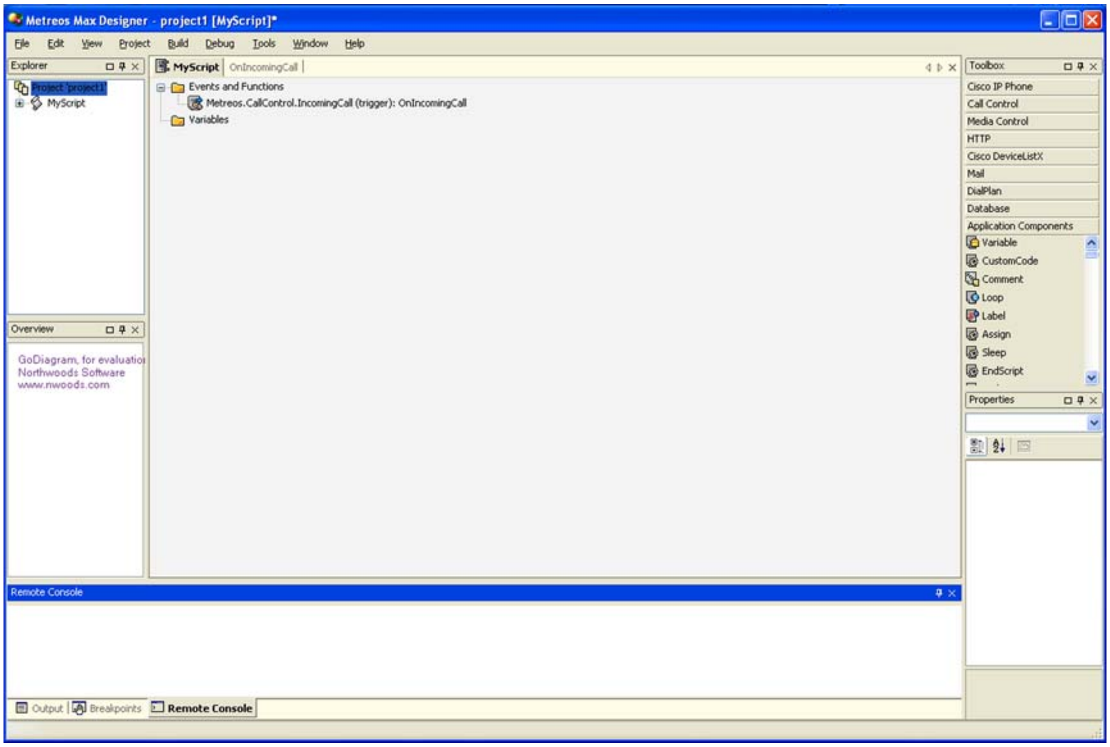

Figure 24: Sample Project Containing a New Script

Additional Scripts and Script Navigation

Visual Designer imposes no limit on the number of scripts in your projects, and scripts can be used in multiple applications. To add additional scripts, repeat the Creating a Script process. You can also add existing scripts using the **Project → Add Script → Existing Script** menu option.

When navigating from one script to another after you have modified the current script, the Visual Designer asks if you want to save the current script. You cannot have unsaved changes in multiple scripts simultaneously.

Event Handlers and Triggering Events

When you create a script, the Metreos Visual Designer creates a function in the Events and Functions folder of the new script. It functions as the event handler for the triggering event you specified and is the first function executed within the script.

Manipulating Elements on the Application Canvas

Select an event handler by first double clicking on a script name in the explorer window. Clicking the first tab on the left above the canvas presents the application script. Clicking the second tab on the left presents the first script function that is executed. This first function is the event handler for the triggering event.

Click the second tab. The Visual Designer displays a light gray background with an Start icon in the shape of an arrow as shown in Figure 25.

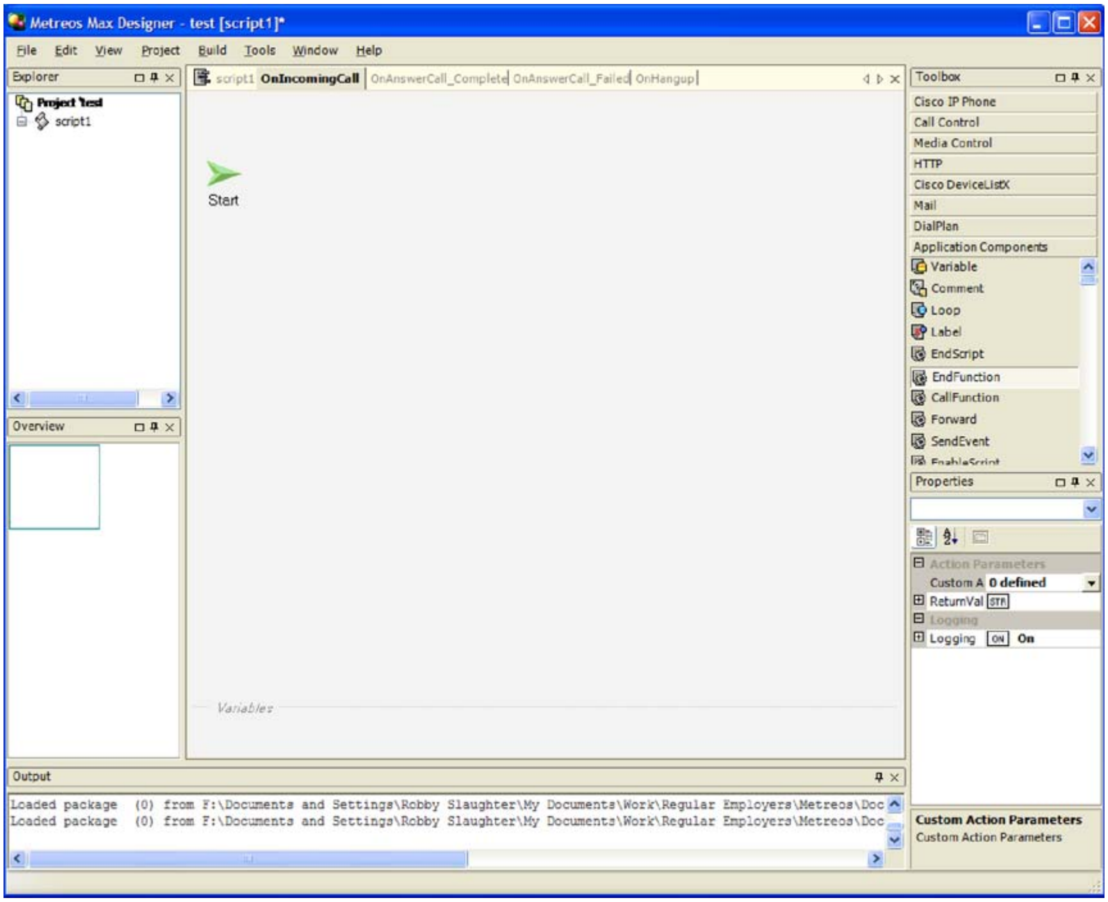

Figure 25: Blank Application Canvas

Every function execution begins with the start node, represented by the green **Start** icon. To move the start node, place your mouse cursor on the node label, left click the mouse and drag the node to a new location.

To create a new node, go to the Toolbox and select a category, such as HTTP. Then, drag and drop an element (such as **SendResponse**) from the toolbox onto the application canvas.

The Metreos Visual Designer automatically connects the start node to the newly created **SendResponse** node using an arrow. The connection indicates that immediately after the function begins executing, control will be passed to the **SendResponse** action.

To delete an arrow, a node or any other element on the application canvas, left click once on the title or border of the element and press the delete key. Optionally, you can also right click on an element and select **Delete**.

You can also select a group of elements by clicking and dragging a box around a set of nodes. Then, you can move or delete the elements as a group.

Manipulating Execution Paths on the Application Canvas

On the application canvas, arrows indicate the path of execution from one element to another. To draw an arrow between two elements, place your mouse cursor in the node, left click and drag the cursor to the node you want to connect. The arrow will point to the second node, indicating the program flows from the first node to the second node.

Execution paths sometimes depend on the result condition of a previous element. In this situation, a text description appears near the midpoint of the arrow. Click once on this description to convert it to a drop down box and select the desired option.

Finally, to help keep application canvas appearance clean, you can change the style of an arrow by right clicking near the middle of the arrow. You can select **curve**, **line** or **bevel**.

Elements That Create Handlers

Some types of elements in the toolbox automatically create additional event handlers. These nodes (such as **Call Control** → **MakeCall**) include a list of names in italics displayed below the icon as show in Figure 26.

Figure 26: Example Action

These items are event handlers. For each event handler, the Visual Developer creates a new function for you, and each new function has a corresponding tab above the application canvas. To complete your application, you must complete these functions to resolve the outcomes of the events.

Special Elements: Variables and Other Application Components

Among the components located in the **Application Components** tab of the Toolbox, are utilities including:

- Loops
- Comments
- Variables

Loops

Loops graphically enclose other elements. Elements enclosed by the loop are executed multiple times depending on the loop properties.

Comments

Comments can be directly edited and automatically resize to fit the new text.

NOTE: *In a comment, SHIFT+ENTER produces a line break.*

Variables

Variables store data within a function for easy access and are created by dragging and dropping the element into the Variables Tray located at the bottom of the canvas. By default, the Variables Tray is not visible if the function does not contain a variable. To view the Variables Tray, shown in Figure 27, right click on any blank area of the function canvas and select **Variables Tray**.

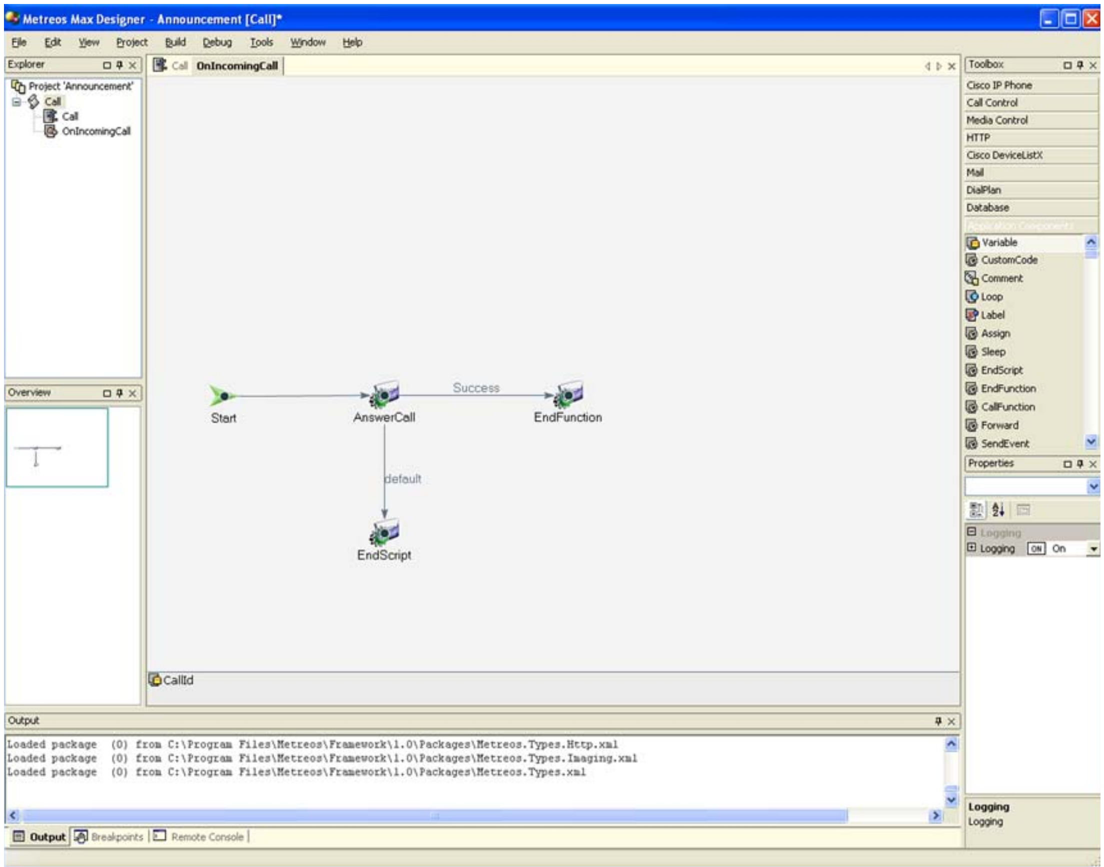

Figure 27: A Simple Function with a Single Variable

The **CallId** variable in the variables tray is a local function variable for the **OnIncomingCall** function. To rename a variable, click once on the variable name and type a new name, or right click the variable and select **Rename**. To delete a variable, click on the variable icon and press the delete key or right-click on the variable and select the **Delete** option.

As previously stated, the Visual Designer supports function variables and script variables. Function variables are available only within the function in which they are defined. Script variables are available to all functions within a script.

In the previous example, **CallId** is a function variable and is available only in the **OnIncomingCall** function. To define a global script variable, go to the script tree and right click the **Variables** branch. Then, select **Add Item**. The Visual Designer adds a variable icon to the tree and displays it in the Property Grid. Then, you can specify the variable properties as shown in Figure 28.

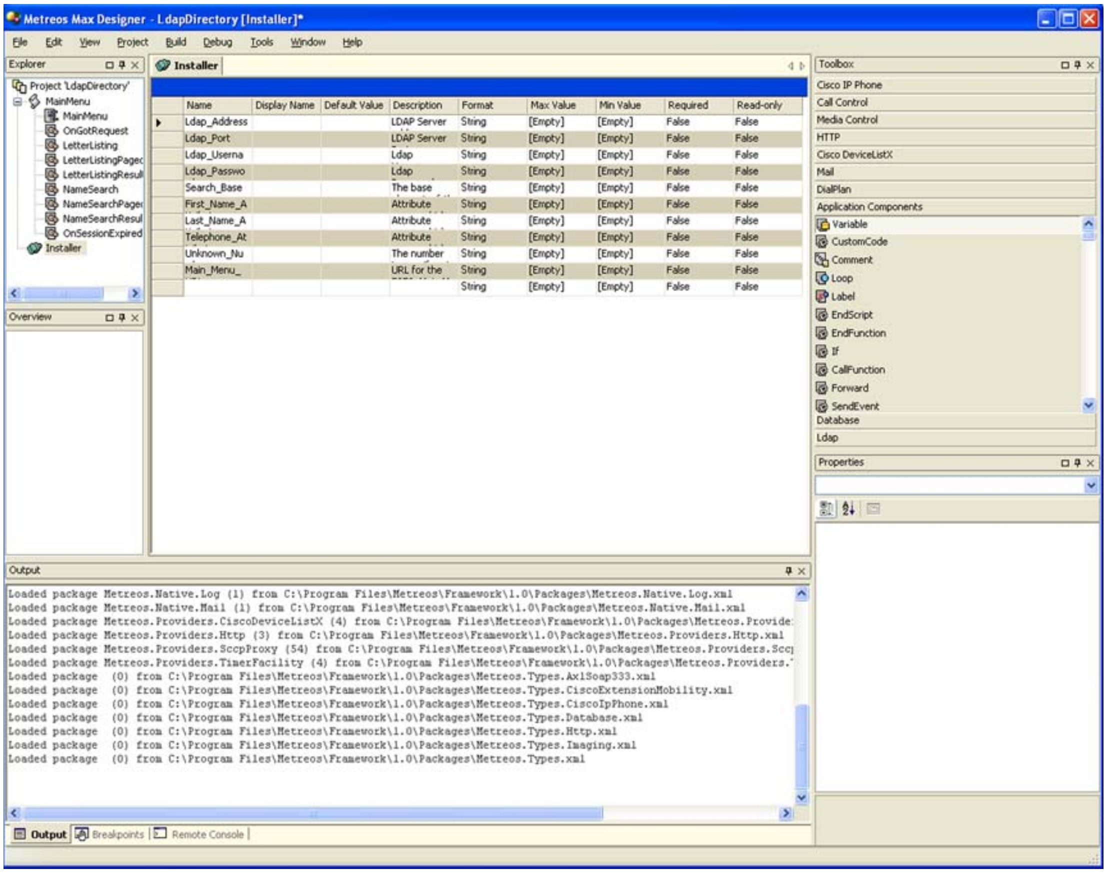

Figure 28: Script Variable

Installer Manager

The Visual Designer provides an *Installer manager* to assist you with install-time configuration data. The Installer manager allows you to specify configuration data so that administrators can configure the application as needed during deployment. For example, if an application requires a database hostname, the hostname may be known only by the administrator at install time.

The Metreos SDK contains several sample applications and some contain configuration variables, such as the LDAP application show in Figure 29.

Figure 29: MCE LDAP Application Installer Contents

To add an Installer item, perform the following procedure:

1. Double click on the **Installer** icon in the Explorer window to view the Installer manager canvas.

2. Right click anywhere on the Installer canvas and select **Add**. The Visual Designer adds an empty row at the bottom of the Installer list as shown in Figure 30.

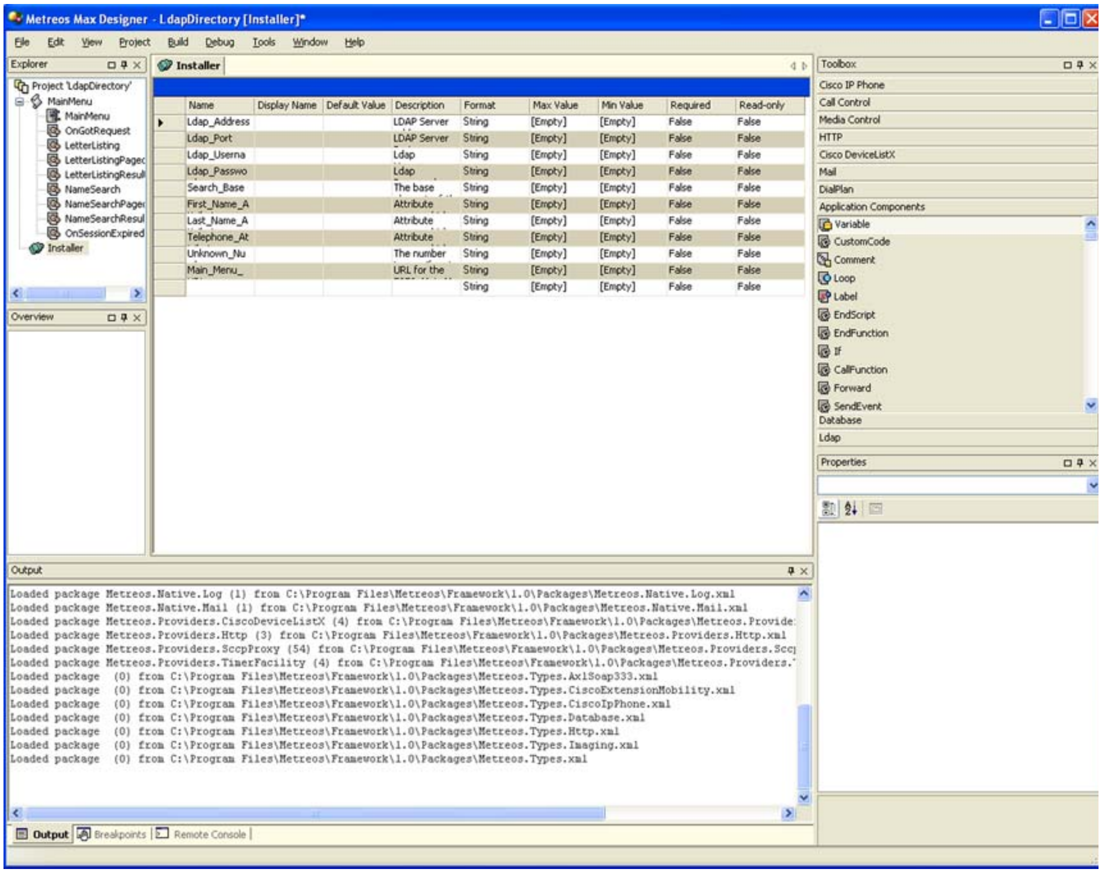

Figure 30: Adding an Installer Manager Item

3. Populate the row with the following information:

- Name — The variable containing the configurable data
- Display Name — The name displayed in the Management Console you want associated with the configurable data (optional)
- Default Value — The variable default value (optional)
- Format — The variable data type (select from dropdown list)
- Max Value — The maximum variable value (only valid for **Number** and **DateTime** format fields)
- Min Value — The minimum variable value (only valid for **Number** and **DateTime** format fields)
- Required — Set to **True** if the item is required. Set to **False** if the item is not required.
- Read-only — Set to **True** for read only (expose the data without being able to modify the data). Set to **False** to allow modification of the data.

The first bullet in the previous list instructs you to include the name of the variable containing the configurable data. Although this is a required field, the Visual Designer does not require you to define the variable in the **Name** field. The variable is required only to expose or modify the configuration data for the current Installer manager item to the administrator. Refer to the "Variables" on page 41 for details about defining variables.

You can also delete an item from the Installer manager by right clicking the Installer manager canvas and selecting the item you want to delete.

Sample Applications 5

The Metreos SDK contains several sample applications designed to assist you in learning to use the the Visual Designer. These applications are located in *install_directory***sdk\examples\applications** and include:

- AmazonWebServices — An application that displays a menu on an IP phone for accessing Amazon Web services, such as searching for a book title over the Internet
- AxlSoap Applications — Several applications that exercise AxlSoap actions, such as changing call forwarding information
- ExtensionMobility — An application demonstrating how to use the Extension Mobility API that CallManager exposes
- ForcedAuthCodes — An application showing the use of audio to prompt users and to authenticate users through the phone keypad
- LdapDirectory — An application that displays a menu on an IP phone for accessing an LDAP directory
- ScheduledConference — An application for managing a teleconference
- Weather — An application that displays a menu on an IP phone for accessing weather information over the Internet

All of these applications are fully functional and can be deployed to the Metreos Application Runtime Environment. This chapter presents the Weather application describing how it was built and how it works. This chapter also describes how to develop, build and deploy a simple application *Announcement*.

Weather Application

The simplest application in the SDK is Weather. The main menu of the Weather application is depicted in Figure 31.

Figure 31: IP Phone with Main Weather Menu

Pressing **Select** at the **Services** → **Weather** prompt displays a menu with a single item called **Find a weather station**. Pressing the **Select** button at the **Find a Weather station prompt** displays another menu from which you can select a U.S. state. Selecting a state displays a menu containing all the weather stations in the selected state that could be located over the Internet.

Selecting a weather station displays the name of the weather station and weather data local to the selected station as shown in Figure 32.

Figure 32: IP Phone Displaying Data from Local Weather Station

The Weather application was written using the Metreos Visual Designer. View the application by double clicking the **Weather MAX** file located at *install_directory\s**dk\examples\applications\Weather\mca*** as shown in Figure 33.

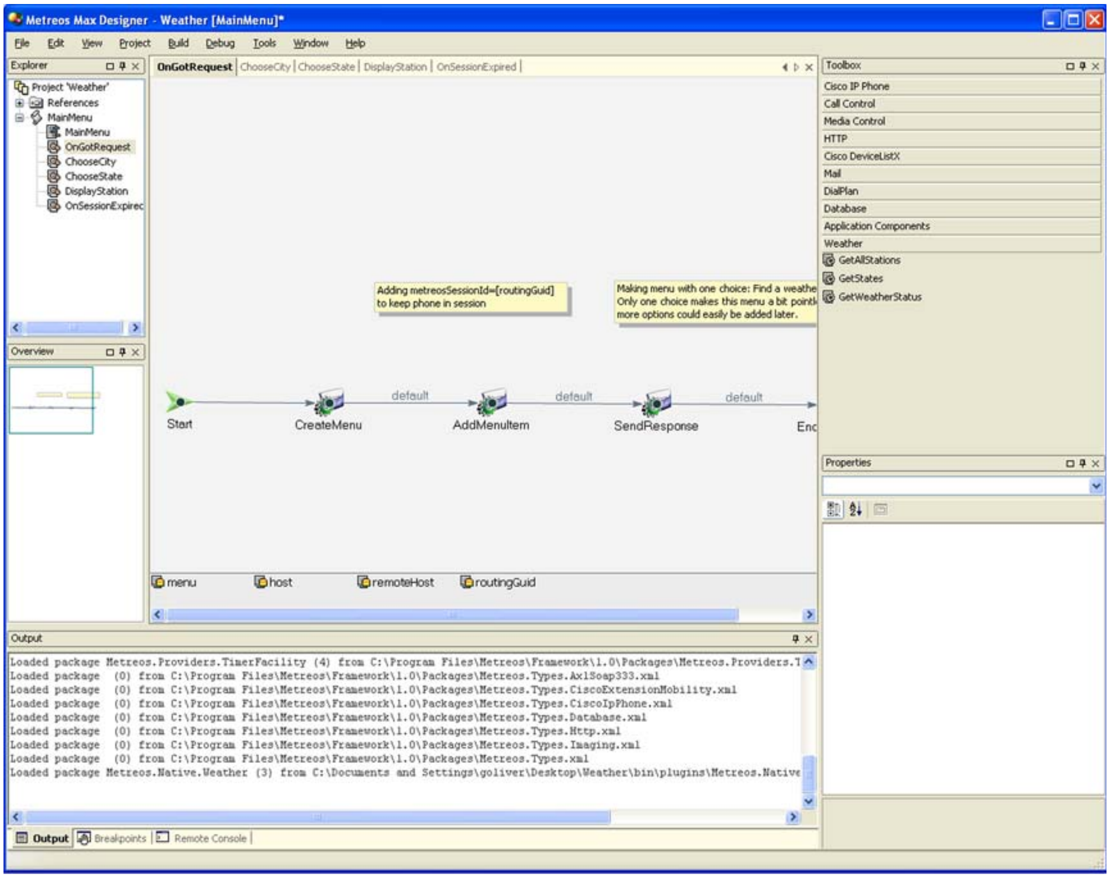

Figure 33: Example Weather Application from SDK

The Explorer Window in the upper left displays the application tree beginning with the Weather Project. The Weather Project contains a single script called **MainMenu**, which contains the following five functions:

- **OnGetRequest**
- **ChooseCity**
- **ChooseState**
- **DisplayStation**
- **OnSessionExpired**

The first function in the tree, **OnGotRequest**, was added automatically when the **MainMenu** script was added. As previously stated, when a script is added, a triggering event must be associated with it as shown in Figure 34.

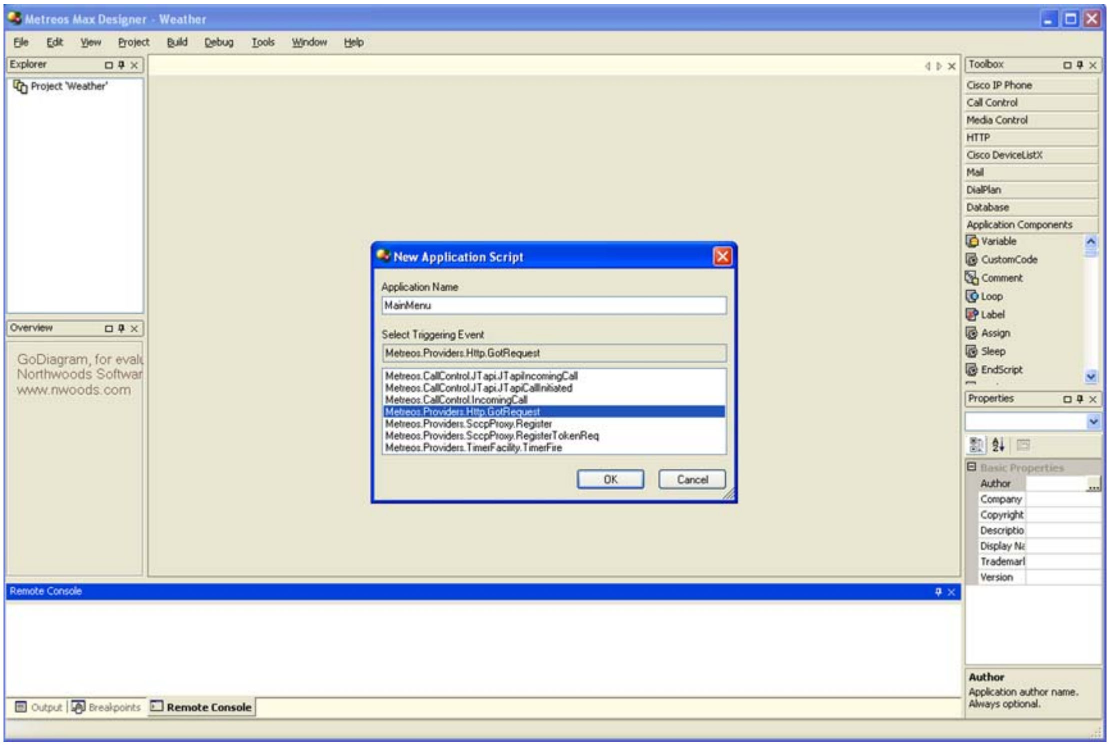

Figure 34: Selecting a Triggering Event When Adding a Script

In this example, the triggering event is an HTTP request that is identified in the MCE as **Metreos.Providers.Http.GotRequest**. This is the appropriate trigger because the phone makes an HTTP request when the user requests the Weather service. The HTTP request made by the phone triggers the application script.

Clicking the **OK** button on the **New Application Script** dialog creates the script and the **OnGotRequest** function. Selecting the **OnGotRequest** tab displays a clean canvas on which the application can be built. See Figure 35.

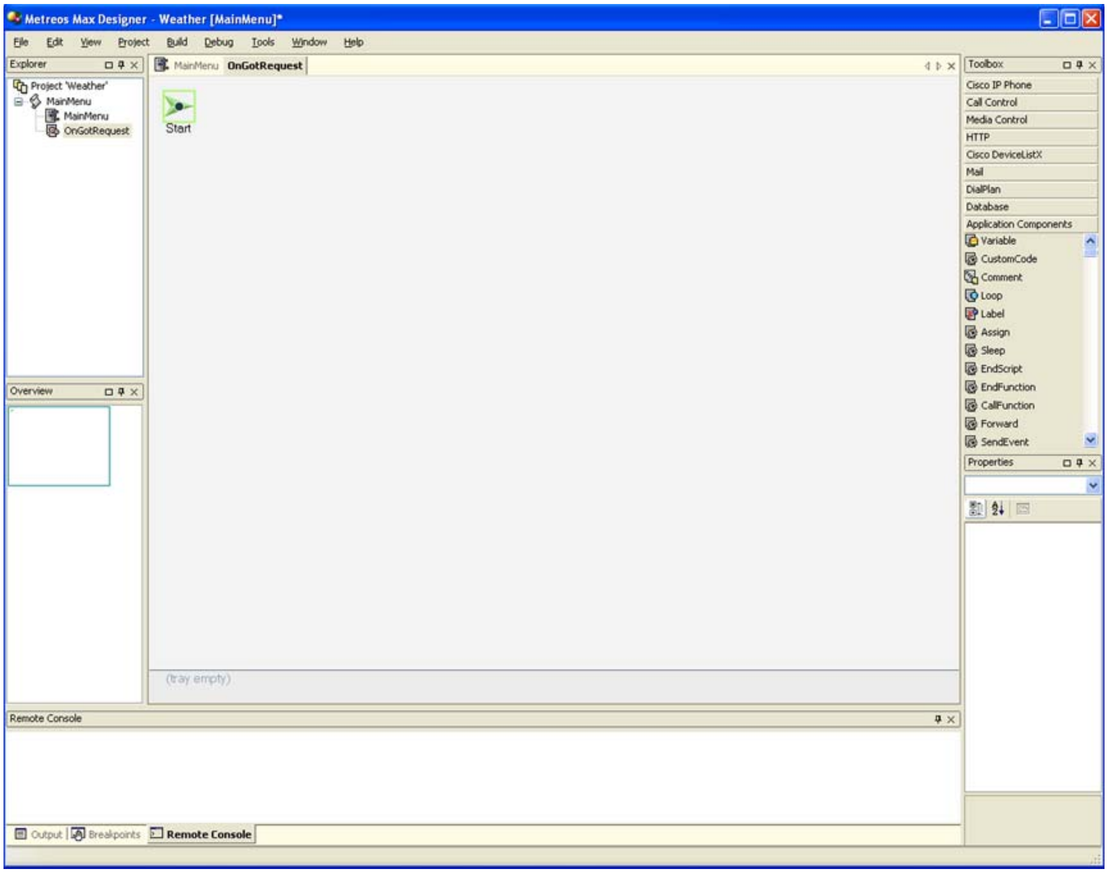

Figure 35: Clean OnGotRequest Weather Function

Nodes and Functions

After the script is created, components can be dragged onto the canvas from the toolbox. The first node, **CreateMenu**, is located in the Cisco IP Phone tab in the toolbox as shown in Figure 36.

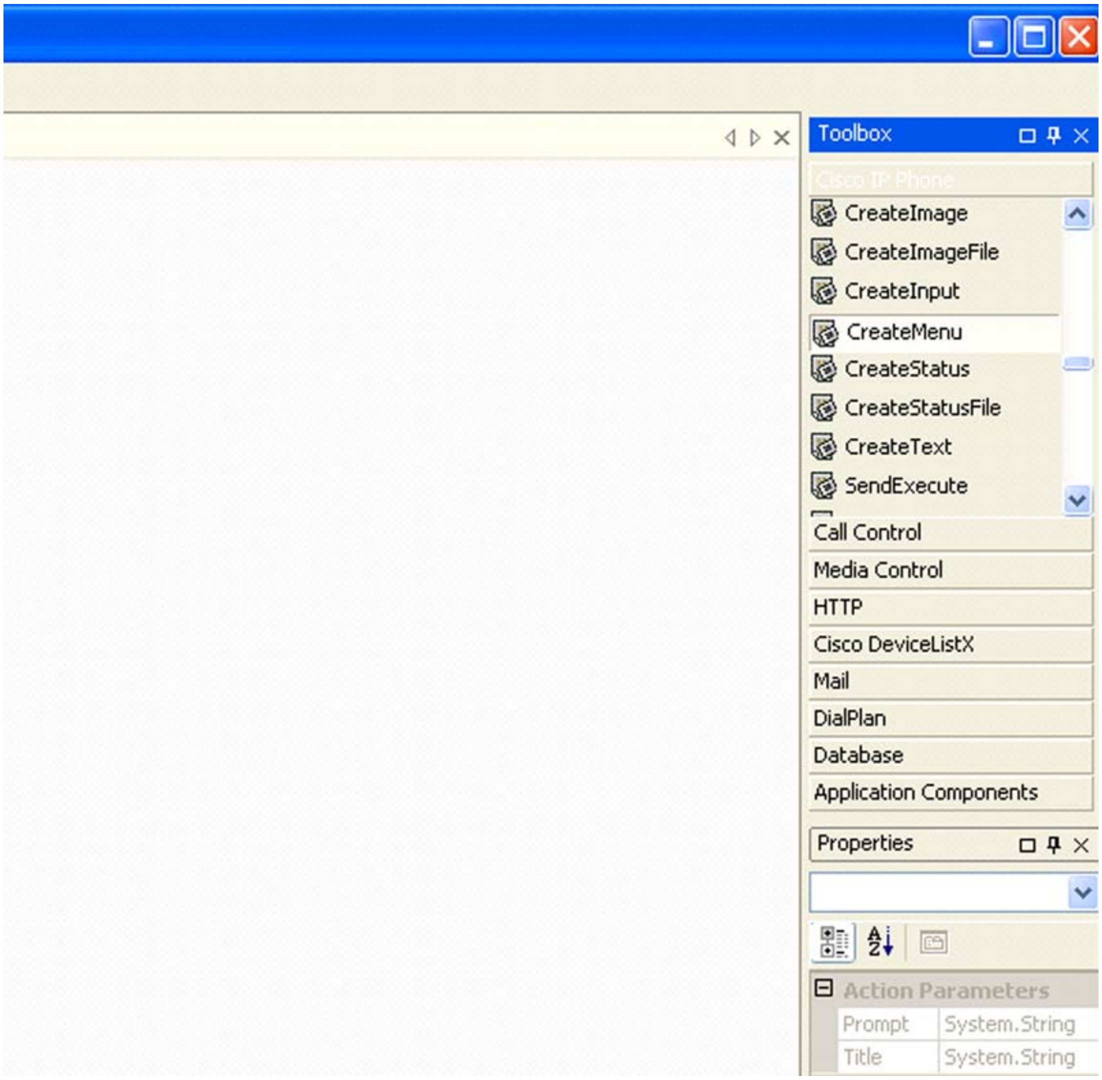

Figure 36: Selecting CreateMenu

Table 1 presents all the nodes used in this application, the Toolbox tab in which each node can be found and the purpose of each node

Table 1. Weather Application Nodes

Node	Category Head	Purpose
CreateMenu	Cisco IP Phone	Creates a menu to display on the IP Phone.
AddMenuItem	Cisco IP Phone	Adds one menu item to the previously created menu.

Table 1. Weather Application Nodes (Continued)

Node	Category Head	Purpose
SendResponse	HTTP	Converts data to a string for displaying in an HTTP browser. The string is then transmitted to the specified remote host. In this case, a Cisco IP Phone menu object is converted and sent to an IP phone for display.
EndFunction	Application Components	Ends the function.
	ChooseState	
CreateInput	Cisco Phone IP	Creates a Cisco IP Phone input object.
AddInputItem	Cisco IP Phone	Adds an input field to the previously created input object. Takes input from phone keypad-in this case the two character abbreviation for the state.
	Choose City	
GetAllStations	Native Action — Weather Tab	Queries all weather stations providing an Internet feed in the selected state.
Loop (AddMenuItem)	Application Components	Adds a menu item for each station found during the loop
CreateText	Cisco IP Phone	Creates text to print on the phone display-in this case prints a standard error message indicating the state could not be found.
AddSoftKeyItem	Cisco IP Phone	Presents an option on the phone display the user can select—in this case two softkey items are displayed in positions 1 and 3: **Menu** and **Exit** respectively.
	DisplayStation	
GetWeatherStatus	Native Action - Weather Tab	Returns a single string containing all the weather data from the selected station.
	OnSessionExpired	

Table 1. Weather Application Nodes (Continued)

Node	Category Head	Purpose
EndScript	Application Components	When the HTTP session expires the script will end and the script instance will be terminated.

The last item in the table, **EndScript**, occurs only in the **OnSessionExpired** function. This function was created by dragging it from the HTTP Toolbox group to the **MainMenu** script tree on the canvas as shown in Figure 37.

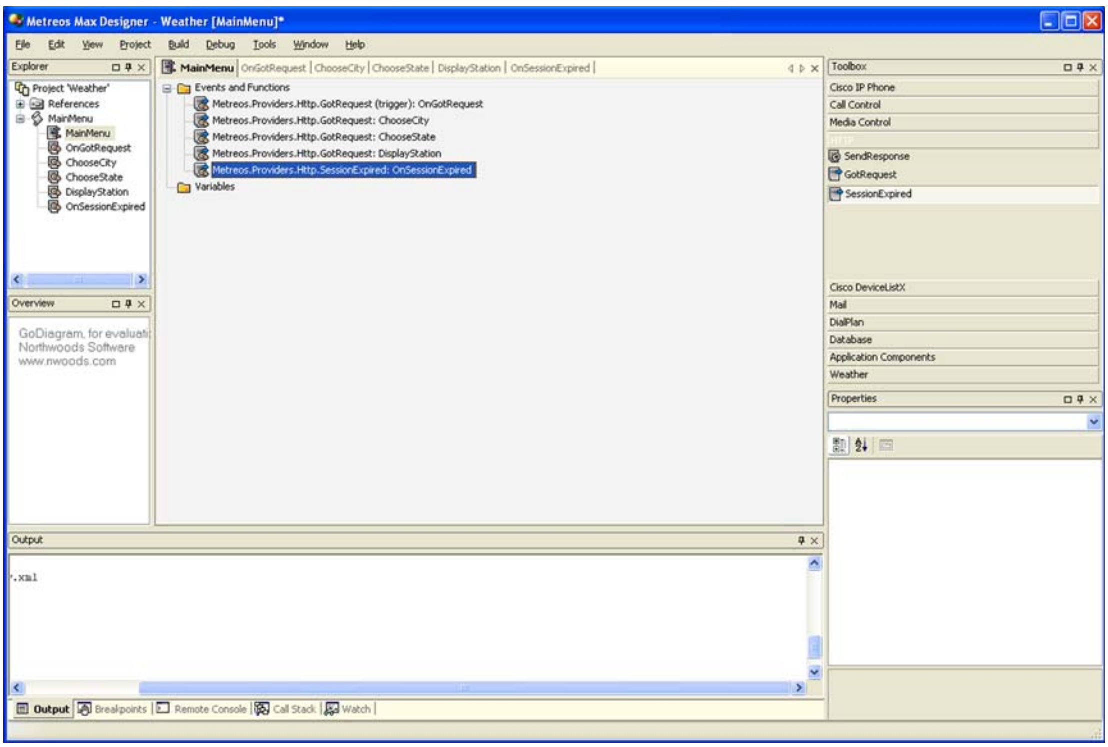

Figure 37: Dragging A SessionExpired function onto the script tree

The **OnGotRequest** function is also available from the **HTTP** toolbox tab. You can drag an **OnGotRequest** function onto the script tree to add it to your application as needed.

All MCE application components, functions and actions are documented in the Appendix A “API Reference” on page 89.

Using Loops

In the Weather sample application, the **ChooseCity** function uses a loop to build the weather stations list on the IP phone after you choose a state as shown in “AddMenuItem Loop” on page 56.

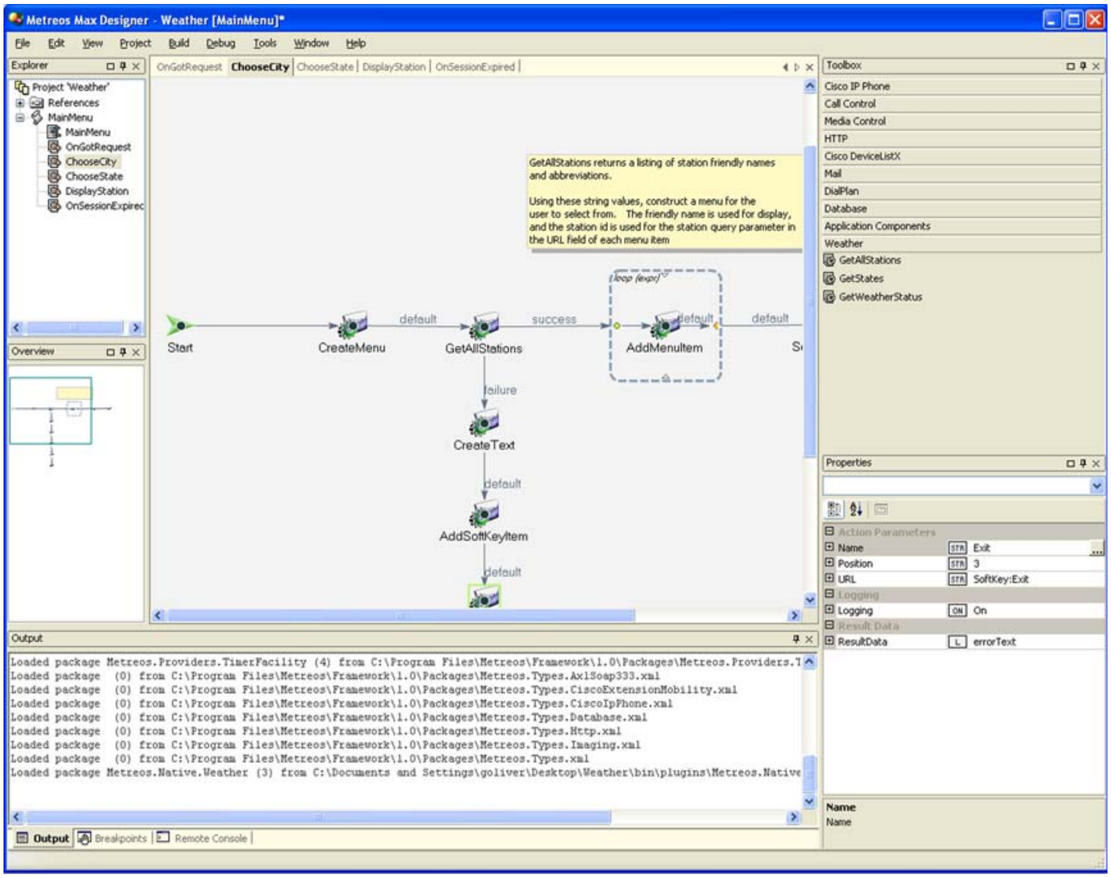

Figure 38: AddMenuItem Loop

You can drag a **Loop** element from the **Application Component** section of the toolbox and drop it on any action or collection of actions. You can also drop elements inside the **Loop** element.

A loop in the Metreos Visual Designer is conceptually the same as a loop in most high-level programming languages. In the Visual Designer, you can specify the number of times to loop a set of actions in any of the following ways:

- Using an integer count — Increment the loop count each time an action is performed.
- IEnumerator collections — Perform the action for each item in a collection.
- IDictionaryEnumerator collections — Perform the action for each key/value pair in a collection.

Review the Loop properties in the Property Grid. The **Loop Iteration Type** is set to **int**. When using the integer count method, the specified Count value must be an integer. The loop executes the action the number of times specified by the integer. Within the loop, you can use a variable, **loopIndex**, to access the current execution count. To do so, use the variable name **loopIndex** in a C# statement within **Action Parameters** for any action in the loop. The **loopIndex** variable is of type **System.Int32**.

In the example of the Weather application, the count was clipped to a maximum of 100 because the IP phone for which it was written cannot display more than 100 items. The **ChooseCity** function gets the first item from the list contained in the Cities variable, increments the Native MCE count and checks to see if the count has reached 100.

If the count is less than 100, then it gets the next item in the list. If the count has reached 100, or if the collection of stations contains no more items, the loop stops executing. It then sends an HTTP response to the phone display, and the function ends. If there are additional stations in the **Cities** variable beyond the first 100, they remain unused.

Exercise caution when using loops. The Weather application contains logic in the loop count to determine whether there are fewer than 100 stations before it begins executing. Without the logic to detect whether the list contains fewer than 100 stations, the loop would fail and script execution would end. The following C# code provides the necessary logic to determine whether the loop will execute exactly 100 times or until it reaches the end of the list of stations:

```
stations.Count > 100 ? 100 : stations.Count
```

The loop uses a previously defined variable, **stations.Count,** to track the loop iteration. Before the loop executes, the Application Runtime Environment checks the value of **stations.Count**. If **stations.Count** has a value greater than 100, the loop executes 100 times. If stations.Count does not have a value greater than 100, it executes the number of times equal to the value of **stations.Count**.

The other two techniques for loop execution are closely tied to .NET. They are equivalent to using a **System.Collections.IEnumerator** or **System.Collections.IDictionaryEnumerator** to control the loop iteration. Refer to the following Web sites for information on **System.Collections.IEnumerator** and **System.Collections.IDictionaryEnumerator**:

IEnumerator:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsienumeratormemberstopic.asp

IDictionaryEnumerator:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemcollectionsidictionaryenumeratormemberstopic.asp

To use an **IEnumerator** collection, you must specify an object implementing **System.Collections.IEnumerable**. The object implementing **System.Collections.IEnumerable** must be specified in the Count property of the loop. Alternatively, you can specify an object resolving to some other object implementing **System.Collections.IEnumerable**.

The **Loop Iteration Type** should be specified as **enum**. You then specify the **loopEnum** keyword in the action parameters of the C# code for the loop. The **loopEnum** keyword is the **IEnumerator** returned by the **GetEnumerator()** method of the **IEnumerator** collection. To access the object to which the loopEnum iterator refers, use the **loopEnum.Current** property and cast it to the desired type.

Using an **IDictionaryEnumerator** to control a loop is very similar to using an **IEnumerable** object. The primary difference is that the **IDictionaryEnumerator** object available for use within the loop is termed **loopDictEnum**. In this case, you can access the current key as **loopDictEnum.Key**; the current iteration value is **loopDictEnum.Value**.

WARNING: A loop iteration value of zero, or an IEnumerator or IDictionaryEnumerator collection with zero items is invalid. Scripts containing such loops are terminated when the loop is invoked. Use a Conditional node, such as the If action, from the Applications Components tab in the Toolbox to check the loop value.

Script Execution

The following outline presents the event/action sequence when the Weather application is triggered.

1. **OnGotRequest**: (function processing state)
   a. Creates menu object: **Find a Weather Station**
   b. Sends response: menu object and script instance GUID
   c. Ends

2. The script rests (*awaiting events* state)

3. Phone displays menu

4. User selects **Find a Weather Station**

5. **ChooseState** (*function processing* state)
   a. Creates prompt object: **Enter State Abbreviation**
   b. Sends response: input object and script instance GUID
   c. Ends

6. The script rests (*awaiting events* state)

7. Phone displays prompt

8. User enters state abbreviation

9. **ChooseCity** (*function processing* state)
   a. Prompt: Choose Station
   b. URL queries all stations in state
   c. Builds weather station menu object (first 100 or fewer)
   d. Sends response: station menu object and script instance GUID
   e. Ends

10. The script rests (*awaiting events* state)

11. Phone displays station menu with prompt

12. User selects station

13. **DisplayStation** (*function processing* state)

    a. Native Action queries selected station
    b. Builds weather display
    c. Builds **Main Menu** option (**SoftKey**, **Position 1**)
    d. Builds **Exit** option (**SoftKey**, **Position 3**)
    e. Sends Response:
        – Weather display
        – Main Menu Option
        – Exit Option
        – Script instance GUID
    f. Ends

14. The script rests (*awaiting events* state)

15. Phone displays:

    a. Weather
    b. Main Menu Option
    c. Exit Option

16. Session expires

17. **OnSessionExpired**

18. Script ends (*uninstantiated* state)

You can verify the event sequence by selecting each node and reviewing its properties on the Property Grid. For example, the first prompt (**Find a weather station**) is a specified property of **AddMenuItem** in **OnGotRequest** as shown in Figure 39.

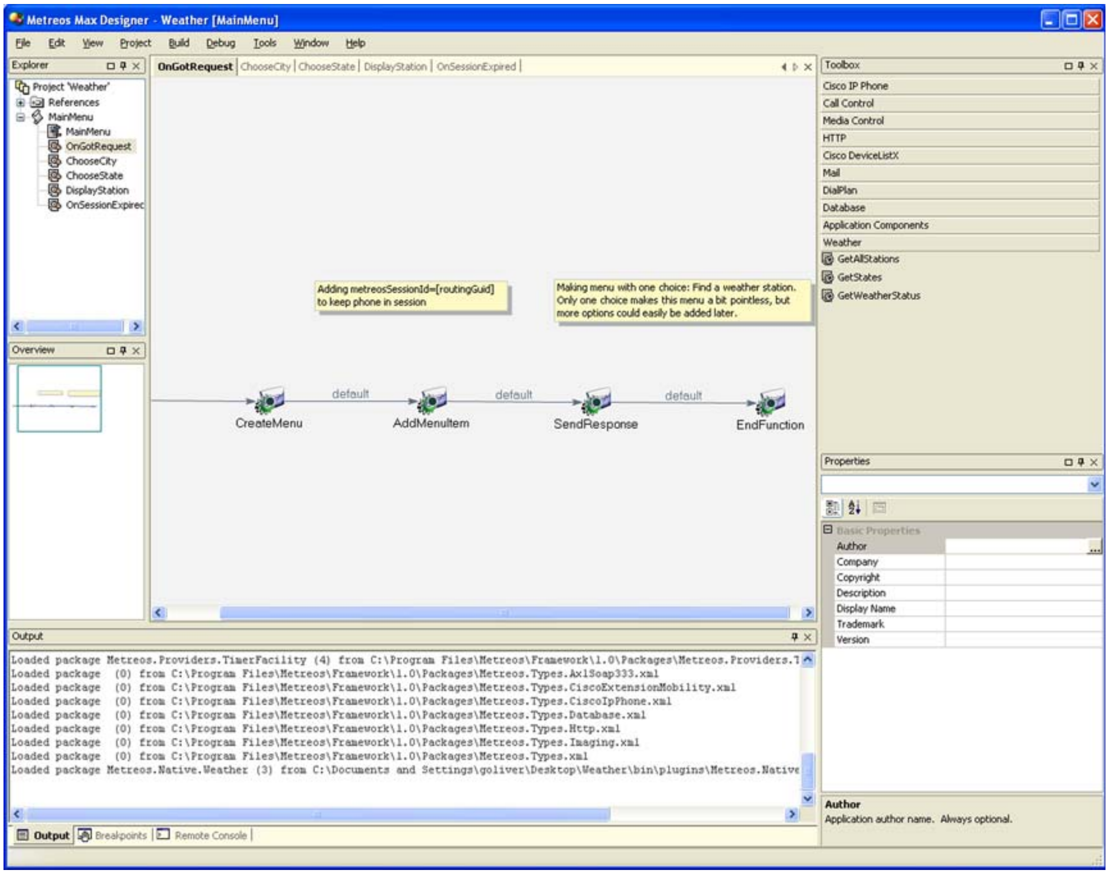

Figure 39: Initial Application Prompt

The Menu object contains a single menu item defined by the **AddMenuItem** action. The object is sent as part of the response from the Application Runtime Environment. The Application Runtime Environment then displays the menu item on the phone shown in Figure 40.

Figure 40: IP Phone with Initial Weather Prompt

Announcement Application

The Weather application is a simple example of how to use the Metreos Visual Designer to develop an application. However, the Weather Application is not typical of most IP telephony applications because the Weather Application requires no media resources. This section presents a very simple application that does require media resources.

The application *Announcement* answers an incoming call and plays an announcement to the user. If the user presses the pound key (#) on the phone keypad during the announcement, the announcement stops playing and actions occur ending both the call and the script. At the end of the announcement the user is asked to respond by pressing # after it has finished playing. The application then waits until the user presses #, at which time actions occur ending the script. If the user does not press # within 10 seconds, the application terminates.

Getting Started

Use the following procedure to begin:

1. Select **File** → **New Project** on the Metreos Visual Designer. The Visual Designer presents a dialog requesting the project name and the path where the project files are stored as shown in Figure 41.

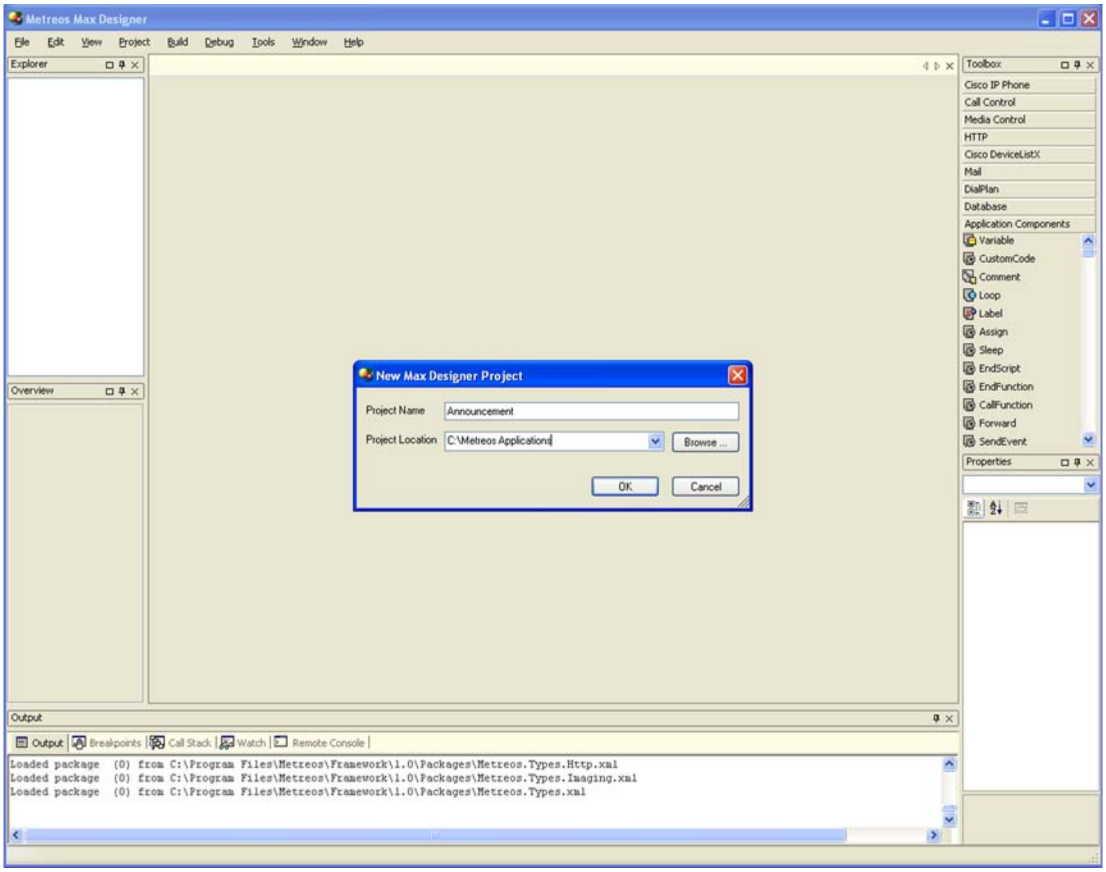

Figure 41: Starting a New Project

2. Type the project name **Announcement** in the **Project Name** field and use the default project location or enter a new location in the **Project Location** field. The Visual Designer presents a new project named Announcement with no canvas as shown in Figure 42.

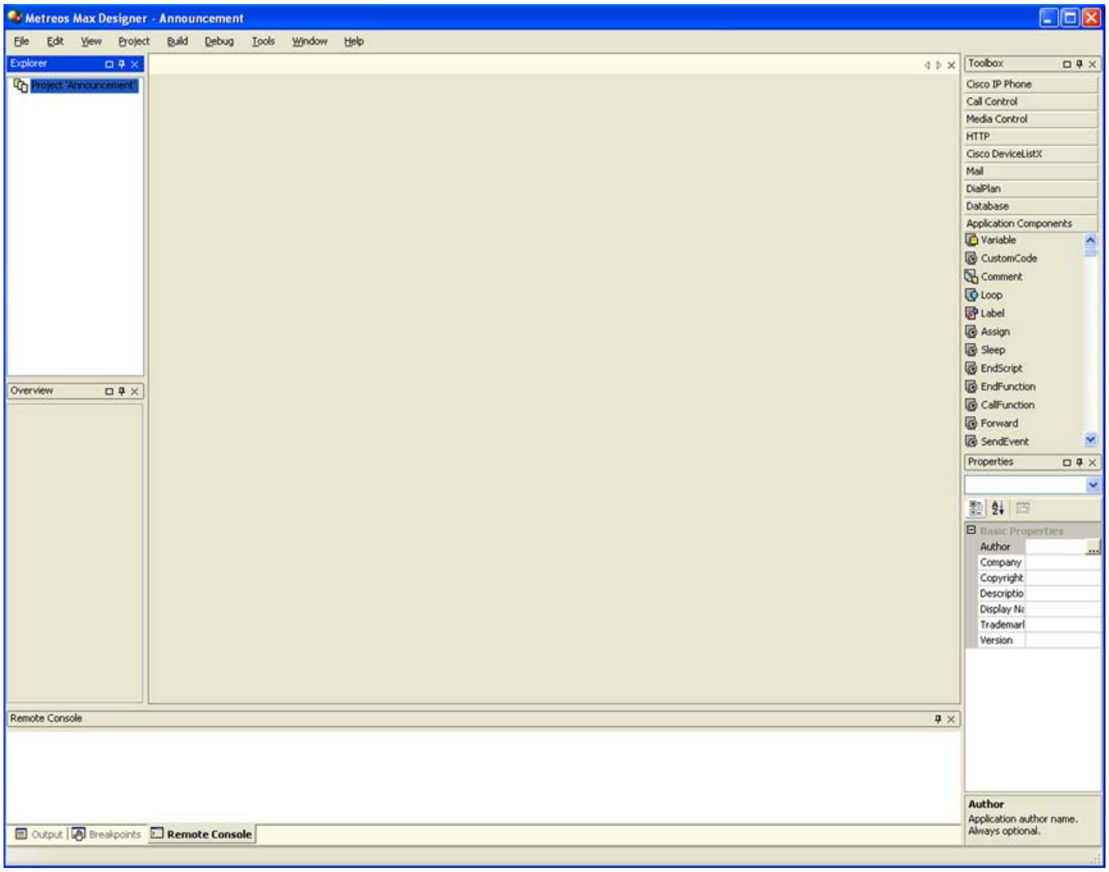

Figure 42: New Project

You can now add a script to the project. When you add a script, you must select a triggering event. Because the sample application begins by answering an incoming call, the triggering event for the sample is **Metreos.CallControl.OnIncomingCall**.

3. Select **File** → **Add Script** → **New Script**. The Visual Designer presents a dialog requesting the name of the script and a triggering event as shown in Figure 43.

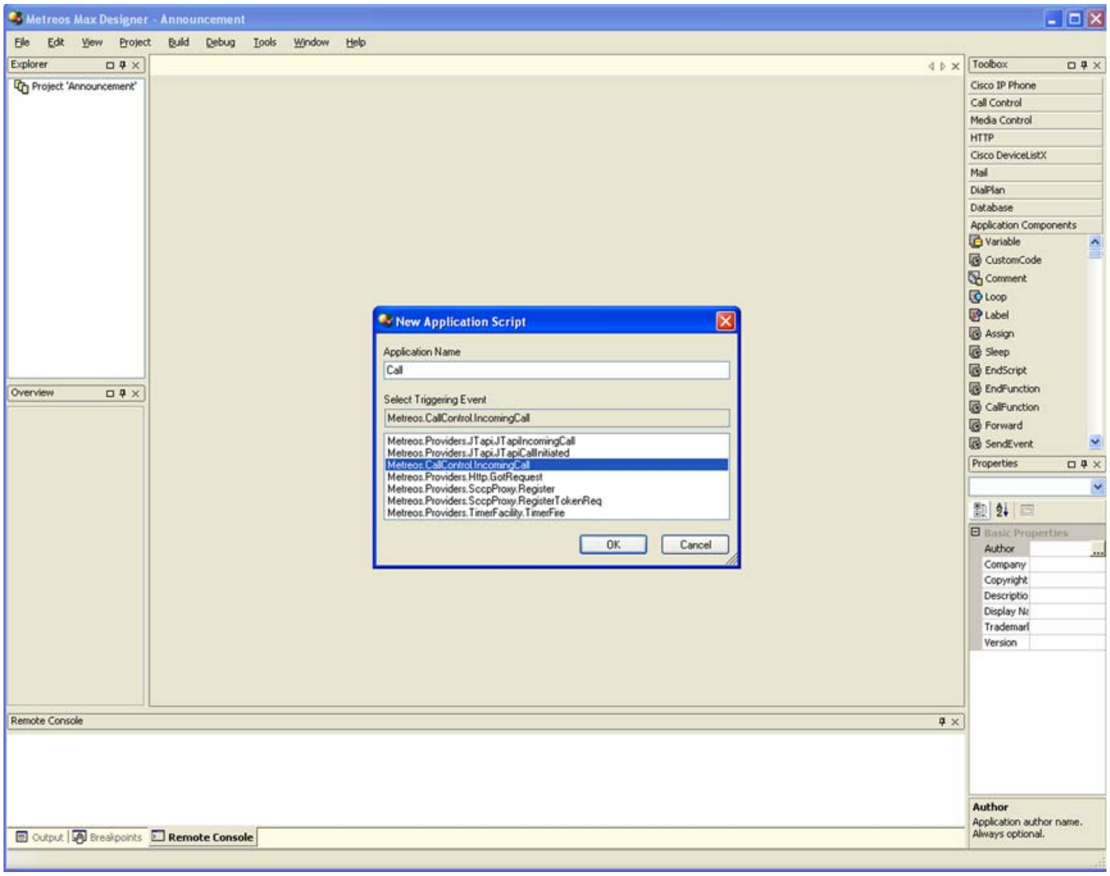

Figure 43: Adding a New Script

4. Enter **Call** in the **Application Name** field and select **Metreos.CallControl.IncomingCall** from the Select Triggering Event list. The Visual Designer will add a script Call in the Exporer Window and present a blank canvas as shown in Figure 44.

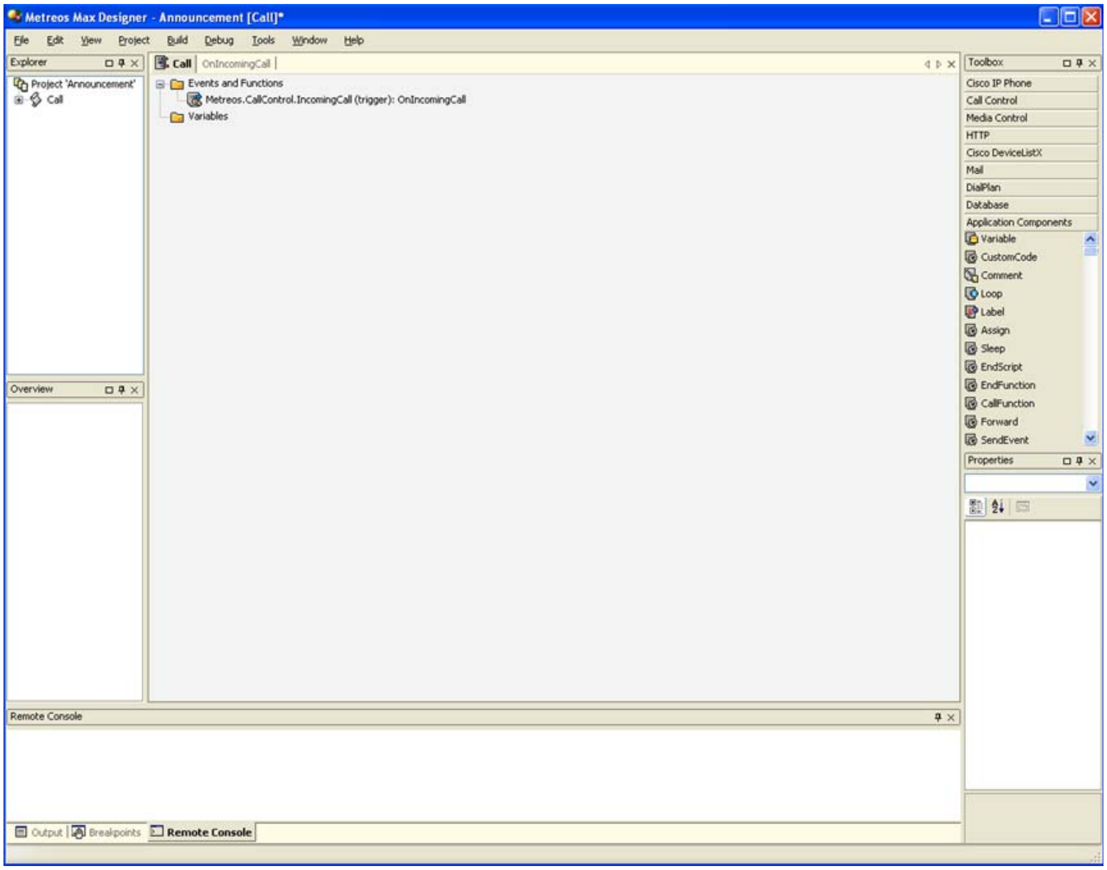

Figure 44: New Script

You can now begin populating the application. Construct the **OnIncomingCall** function as shown in Figure 45.

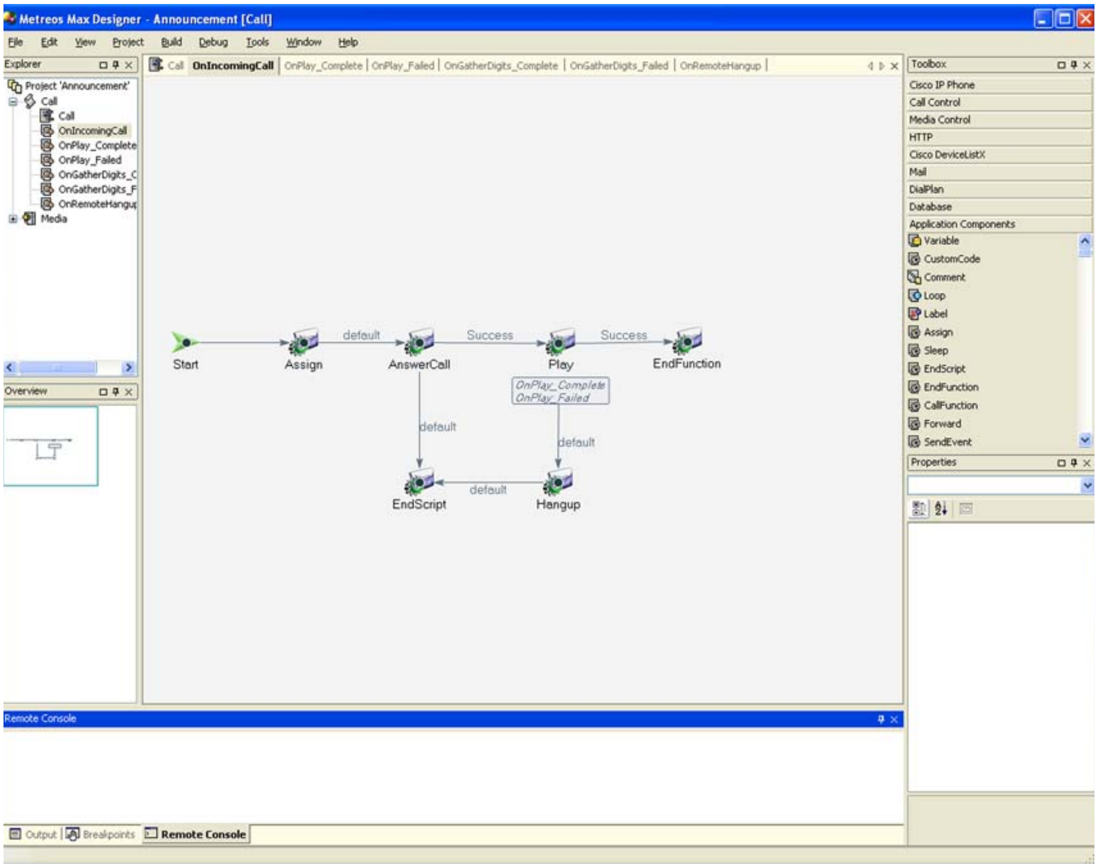

Figure 45: Announcement Application — OnIncomingCall Function

Table 2 presents all the nodes used in the **Announcement OnIncomingCall** function and the catagory in which each function can be found.

Table 2. Announcement OnIncommingCall Function

OnGotRequest		
Node	**Catagory Heading**	**Purpose**
Assign	Application Components	Assigns a specified value to a variable or assigns the value of a variable to another variable
AnswerCall	Call Control	Answers incoming calls
Play	Media Control	Plays an announcement
EndFunction	Application Components	Ends the function
Hangup	Call Control	Terminates a call
EndScript	Application Components	Ends the Script

Assigning Flow Control Labels

Take note of the logic branch at **AnswerCall**. The flow control arrow connecting **AnswerCall** and **Play** is labled **Success**, and the arrow connecting **AnswerCall** and **EndScript** is labeled **default**. If the call is succesfully answered, the script uses **Success** and proceeds to **Play**. If the call is not succesfully answered, the script uses **default** and terminates. Instead of **default**, the developer could have labeled the arrow **Failure**, yielding the same result. **Default** is used because good programming practices suggest that you always provide a default path regardless of the number of possible results.

Variables

The **OnIncomingCall** function requires a variable to allow the Application Runtime Environment to identify and track the call. For example, to terminate the call, the Application Runtime Environment must have a way to correlate the **Hangup** action in the script and the incoming call. The Application Runtime Environment therefore assigns an identity value **CallId** to every call to perform the necessary tracking.

The required variable in the sample application is assigned the MCE **CallId** value. The Application Runtime Environment can then associate the incoming call with any **Call Control** action. Use the following procedure to define the **incomingCallID** variable:

1. Drag a **Variable** node from the Application Components tab in the Toolbox to the lower portion of the canvas. The Visual Designer automatically displays the Variables Tray. Alternatively, you may select *Variables* Tray from the *View* menu.

2. Drop the **Variable** node into the Variables Tray.

3. Assign the following properties to the variable in the Property Grid:

- Initialize: **CallId**
- Name: **incomingCallID**

All nodes in the **OnIncomingCall** function now have access to the value of **CallId** through **incomingCallID**. Many of the actions in other functions of the script also require access to the **CallId**. However, recall that function-level variables cannot be accessed by other functions. It is therefore necessary to define a script-level variable containing the incoming **CallId** value. Define a script-level CallID variable using the following procedure:

1. Click on the **Call** tab above the canvas to view the **Call** script tree.

2. Drag a **Variable** node from the Toolbox to the script canvas. The Visual Designer expands the script tree and adds the variable to the variable tree.

3. Assign a name of **g_incomingCallID** to the variable in the Property Grid.

You can now provide the **CallId** value stored in the **incomingCallID** variable to **g_incomingCallID**, and each function will have access to the incoming CallId. Perform the following procedure to assign the **incomingCallID** value to **g_incomingCallID**:

1. Click the **OnIncomingCall** tab above the canvas to return to the **OnIncomingCall** function.

2. Right click on the **Assignment** node and select **Properties**.

3. Assign the following properties in the Property Grid:

- **Value**: select **incomingCallID** from the dropdown list.
- **ResultData**: select **g_incomingCallID** from the dropdown list.

You should now assign the **g_incomingCallID** value to the **AnswerCall** node using the Property Grid:

1. Right click the **AnswerCall** node and select **Properties**.

2. Select **g_incomingCallID** from the dropdown list next to **CallId** in the Property Grid.

Just as the Application Runtime Environment must track the **CallId** to manage calls, the Metreos Media Engine must track connections to manage media. For example, when the media server begins to stream the audio to play the announcement, the Metreos Media Engine must know the specific connection to which the stream will be sent.

If a media server connection is successfully established after the call is answered, the Metreos Media Engine automatically assigns a connection identifier **ConnectionId** to the connection. The Metreos Media Engine can then track and manage the media. You should assign the **ConnectionId** value to a variable so that connection data is available to all Media Control actions. Perform the following procedure to define a script-level variable containing the **ConnectionId** data:

1. Click the Call tab above the canvas to return to the Call canvas.

2. Drag a variable from the Toolbox to the Call canvas.

3. Assign a name of **g_connectionID** to the variable.

4. Click the **OnIncomingCall** tab above the canvas.

5. Right click **AnswerCall** and select Properties.

6. Select **g_connectionID** from the dropdown list next to **ConnectionID** in the Result Data section of the Property Grid.

Other Action Properties

Complete the **OnIncomingCall** function by assigning the necessary properties to the other nodes in the function using the following list:

- **Play**:
  - **ConnectionID**: **g_connectionID**
  - **Prompt**: *audio_filename*.**wav** — the audio file containing your announcement

NOTE: *The only valid audio formats for MCE media servers are wav and vox.*

  - **TermCondDigit**: # — the digit on the keypad the user can press to terminate the announcement before it finishes playing

- **Hangup**: CallId: **g_incomingCallID**

Additional Media Resources

In addition to specifying a **.wav** or **.vox** audio file, you also have the option of entering the prompt as text. When you build the application script, the prompt text will be converted to audio media using NeoSpeech TTS™. You can also use markup tags to specify audio attributes, such as voice pithc and pauses between words as described in Table 3.

Table 3. Speech Attribute XML

Attribute	Markup	Description
Pause	[**vt_pause**=*pause_time*]	*pause_time* is the pause duration interval in milliseconds. Example: "But wait [vt_pause=500] there's more." Pauses are automatically added for commas, periods, and question marks.
Speed	[**vt_Speed**=*speed*]	*speed* is the specified percentage of the default speed. Valid range is **25** to **400**. Example: "Normal speaking speed, [vt_Speed=150] and faster speaking speed, [/vt_Speed] and normal speaking speed."
Volume	[**vt_Volume**=*volume*]	*volume* is the specified percentage of the default volume. Valid range is **0** to **500**. Example: "Normal volume, [vt_Volume=150] and increased volume, [/vt_Speed] and normal volume."
Pitch	[**vt_Pitch**=*pitch*]	*pitch* is the specified percentage of the default pitch. Valid range is **50** to **200**. Example: "Normal pitch, [vt_Pitch=150] and higher pitch, [/vt_Speed] and normal pitch."

In addition to inserting appropriate pauses, punctuation also appropriately alters speech inflection. For example, adding a question mark automatically increases the pitch of the word immediately preceding the question mark.

You can also specify the pronunciation of individual words by adding an entry to the dictionary. To do so, execute c:\program files\VW\VT\Kate\M08\bin\UserDicEng.exe from the command line.

You also have the option of listening to the speech audio and creating wav audio files of the speech before deploying the application:

1. Execute c:\program files\VW\VT\Kate\M08\bin\vt_eng.exe from the command line to launch NeoSpeech TTS and to view the NeoSpeech TTS Voice Text Editor as shown in Figure 46.

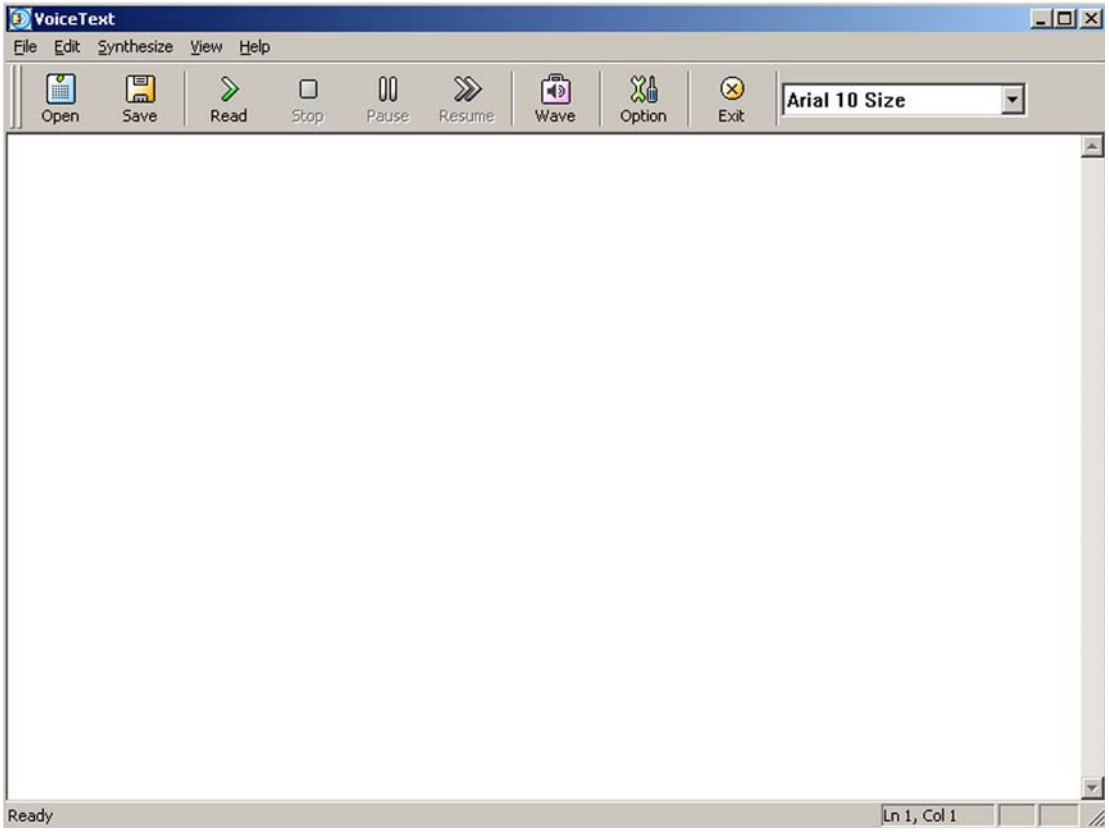

Figure 46: NeoSpeech TTS VoiceText Editor

2. Type the text you want to convert to speech in the editor.

NOTE: *If you apply any markup to modify the speech attributes, replace the square brackets ([and]) in table 1 with angle brackets (< and >). Square brackets are not valid in the NeoSpeech TTS Voice Text editor.*

3. Click the **Read** button to play the speech audio, or click the **Wave** button to create a wav audio file.

Hint: The TextToSpeech feature for all automated audio prompts can be beneficial. It provides a single consistent voice for all the audio prompts in your applications.

Asynchronous Events

When you added the **Play** node onto the canvas, the Visual Designer added **OnPlay_Complete** and **OnPlay_Failed** functions. These functions correspond to the two possible asynchronous events that could occur when the Play action completes:

- **Metreos.MediaControl.Play_Complete** — Occurs if the announcement plays succesfully
- **Metreos.MediaControl.Play_Failed** — Occurs if the announcement fails to play succesfully

You can view the corresponding events and actions at any time by expanding the Events and Functions branch of the **Call** script canvas as shown in Figure 47.

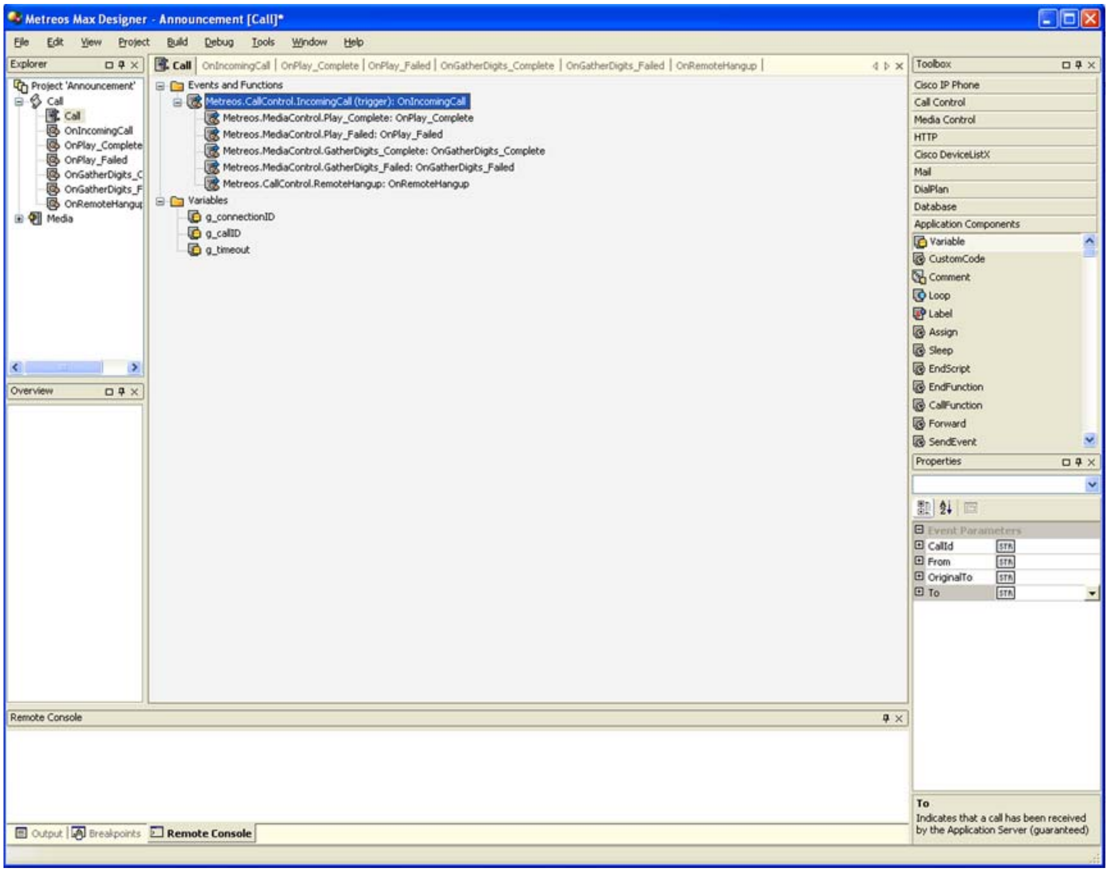

Figure 47: Script Event Handler List

OnPlay_Complete

Figure 48 depicts the **OnPlay_Complete function** canvas.

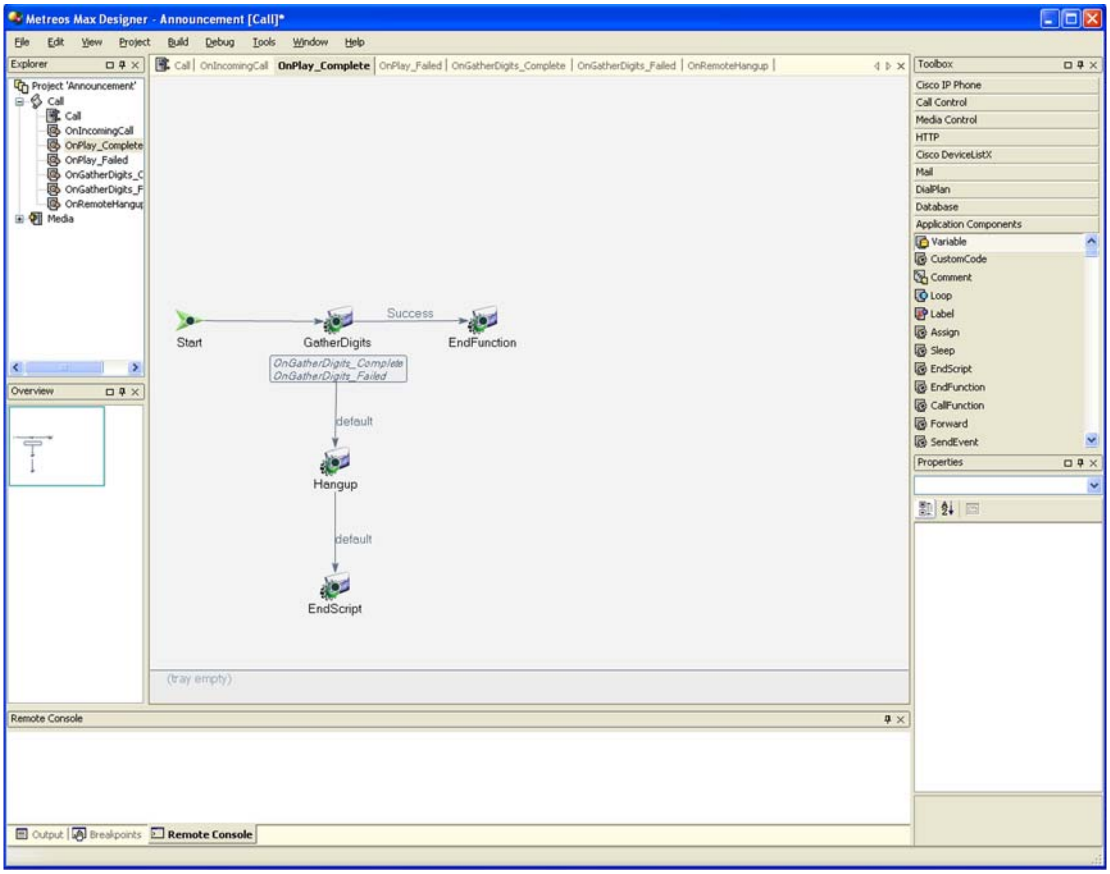

Figure 48: OnPlayComplete Function Canvas

Construct the **OnPlay_Complete** function as shown in Figure 48. Table 4 presents all the nodes used in this application and the catagory in which each can be found.

Table 4. Announce OnPlay_Complete

Node	Catagory Heading	Purpose
	OnPlay_Com	
GatherDigits	Media Control	Records the keys the user pushes
Hangup	Call Control	Terminates the call
Play	Media Control	Plays an announcement
EndFunction	Application Components	Ends the function
EndScript	Call Control	Ends the script

If the announcement begins to play successfully, the following three results are possible:

- The user presses the # key, terminating the announcement, the call and the script.
- The announcement finishes playing and the user presses the # key as directed in the announcement, terminating the call and the script.

- The user never presses the # key, terminating the call and the script after 10 seconds.

Terminating the announcement is handled by the **Play** action in the **OnIncomingCall** function (**TermCondDigit**: #). **GatherDigits** handles the *user response* (user presses the # key after the announcement plays) and the *maximum time* condition.

To specify a maximum time condition another variable is required, the value of which is the interval in milliseconds that the application will wait before terminating the function. Use the following procedure to define the maximum time variable:

1. Click the **Call** tab to navigate to the **Call** script canvas.
2. Drag a variable from the toolbox to the script canvas.
3. Assign the following properties to the variable:
   1. DefaultValue: 10000
   2. **Type**: Select **UInt** from the dropdown box.
   3. Variable Name: g_maxTime
4. Click the **OnPlayComplete** tab to navigate to the **OnPlayComplete** function.
5. Right click on **GatherDigits** and select **Properties**.
6. Select **g_maxTime** from the dropdown box next to **TermCondMaxTime** in the Property Grid.

g_maxTime is assigned a **Type** of **UInt** because the **TermCondMaxTime** property requires a value of the **Type UInt**. You can view the required data **Type** by expanding the **TermCondMaxTime** tree in the property grid.

You can now assign the other properties necessary to complete the action:

- **ConnectionID: g_connectionID**
- **TermCondDigit**: #
- **TermCondDigit** → **Required Type** → **User Type: literal**

Right click on **Hangup** and select **Properties**. Select **g_incomingcallID** from the dropdown box next to CallId in the Property Grid. The **OnPlayComplete** function is now complete.

OnPlay_Failed

The **OnPlay_Failed** function terminates the call and ends the script as shown in Figure 49.

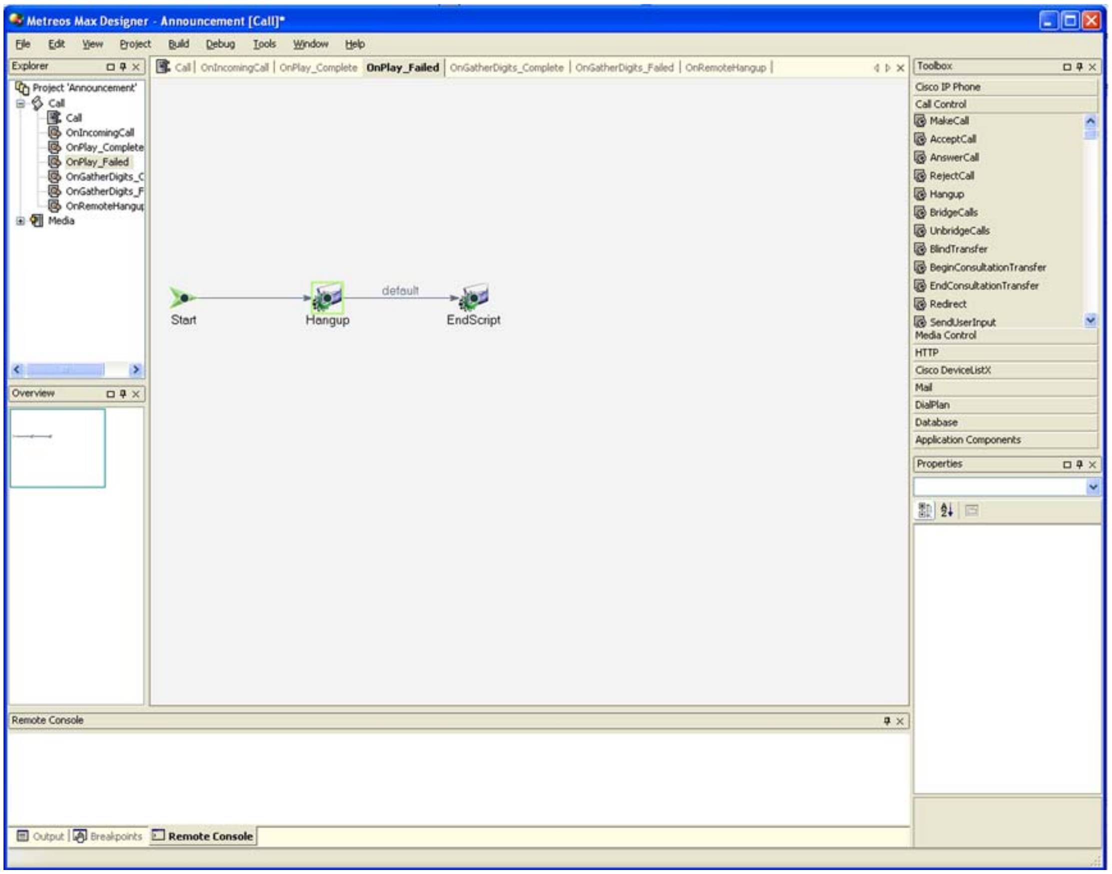

Figure 49: OnPlay_Failed Function Canvas

Right click on **Hangup** and select **Properties**. Select **g_incomingcallID** from the dropdown box next to **CallId** in the Property Grid. The **OnPlay_Failed** function is now complete.

OnGatherDigits_Complete and OnGatherDigits_Failed

When you added the GatherDigits node to the **OnPlayComplete** function, the Visual designer added the following two asynchronous event handlers to the script:

- **OnGatherDigits_Complete** — Invoked if the user presses # after the announcement plays
- **OnGatherDigits_Failed** — Invoked if the media server is unable to succesfully execute the command

In both cases, the system terminates the call and ends the script as depicted in the **OnGatherDigits_Complete** Figure 50.

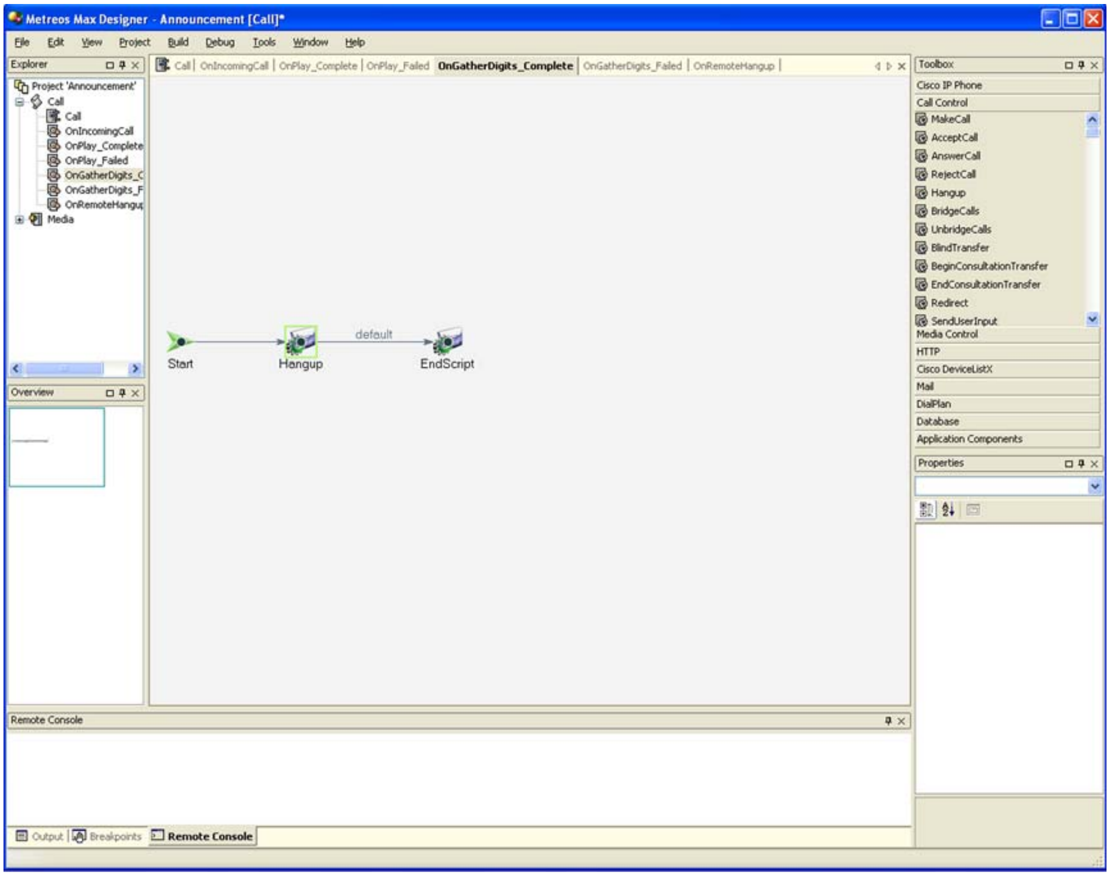

Figure 50: OnGetDigits_Complete Function Canvas

Construct the **OnGetDigits_Complete** function as shown in Figure 50. Right click on **Hangup** and select **Properties**. Select **g_incomingCallID** from the dropdown box next to CallId. The **OnGetDigits_Complete** function is now complete.

Move the mouse pointer on the **OnGetDigits_Complete** canvas to the right of and below the **EndScript** node. Click and hold down the left mouse button and down drag the mouse pointer to the left and above the **Hangup** node. The Visual Designer draws a rectangle around the two components as shown and then highlights them as shown in Figure 51.

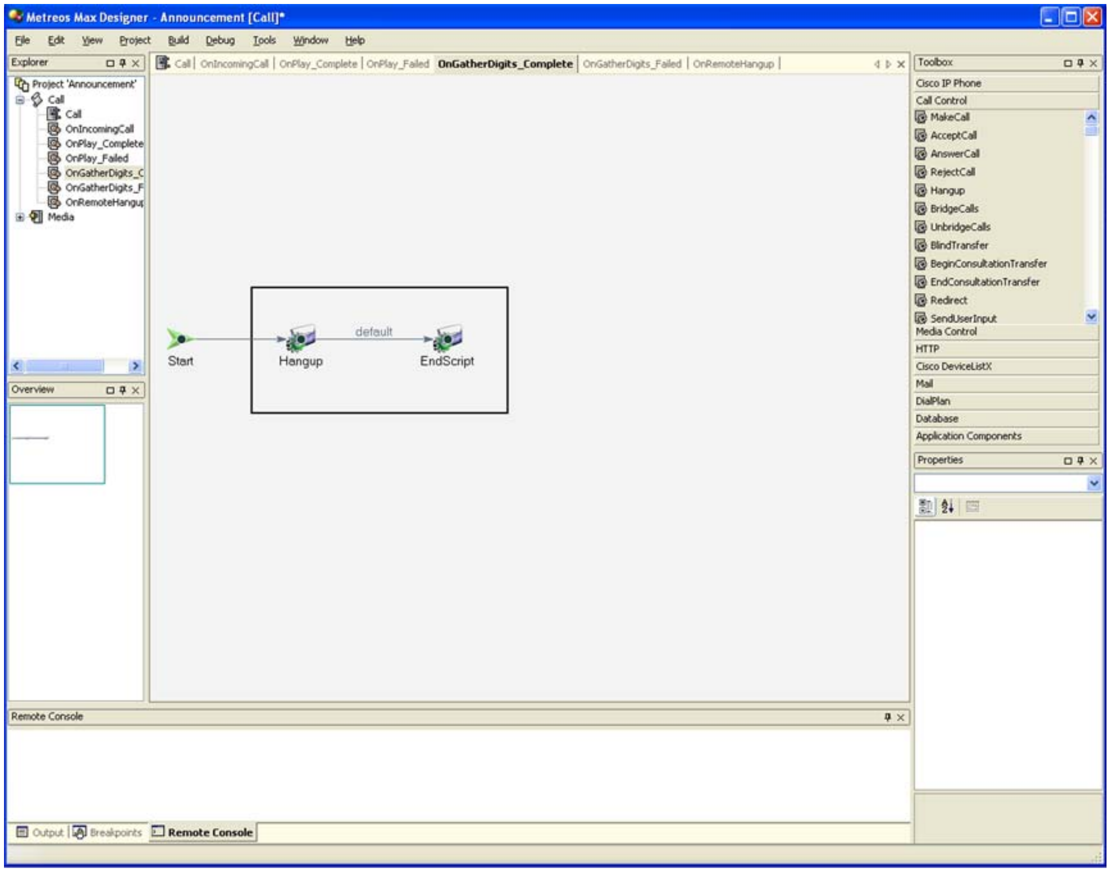

Figure 51: Selecting OnGatherDigits_Complete Function Components

Copy the components to the clipboard by pressing CTRL+C. You can populate the **OnGatherDigits_Failed** by pasting the components directly onto the canvas:

1. Click the **OnGatherDigits_Failed** tab.

2. Press CTRL+V.

The Visual Designer will retain the **g_incomingcallID** variable in the **Hangup** CallId property. There is no need to manually set the CallId property.

OnRemoteHangup

The script must be able to handle a remote hangup event. A remote hangup event occurs when the user hangs up before the script is complete. If the condition occurs, the only course of action is to end the script as shown in Figure 52.

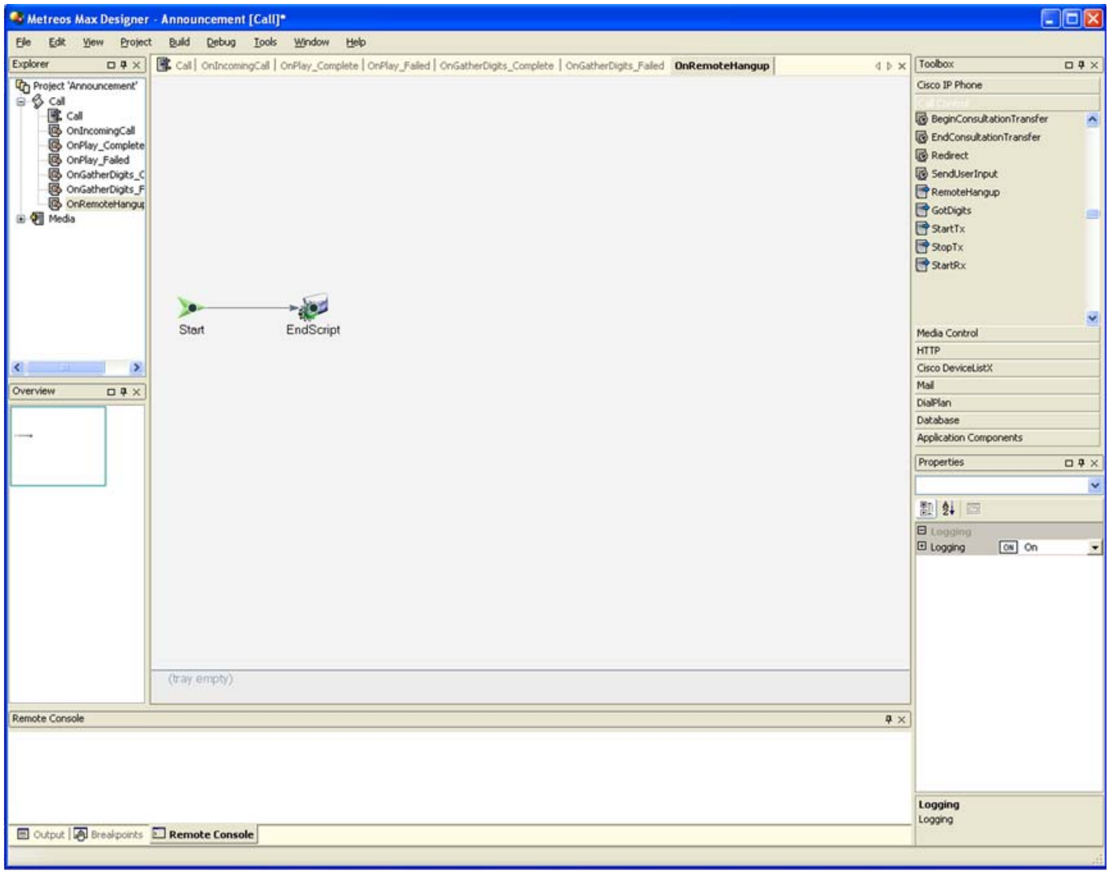

Figure 52: OnRemoteHangup

Drag a **RemoteHangup** event onto the canvas and construct the **OnRemoteHangup** function as shown in Figure 52. The **OnRemoteHangup** function is now complete.

Adding Media

The final step in developing the Announcement application is to add the audio file. Use the following procedure:

1. Select **File** → **Add Media** → **Audio File**. The visual designer displays a dialog requesting the file location as shown in Figure 53.

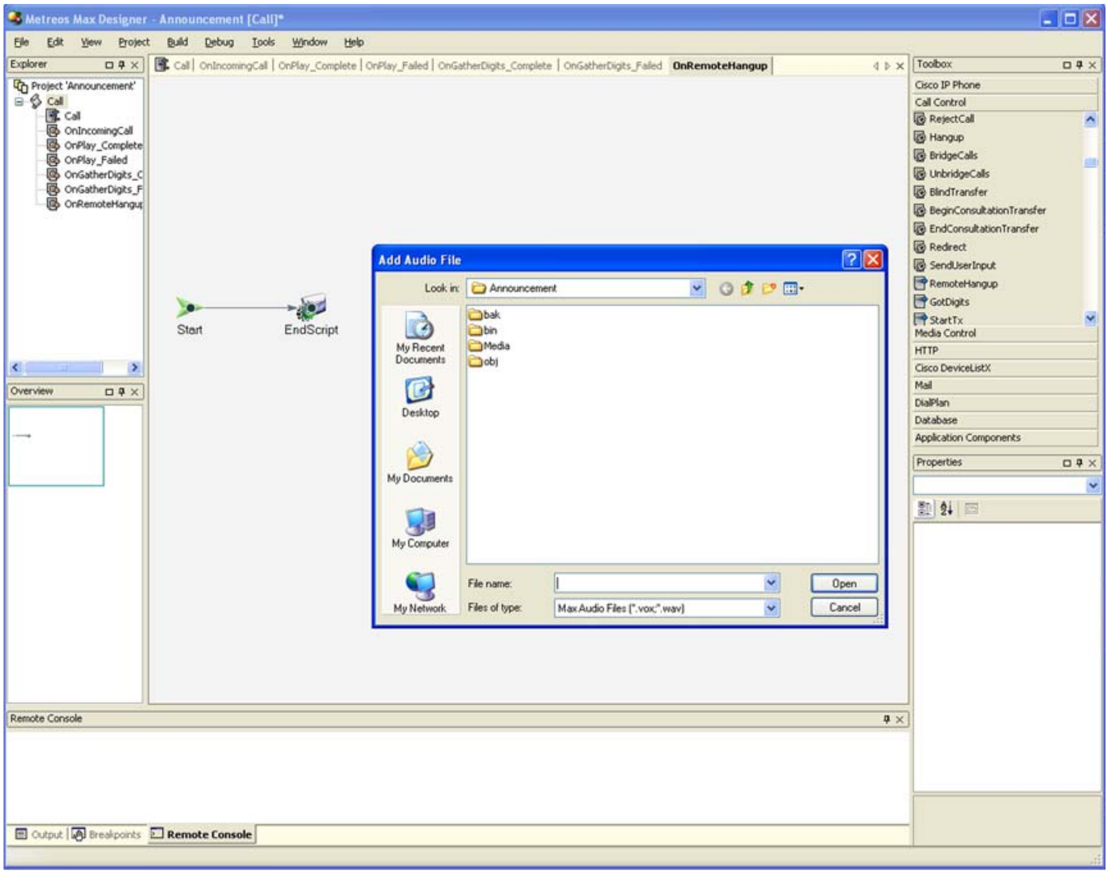

Figure 53: Adding an Audio File

2. Enter the path to the audio file including the filename, or navigate to the file using the exporer interface on the dialog.

3. Double click the file or click the file once and click **Open**.

The Visual Designer adds a new branch to the project tree in the Explorer Window called **Media** and adds the audio file to that branch as shown in Figure 54.

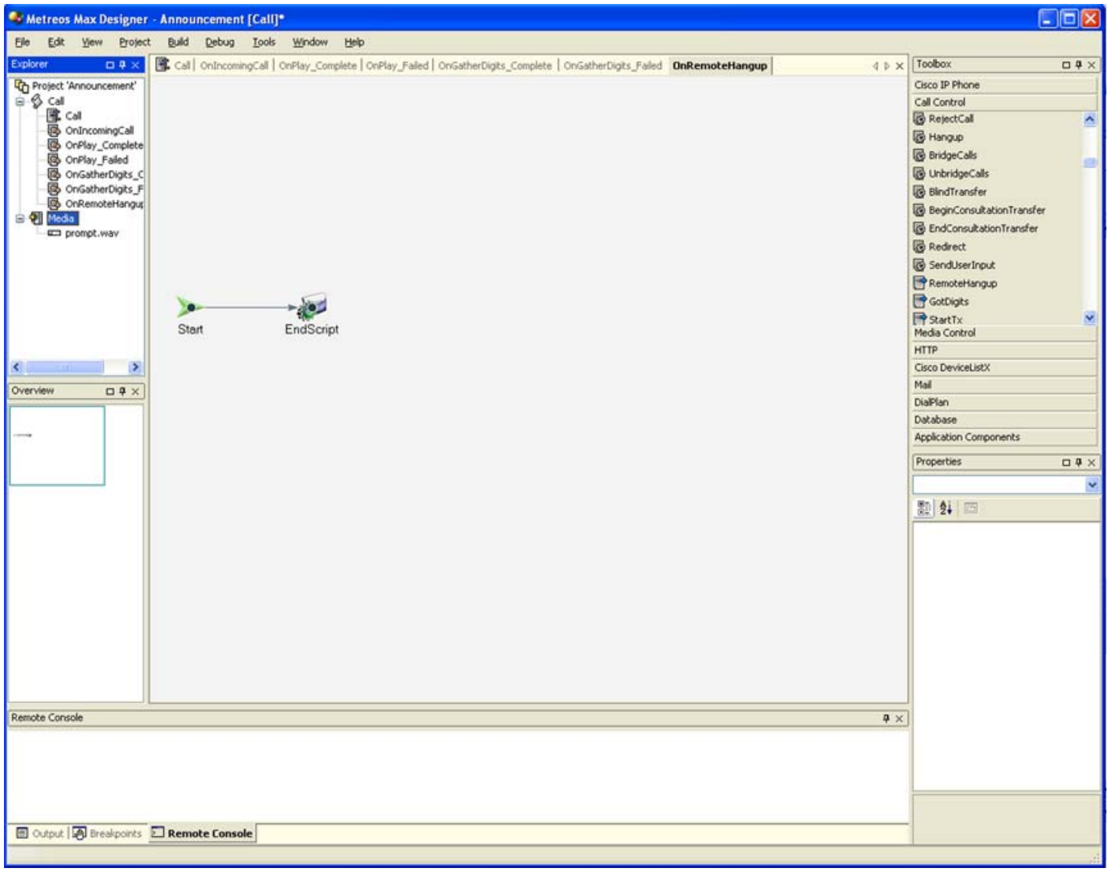

Figure 54: Completed Application

The Announcement application is now complete and can be built. To build, select **Build → Build Application**. You can then deploy the application by selecting **Build → Deploy**.

Script Execution

The following outline presents the event/action sequence when the Announcement application completes succesfully:

1. **OnIncomingCall**: (*function processing* state)
    1. Assigns CallId to a script-level variable
    2. Answers Call
    3. Plays Announcement
    4. Ends

5. **OnPlayComplete**: (*function processing* state)
    1. Records keys pressed by the user
    2. Ends

3. **OnGatherDigitsComplete**: (*function processing* state)

1. Terminates the call
2. Ends

3. Exits: (*uninstantiated* state)

Native Actions and Native Types 6

As previously mentioned, Native Actions allow you to add custom logic that executes within the process space of a script instance. Using Native Actions, you can extend the capability of the toolbox in the Metreos Visual Designer.

Developing Native Actions

Code Listing 3 shows a fragment that is an example implementation of a native action:

```
namespace Company.Native.Example
{
      [PackageDecl("Company.Native.Example", "Group of
actions")]
      public class Action : INativeAction
      {
      [ActionParamField("Parameter", true)]
          public  string Parameter { set { parameter = value; } }
             private string parameter;
  [ResultDataField("Result from action")]
             public  string Result { get { return result; } }
             private string result;
public  LogWriter Log { set { log = value; } }
             private LogWriter log;
public Action () {}
[Action("Action", true, "Action", "Performs task")]
             public string Execute(SessionData sessionData,
IConfigUtility configUtility)
             {
                   result = parameter;
                   return IApp.VALUE_SUCCESS;
      }
public void Clear()
             {
                   parameter  = null;
                   result = null;
             }
public bool ValidateInputs()
             {
                   return true;
           }
      }
}
```

Code listing 3. Simple INativeAction implementation

All Native Actions must implement the **INativeAction** interface. The Native Action must implement a default public constructor and the following three methods:

- ValidateInput
- Execute
- Clear

The **Execute** method is called by the runtime environment when the action is invoked. **ValidateInput** occurs immediately before **Execute**. **Clear** occurs immediately after **Execute**.

A Native Action can support multiple incoming parameters, as well as multiple outgoing parameters. The incoming action parameters are delineated by decorating a .NET property with an **ActionParamField** attribute. In this attribute, you can specify the name of the parameter as it will appear with in the Metreos Visual Designer.

You can also specify whether the action parameter is required by specifying a boolean value after the parameter name. A value of **true** indicates the parameter is required; a value of **false** indicates it is not required. In the case of our example, the parameter name **parameter** is followed by the boolean **true**, and it is therefore required.

NOTE*:* *If a required parameter is missing at runtime or if the parameter is not the specified type (or convertible to the specified type), the script will fail when the native action is invoked.*

Outgoing parameters are specified delineated by decorating a .NET property with **ResultDataField** attribute. This attribute allows you to optionally specify a description for the result data field.

You are assured parameters marked *as required* are present when the Execute method is invoked. All parameters are guaranteed to be of the correct type.

The **ValidateInputs** method allows you to cleanly perform additional validation of incoming parameters. If the validation fails, a return value of **false** will immediately stop script execution.

In the **Execute** method, you can perform any logic necessary to fulfill the task of the native action. Before exiting the method, assign the correct values to the result data fields. These values will be assigned to the corresponding variables in the **Result Data** section of the Native Action.

The Execute method returns a string. The value of the string determines what branch is taken after the action is executed. For example, if the action fails, you can set the string **failure** as shown in the following code fragment:

```
IApp.VALUE_FAILURE ("failure")
```

The constant **IApp.VALUE_FAILURE** returns the value you specify in the parentheses if the action fails. If the action succeeds you can specify that the constant **IApp.VALUE_SUCCESS** return **success** as shown below:

```
IApp.VALUE_SUCCESS ("success")
```

You can also specify a default return value as shown below:

```
IApp.VALUE_DEFAULT ("default")
```

These constants are located in the **Metreos.Interfaces** namespace so they are available to the native action. Return values of **success**, **failure**, or **default** are not required; any value is valid for branching. For example, you could instead use values of **successful**, **failed** and **other**, respectively.

You will find the **LogWriter** provided in the Execute method useful. Using this object through the Write method produces text output to the console (subject to configured LogLevel constraints). The following code fragment is an example of using the **LogWriter** to produce log messages:

```
using System.Diagnostics;
log.Write(TraceLevel.Error, "My error message");
```

The **SessionData** object in Code Listing 4 is provided as a convenience because it can be accessed in both CustomCode and Native Actions code. For more information about **SessionData**, refer to the **SessionData** table in "Application Control" on page 148 of Appendix A.

The **PackageDecl** attribute allows the specification of the namespace of the package in which the action resides and a description of the package. If any other actions exist in the project, it is not necessary to use the **PackageDecl** attribute. If they are in the same assembly, they are also in the same package.

LogWriter Property

Your Native Action must provide the following implementation for the **LogWriter** property:

public LogWriter Log { set { log = value; } }

private LogWriter Log

The **LogWriter** property uses its Write method to write messages to the Application Runtime Environment log. The Write method takes two parameters. The first parameter is of type **System.Diagnostics.TraceLevel**, and it is used to specify the output log level. The second parameter is the string that will be printed to the log. Details about the TraceLevel enumeration are available at the following URL:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfsystemdiagnosticstracelevelclasstopic.asp

Action Attribute

The Execute method must be decorated with the Action attribute as shown in the original example.

```
namespace Company.Native.Example
{
      [PackageDecl("Company.Native.Example", "Group of actions")]
      public class Action : INativeAction
      {
        [ActionParamField("Parameter", true)]
           public  string Parameter { set { parameter = value; } }
           private string parameter;
        [ResultDataField("Result from action")]
           public  string Result { get { return result; } }
           private string result;
        public  LogWriter Log { set { log = value; } }
           private LogWriter log;
        public Action () {}
        [Action("Action", true, "Action", "Performs task")]
           public string Execute(SessionData sessionData,
IConfigUtility configUtility)
           {
                 result = parameter;
                 return IApp.VALUE_SUCCESS;
      }
        public void Clear()
           {
                 parameter  = null;
                 result = null;
           }
        public bool ValidateInputs()
           {
                 return true;
        }
      }
}
```

Code listing 4. Simple INativeAction implementation

The **Action** attribute construct from Code Listing 4 is the following code:

```
[Action("ActionName", true, "ActionDisplay", "Performs task")]
public string Execute(SessionData sessionData, IConfigUtility
configUtility
```

The following parameters are defined in the Action attribute:

- *ActionName* **—** The name of the action
- **True/False —** A boolean describing whether the action allows custom parameters
- *ActionDisplay* **—** The display name in the Metreos Visual Designer
- *Performs task* **—** A description of the action

ReturnValue

When a Native Action is executed, it must return a **ReturnValue**. This requirement makes possible the branching that occurs in application scripts that use the Native Action you defined.

Return values are specified during development. After you drag the Native Action element onto the function canvas and connect it to another element, you can label the logic flow arrow with a valid return value. Refer to "Assigning Flow Control Labels" on page 67 for information about assigning return values to actions for flow control.

The Metreos Visual Designer allows two techniques for assigning return values during application development:

- Manually type the valid value at the time you connect the Native Action element during development
- Select the element from a dropdown list at the time you connect the Native Action element during development

If you want your Native Action to return a set of values other than **Success**, **Failure** or **default**, you must decorate the **Execute** method with the **ReturnValue** attribute. You can use one of three techniques to define the **ReturnValue** parameter. Code Listing 5 below shows a code fragment that represents the simplest technique:

```
[ReturnValue()]
```

Code listing 5. Implicit Return Value SuccessFailure

The code fragment specifies that the action returns either success or failure. The dropdown list on the flow control will contain **Success** and **Failure**.

NOTE: *This attribute is always implicitly assumed by the application server, and hence will always be present.*

A second technique is show below in Code Listing 6.

```
[ReturnValue(Type enumType, string description)]
```

Code listing 6. Explicit Return Value

The code fragment explicitly specifies the return value. The *enumType* argument must be of type **System.Enum**. The names of the enumerations within *enumType* constitutes the set of the action's return values, in addition to the standard SuccessFailure set. You must provide one of these statements for each return value you want to specify. Code Listing 7 shows a code fragment that describes a third technique for specifying return values:

```
ReturnValueAttribute(bool openSet, string description)]
```

Code listing 7. Open Set Return Value

The code fragment returns an *open set*. An open set is one in which any return is valid.

Adding a Native Action to a Metreos Visual Designer Project

Use the following procedure to add a Native Action to a specific project:

1. Build your Native Action code
2. Open the project to which you want to add the Native Action using Metreos Visual Designer
3. Select **Project** ⟶ **Add Reference** as shown in Figure 55.

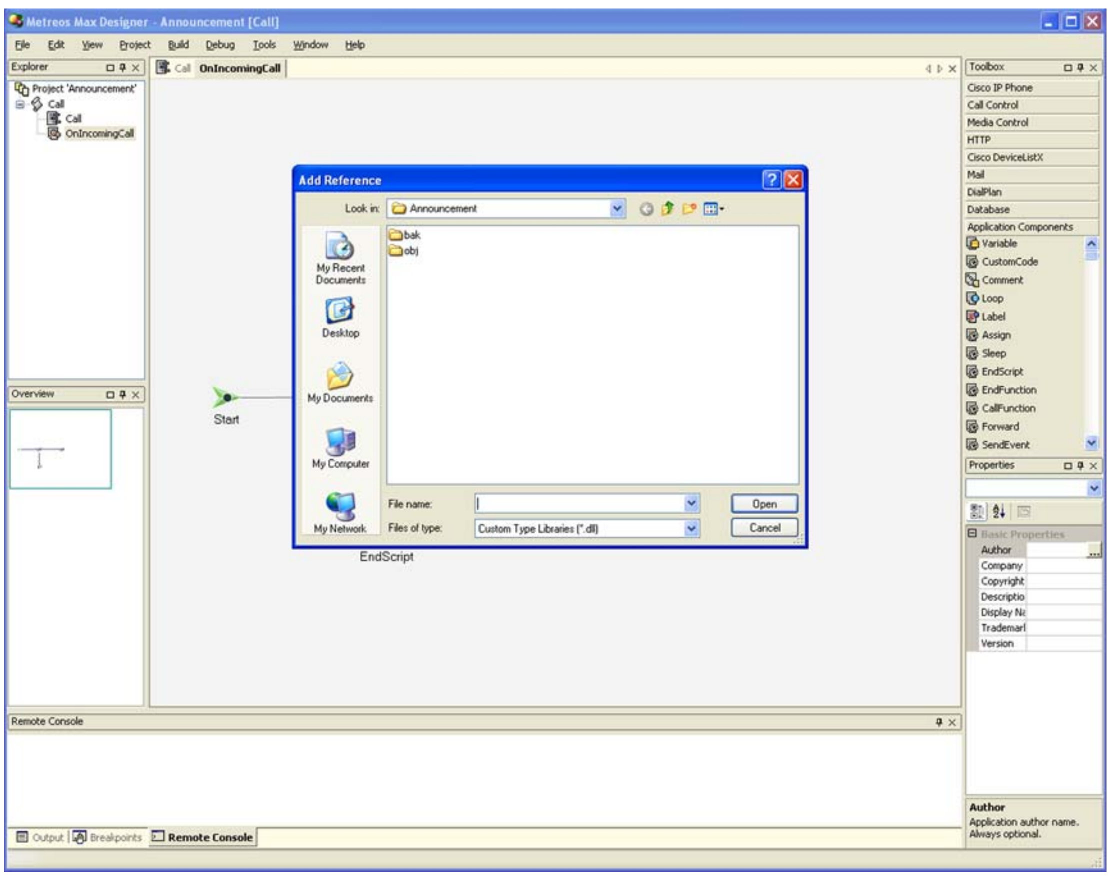

Figure 55: Adding DLL Reference

4. Enter the DLL path.

5. Create a new tab in the toolbox to contain your Native Action elements.
    1. Right click on any tab and select Add Tab. Visual Designer will create a tab with a sequentially assigned default name such as New Tab 1.
    2. Replace the assigned name with a name of your choosing.

3. Add the Native Action elements to your new tab in the Toolbox.
    1. Right click the new tab and select Add/Remove Items.
    2. Select the Native Action Elements.

Developing Native Types

When you develop applications, new data types must sometimes be created to encapsulate different forms of data. The MCE allows developers to extend its type system by building *native types*. A native type is a .NET class implementing a well defined interface exposed by the MCE Application Framework.

```
namespace Company.Types.Example
{
      public class Type : IVariable
      {
            public  string ExampleProperty { get { return _value; } }
            private string _value;

            public Type()
             {
              }
            public bool Parse(string newValue);
            public string ExampleMethod(string var)
            {
            return _value + var;
            }
      }
}
```

Code listing 8. Simple IVariable implementation

All native types must implement the **IVariable** interface. This interface requires the implementation of **Parse(string)**.

Parse(string) is required so the data can be initialized with default values in the Metreos Visual Designer. Users can specify only strings for a default value in the Visual Designer; other data types such as **int** cannot be specified for default values, and **Parse(string)** is therefore required.

It is not necessary for the **Parse** to actually do anything; default values have no meaning in the Visual Designer for this variable type. It is required only to pass the default value to the variable.

Parse should always return true except in a case of critical failure; otherwise, the script attempting to initialize the variable will immediately stop.

Variables that implement the **IVariable** type interface are also assigned through the **Result Data** fields of native or provider actions. If the type being returned by an action is the same type as the variable receiving the value, a direct assignment is made. If this condition is not met, then Parse(string) is invoked. Alternatively, any overload of Parse better matching the incoming type from the action will be invoked.

Various incoming types can be assigned to your Native Type, based on the number of Parse overloads you provide. The type used in a Result Data field of an action is the type passed into the Parse method. Alternatively, you can catch all assignment attempts with Parse(object obj), and use the .NET is operator to check for various types.

In Code Listing 8, **ExampleProperty** and **ExampleMethod** are included to demonstrate how to access data contained by this type in actions.

Deployment of Native Types is very similar to deployment of native actions. Both use fully qualified names whose namespaces must match the name of the .NET assembly file holding the code. In Code Listing 8, the code would be written in the **Company.Types.Example** namespace and would be located in **Company.Types.Example.dll**.