Metreos Communications Environment

Application Developer's Guide

August 2004

Version 1.1

1. INTRODUCTION AND OVERVIEW

Welcome to the Metreos Communications Environment Application Developers Guide. This document explains how to design, develop, and deploy applications for the Metreos Communications Environment, or MCE.

Who Should Read this Book

This book is intended as a primer on the principles, architecture and capabilities of the Metreos Communications Environment, and provides a foundation for building IP telephony applications using the Metreos Visual Designer. If you are a developer interested in building IP telephony applications using the Metreos Communications Environment, a system administrator responsible for maintaining installations of the MCE, or a manager evaluating the Metreos platform for deployment, this document is for you.

Style and Formatting Conventions

A variety of style and formatting conventions are used in this document. These conventions, along with examples, include:

Inline Code: Code fragments and other text which might appear as part of a program listing are shown as single-spaced text. For example, `inline code snippets`. Note that spacing and special characters within code samples indicate actual requirements for development.

Code Snippets: Full sections of code appear as gray blocks encasing code blocks. Bold-faced code represents **`reserved words`** or other key concepts. Example:

```
Code Snippet
Code Snippet
```

Key Concepts: Important ideas or meaningful terms within this document appear in **bold**. These words also appear in the index.

Internal References: Text which leads the reader to other chapters or nearby figures appears in *italics*.

How to use this book

This book is intended as a companion to a Metreos training program, and is distributed only to participants in such a program. Diagrams, examples, and explanations in this book receive extended coverage during these sessions.
This book can be used by developers interested in learning about application development for the Metreos Communications Environment. The material presented herein will enable the software engineer to develop powerful telephony applications.

Design Goals

Much of this book consists of the technical details of the Metreos Communications Environment, with a few sample applications to illustrate concepts. Although this book contains a variety of examples, it is neither a "cookbook" nor simply a reference manual. Instead, this book aims to educate developers on the fundamental structure of the Metreos platform. With this foundation, engineers will be able to lead their organization in developing powerful telephony applications on the Metreos Communications Environment.

The Metreos Communications Environment

The Metreos Communications Environment (MCE) is a feature-rich platform for developing and hosting IP telephony applications. At the core of the MCE is a powerful application server, which controls media and external resources under the direction of custom telephony applications. These tools control a scalable software-based media server which processes, mixes, analyzes, and routes digital audio data. Building these software programs is accomplished through an easy-to-use visual designer, which allows developers to create complex telephony applications with literally a few mouse clicks. System administrators manage the MCE through a web-based interface referred to as the System Control Panel.

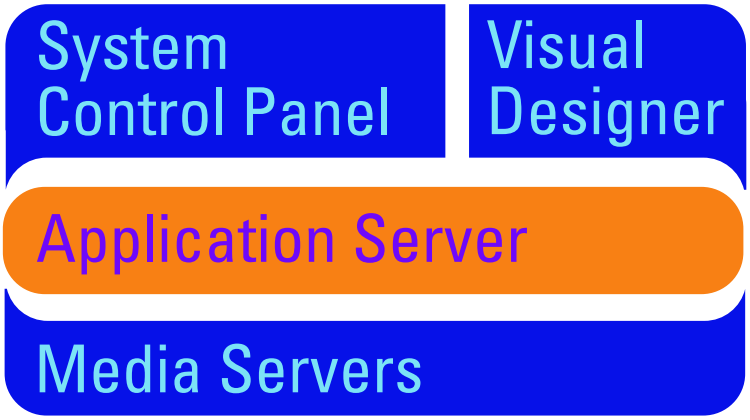

Figure 1: Metreos Communications Environment Architecture

The Metreos Application Server

The Metreos Application Server (MAS) serves as the core component of the MCE. Within the MAS reside all installed applications, a virtual machine for executing applications and a variety of protocol providers which extend the capability of the platform to reach third-party components.

Just as applications designed for Microsoft Windows™ must be installed on a system before use, applications designed for the MCE reside within an application pool in the MAS. The platform provides automatic management of application state, handles side-by-side existence of varying application versions, supports seamless installation of new applications and trouble-free un-installation of any application not currently in use.

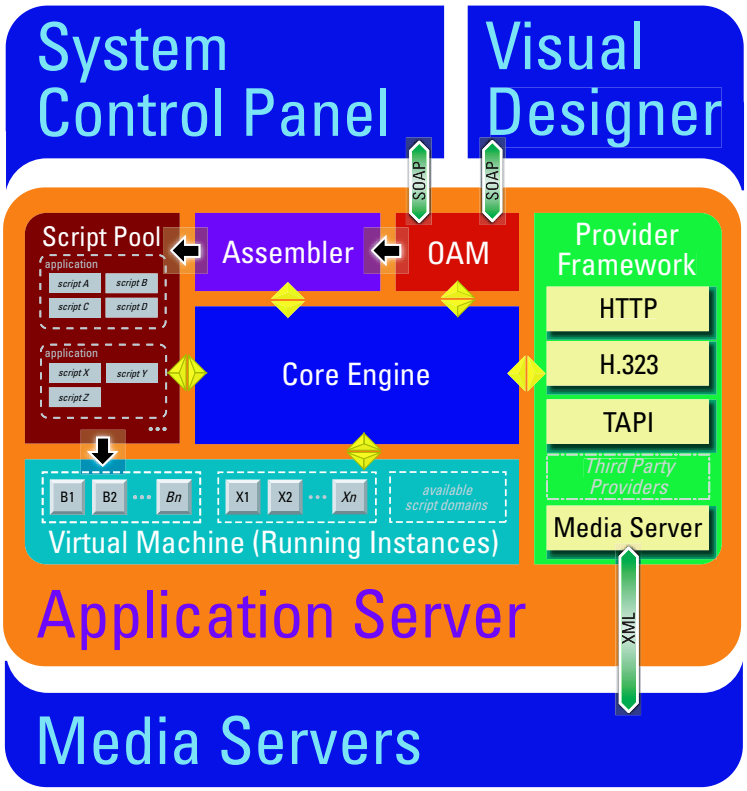

Figure 2: Application Server Diagram

Applications communicate with other components of the MCE as well as third-party systems via protocol providers. Metreos provides an H.323 provider for first-party call control, a TAPI provider for third-party call control, an HTTP provider for network communication, a Timer provider for creating event-driven delays, a Cisco DeviceListX provider for caching CallManager device information, and finally a Media Server provider for communication with the Metreos Media Server (MMS). Metreos continually releases additional providers to integrate the MCE with other enterprise systems. Developers interested in creating protocol providers should refer to the *Metreos Communications Environment: Advanced Developer's Guide.*

The Metreos Media Server

At the heart of any digital telephony system is media. Managing the digital audio of IP telephone conversations is a task traditionally reserved for expensive, high performance hardware switches. The Metreos Media Server (MMS) provides the full functionality of hardware-based media servers but with a pure software implementation, thus freeing customers from the burden of investment in expensive hardware which soon becomes outdated. Inherent in this software-only design is the ability to automatically keep pace with inevitable advancements in processor speed and capabilities. Furthermore, the design guarantees interoperability with standard telephony and networking protocols, such as RTP, as well as scalability and redundancy through multiple MMS installations on parallel servers.

Each MMS installation supports up to 120 simultaneous connections, which equates to dozens of simultaneous user sessions spread across various applications. Since as many as eight media servers may be paired with one application server, additional service levels can be instantly provided by simply adding additional media servers.

The MMS includes a variety of powerful features to enable complex telephony applications, including support for media streaming, DTMF interpretation, multi-party conferencing and recording. Metreos plans additional features in future releases, including text-to-speech, speech recognition and Voice XML support.

Metreos Visual Designer

Traditionally, developing IP telephony applications required immense resources and intimate knowledge of obscure systems. The MCE radically simplifies the process of developing and deploying applications through use of the Metreos Visual Designer (MVD).

Developing an application requires neither programming expertise, paging through countless lines of arcane program code, nor a deep understanding of call control or media processing. Instead, the MVD presents a graphical user interface where application components may be created and interconnected with a few clicks of the mouse. Finished applications can be deployed to the MCE via a menu selection or uploaded using the System Control Panel as described below.

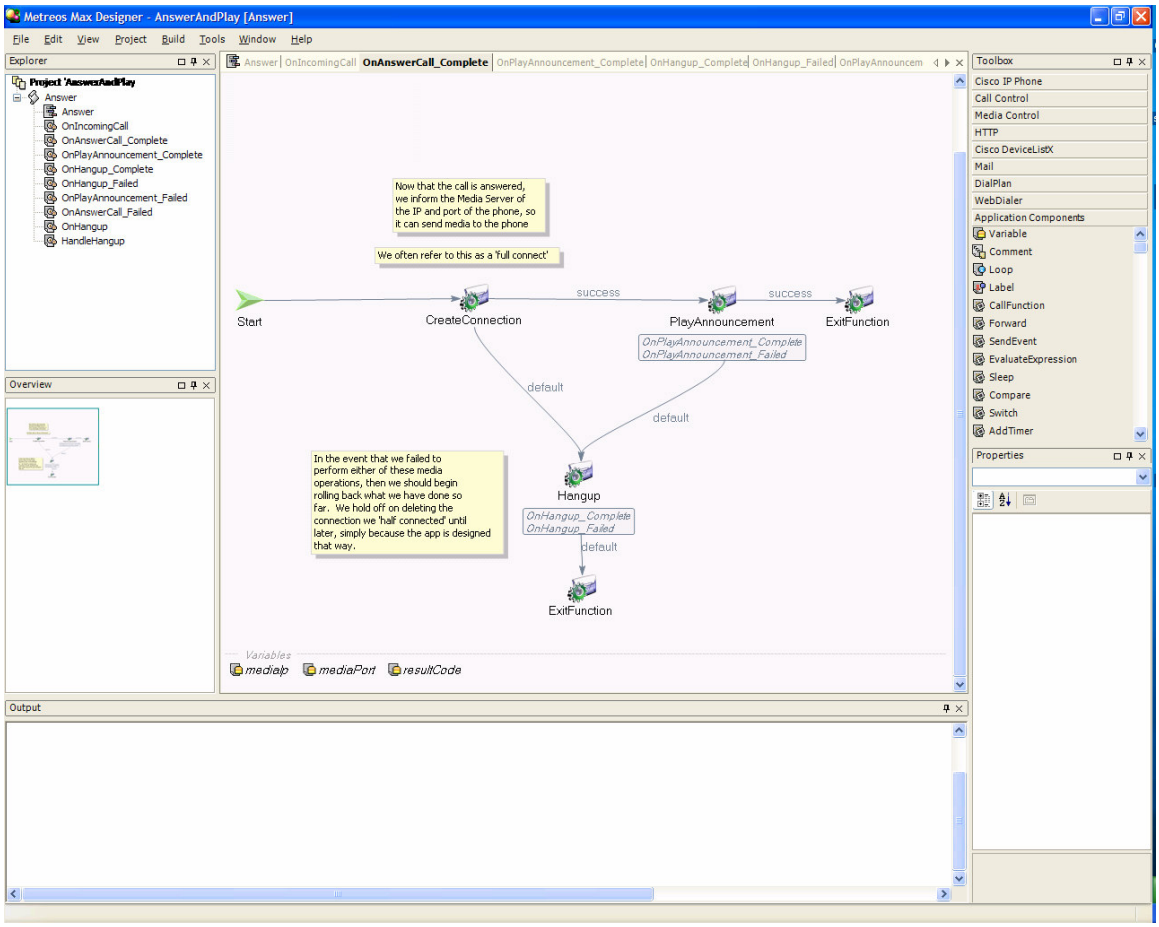

Figure 3: The Metreos Visual Designer

System Control Panel

Administrators responsible for managing the MCE do so by using the System Control Panel (SCP). This web-based interface permits configuration management of the Metreos Application Server, all associated installations of the Metreos Media Server, user and license management, as well as control over system configurations, providers, and applications.

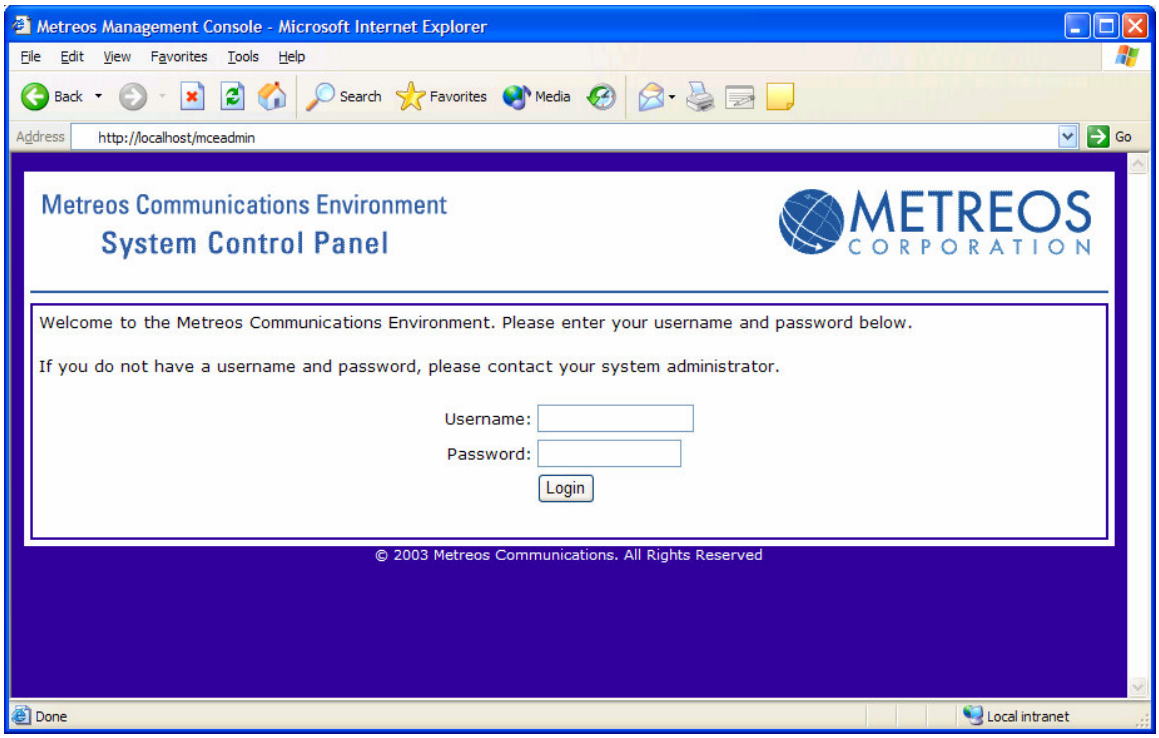

Figure 4: The System Control Panel

The System Control Panel is resident on the same machine as the Metreos Application Server by default, but can be relocated to a dedicated web server if desired. Details regarding use of the SCP reside the document entitled *Metreos Communications Environment: Administrator's Guide.*

Examples of Use

Harnessing the power of the Metreos Communications Environment enables developers to produce virtually any telephony application imaginable at a fraction of the cost and time required by traditional systems. Some potential applications include:

Microsoft Exchange™ Integration: Provide alerts when meeting events occur by ringing users on their mobile phones.

Instant Messaging and Collaboration Integration: Easily start collaborative conference calls from instant messaging sessions.

Voicemail: Provide customized, flexible voicemail services to meet individual organizational needs.

Conferencing: Take advantage of the rich features in the Metreos Media Server to support instant recordable conferencing with participant mute and kick.

Click-To-Talk: Extend a desktop PIM client (such as Microsoft Outlook™) with a protocol provider to enable one-click calling between any parties in your address book.

Location-Based Forwarding: Integrate with your enterprise IT authentication system to automatically forward incoming calls to your home phone, mobile phone, desk phone, based on your system login trail.

These examples can be produced at remarkably low costs and on short schedules due to the flexibility and power offered by the Metreos Communications Environment. For more information on developing these specific applications as well as other examples, please see *Metreos Communications Environment: Developer's Cookbook.*

2. IP TELEPHONY APPLICATIONS AND THE MCE

An **IP telephony application** within the Metreos Communications Environment typically combines voice and data into a unified application. A basic understanding of both traditional telephony and data networking features is necessary for the development of applications which fully realize the potential of convergence.

If you have ever used a touch-tone telephone to navigate a support menu, to leave a voice mail message, or to participate in a multi-party call, then you are acquainted with traditional telephony applications. Telephony applications such as these have become so commonplace, we may often not be aware of their existence.

Figure 5: Cisco IP Telephone

IP telephony applications introduce the telephone as a node on the IP network. In addition to point-to-point calling, telephones can serve as the origin or destination of a variety of network services. Furthermore, telephones are no longer limited to voice and the painful limitations of touch-tone menus. IP phones sport rich interfaces with interactive touch-screen color displays.

Traditional Telephony Application Development

The telephone system initially emerged as a simple, circuit-switched network, with call control centrally managed by human operators. During the first fifty years of the telephone, placing a call involved lifting the receiver, telling the operator an identifying code for the remote party, and waiting to be connected. Then, the operator would insert a connecting cable between the two lines on their switchboard to connect the call.

Skilled operators on low-traffic circuits developed the first telephony applications by simply building relationships with customers. If a call went unanswered, operators could

volunteer to take a message from the caller, passing it along the next time that receiver tried to place an outbound call. This crude system forms the basis for modern day voice mail in which audio messages may be saved and retrieved.
As technology advanced, electronic components replaced human operators. People began placing calls by dialing instead of asking to be connected. Dial pulses (eventually replaced by dial tones), signaled telephone equipment to route calls. Although calls could be placed more quickly and more reliably, the personal touch of friendly operators disappeared, and the additional services provided to the customer community vanished.

As the use of computers became prevalent in the telephone industry, companies tried to provide the same telephony applications once offered only incidentally by switchboard operators. The architecture of the public switched telephone network made this task exceedingly difficult. The most obvious method for providing access to new features would involve the sounds produced in touch-tone dialing. While telephone switch equipment could use these tones for call routing, the additional processing needed to manage this data while providing application functionality proved extremely costly. To install a new feature, such as voice mail, meant purchasing expensive new hardware for each circuit.

As a consequence, telephony applications reappeared first at the edges of the network, in small offices or remote communities–just as they did when helpful switchboard operators had enough spare time to take a note or forward calls. This revolution helped catapult the rise of the PBX, or Private Branch eXchange, as the internal telephony interconnect within a building or organization.

A PBX originally served two purposes: permitting a large number of end-users to share access to a small number of outbound lines and connecting internal users without having to make a complete circuit to the telephone company and back. As the PBX became more powerful and commonplace, it began to include additional services such as call forwarding and voice mail. The increasing number of PBX vendors helped drive costs down but also resulted in each vendor utilizing proprietary systems for application development, making widespread application development prohibitively expensive, or even impossible.

Two principal issues plague application development for PBX systems. The first involves the lack of standards in programming language, development environment, and signaling protocols. Telephony programming languages are frequently arcane and lacking in structure. Many utilize a non-procedural architecture and therefore have little similarity to the familiar coding styles of programming languages like C/C++, C# or Java. Additionally, communication with PBX hardware often involves closed, proprietary signaling protocols. Since the PBX provider rarely anticipated a need for any cross-platform or inter-system development, documentation on the structure of these protocols is often unavailable.

The second issue hampering PBX application development relates to the inherit extensibility and scalability problems in pure hardware media processing. Although the

use of hardware to analyze media streams and convert analog data for digital manipulation and storage is inexpensive and fast, most PBX signal processing hardware lacks the capability to "learn new tricks". Adding media services, such as speech synthesis and recognition, either requires buying a complete replacement PBX or is, in many cases, simply not offered by the vendor.

Furthermore, PBX support for scalability only rises to meet the initial design expectations of the vendor. If an organization doubles in size, adds a support department, or otherwise changes their telephony needs, their existing PBX may not be able to handle the new services by simply using drop-in expansion cards. It may require an additional parallel system, replacement of the existing system with a larger installation, and for many PBX vendors, costly custom development. Often, an organization's telephony goals cannot be accomplished within the rigid framework defined by PBX hardware.

In the 1980's and 1990's, the industry worked to resolve this issue with the development of a standard for telephony application development: TAPI, the Telephony Application Programming Interface. Unfortunately, vendors were heavily invested in their own systems and few moved quickly to adopt the standard. Furthermore, the specifications for TAPI lacked adequate clarity so implementations varied from system to system.

Perhaps most importantly, TAPI and its successor, JTAPI, were incredibly difficult to use since they required developers to have a deep understanding of the telephony switching to perform even the most basic functionality. Consequently, developers and IT organizations avoided the standard and continued to invest in vendor-specific PBX systems.

Thus, the industry has seen relatively little advancement in telephony applications in recent years, even with the gradual adoption of Voice over Internet Protocol (VoIP) technologies which reduce networking costs and potentially bridge voice and data networks. So while current PBX systems can communicate with IP telephones using the same data network shared by computers, they continue to lack the programmability, flexibility, and scalability promised by the advent of network convergence. Consequently, many enterprises continue to reply upon administrative staff for telephony applications, harkening back to the switchboard operators of yesteryear.

Advantages of MCE Applications

The Metreos Communications Environment enables enterprises to leave behind the problems and frustrations of telephony application development using traditional PBX systems. In contrast with esoteric programming languages and vendor-specific call control and signaling, the MCE provides a simple drag-and-drop interface with built-in support for international telephony standards. Whereas PBX systems use limited custom hardware to process media, the MCE boasts a pure software media server, which automatically scales to support growth. Furthermore, the quality of service and overall performance of applications developed with the MCE will automatically increase proportionally with advances in processor speed and capabilities.

Perhaps the most marked contrast between the Metreos Communications Environment and legacy hardware-based offerings from other vendors manifests itself through the convergence of data and voice networks. With the MCE, developers can easily connect to third party resources and infrastructures to exchange data, execute remote operations, and shuttle information to and from users. **Protocol providers** are hot-swappable components which enable applications to respond to events from the network and utilize any available services. Developers can easily create custom protocol providers to connect with any third-party system. For more information on provider development, see the *Metreos Communications Environment: Advanced Developer's Guide*.

Finally, the architecture of the MCE facilitates automatic application independence. Developers may write multiple applications which respond to similar user events, each of which may execute simultaneously. The MCE manages any shared resources within applications through a process called **provisioning**.

Applications and End Users

Users accessing applications by way of a telephone will experience much of the interface as prerecorded messages, or **announcements**. If your application needs to present unique messages to users regarding system status, interface feedback, or other data, you will need to use recorded **announcements**.

Additionally, some users and applications may take advantage of the interactive capabilities of devices like the Cisco™ line of IP phones. Cisco provides an extensive framework for presenting images, menus, forms and options on large, touch-sensitive screens. Developers using the Metreos Visual Designer can quickly and easily produce professional interfaces for the Cisco™ IP Phone. For more information on the Cisco™ IP Phone services available within the MCE, see the document entitled *Metreos Communications Environment: Framework API Reference*.

Other applications may require connectivity with external resources to retrieve data for users or facilitate user input. For example, consider a telephony application which enables a website visitor to initiate a call between two parties. The web pages leading to this effect serve as the primary interface, and thus define the quality of the user experience. Developers should consider these requirements and plan accordingly, preferably before development begins.

Existing Telephony Applications and Features

Telephony services can provide so much more than a simple voice connection, but in all cases, the phone call forms the basis for the application. Given the prevalence of telephones in our personal and business lives, one can fairly easily list examples of common telephony apps:

Call Forwarding – Perhaps the simplest of all apps, forwarding occurs when the system redirects an attempt to dial a number to a different end point. Such an application can be

used by travelers, those who work in multiple offices, or who utilize other answering services.

Call Waiting – Now commonplace in business and the home, call waiting enables a customer to accept an incoming call while currently on an existing call. This application demonstrates an ideal enhancement, as it gives the user the option to accept an interruption, rather than blocking all interruptions (a busy signal) or accepting unwanted interruptions.

Voice Mail – Like the switchboard operators of the early telephone system, voice mail lets us decline a call but still receive the message. With voice mail, telephony transcends the chasm between synchronous and asynchronous communication. Users no longer need to be available at the right time to communicate anywhere in the world.

Conferencing – Early telephone systems automatically supported limited conferencing, as everyone within a local loop literally shared one extension. With some costly cooperation from the telephone companies, moderns systems can connect multiple parties in disparate locations across the globe. The Metreos Communications Environment enables you to produce powerful instant, multi-party conferencing applications at no additional cost.

These applications merely begin to demonstrate the range of services possible with either traditional or IP telephony systems. Implementing such applications using a traditional PBX requires months or even years of planning, testing, and costly development. Using the Metreos Communication Environment, however, a single developer could easily produce any of these examples in a small fraction of the time.

Potential IP Telephony Applications

Countless possibilities exist for new and unique IP telephony applications given the technologies and ease of development offered by the MCE. Some examples include:

Click-to-Talk – Placing a call usually begins by flipping through an electronic address book, picking up your receiver, dialing each contact number in sequence until finally reaching your party. Imagine a button within the address book which rings each destination number in sequence (or knows the user's current location), and then rings the calling phone once a connection has been established. This application, which would traditionally require several months of development by a team, can be created in just a few days by one or more developers using the MVD.

Factory Timecards – Rather than purchase reams of paper cards, time clocks, and hire scores of accountants, why not use a regular phone and a simple telephony application to enable employees to clock in and out with the punch of a button. Work hours could then be automatically stored and tabulated in a financial application for company records and performance reviews.

Sales/Support Call Request – Enable website visitors to enter a telephone number and request an immediate call from a company representative. Not only can the Metreos Communications Environment respond to incoming HTTP connections from a website, but it can automatically manage a phone pool to ensure that calls are routed to the next available representative.

Pizza Delivery – Reduce costs and increase efficiency with a powerful system for placing food delivery orders. Tie the menu system to inventory control to ensure offerings match deliverables, to account information to provide incentive-based pricing for return customers, and to billing services for instant credit or check charges.

Self Service Reminder and Wake Up Calls – Help people remember important events and appointments with an automated reminder application. Calls can be programmed through a website or auto-attendant system, with recorded messages provided by the user or drawn from a provided library of announcements. Customers might use this service to automatically remind all invited guests of an upcoming party, or manage RSVPs without time-consuming intervention.

Unified Messaging – Eliminate the gap between disparate communication mediums like email, telephones, and instant messaging with an MCE application designed to span all mediums. Build an access service to a universal inbox, with all incoming voicemail, email and instant messages. Provide location-based delivery service, so that recipients receive messages through the most convenient media at the time of receipt.

Sales Chain Integration – Use telephony and data networks to coordinate inventory from suppliers with demand from customers by integrating touch-tone sales menu into restock ordering. Optimize returns by profiling customer sales/support calls and coordinating with promotional opportunities.

Emergency Broadcast – Quickly push critical information to parties including announcements, maps, or other audio/visual data in the event of a crisis. Use automated services to require a verbal response from all staff to confirm their safety.

SMS Integration – Bridge to the wireless telephony space. Register user location and push SMS messages to a desktop or a nearby IP phone. Enable automated SMS responses for news and information services.

LDAP Directory Integration – Automatically display caller name, details, and photograph on the computer screen or IP phone screen when a call is placed or received. Navigate the LDAP directory from any phone anywhere within the organization.

3. MCE System Architecture

The flexible architecture of the Metreos Communications Environment (MCE) lets developers produce powerful IP communications applications with minimal effort. Understanding this architecture is crucial for successful development and deployment.

MCE Overview

The MCE consists of four principal components: the System Control Panel, the Visual Designer, the Application Server, and the Media Server. These components constitute the MCE architecture as shown in Figure 6.

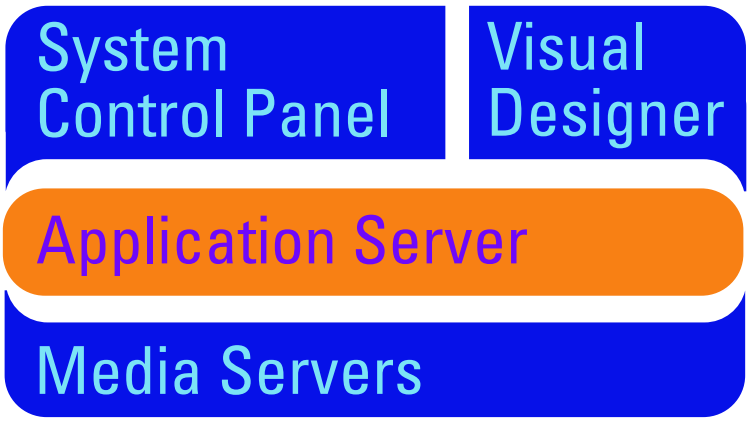

Figure 6: MCE Architecture Overview

The Application Server acts as the core component of the system facilitating interaction between the administrator, development interfaces (the System Control Panel and the Visual Designer) and the Media Server. The Media Server comprises the base of the MCE platform. Since telephony applications rely heavily on the ability to record, play, analyze and produce media, this component is of paramount importance. The MCE supports up to eight media servers per application server, allowing for hundreds of media processing ports to be deployed.

The System Control Panel permits administrators to manage the configuration of the MCE, install and remove applications and providers, and control other aspects of routine operation. Use of this component is covered extensively in the document *Metreos Communications Environment: Administrator's Guide*.

Finally, the Visual Designer enables knowledge workers, network administrators, and software developers to rapidly build, deploy, and execute new IP communications applications using an easy-to-use drag and drop visual interface.

Application Server Architecture

The Metreos Application Server handles all communication between components, processes user input, controls the media servers, and provides interfaces for the System Control Panel and the Metreos Visual Designer. These capabilities make the architecture

of the application server necessarily complex, and thus the primary functional units can be grouped into subcomponents.

Figure 7 reveals the six subcomponents of the application server: the **Script Pool**, **Assembler**, **OAM**, **Provider Framework**, **Virtual Machine**, and **Core Engine**. Each element communicates with other components through a messaging infrastructure controlled by the Core Engine. This interconnection is shown as yellow arrows drawn between each component and the Core Engine.

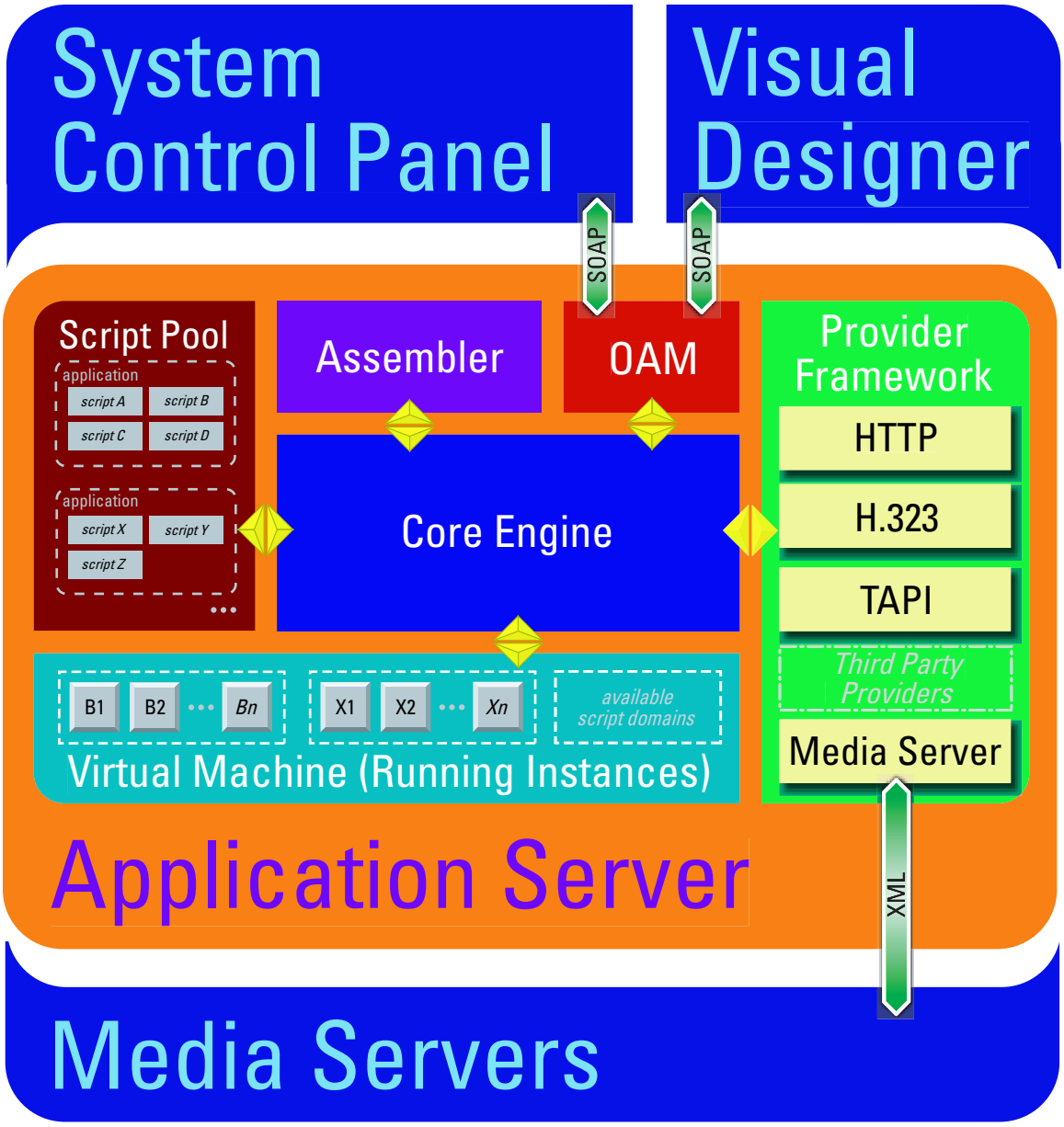

Figure 7: Metreos Application Server Architecture

Provider Framework

IP telephony applications within the Metreos Communications Environment interact with the outside world through **protocol providers**. Providers offer a family of services typically associated with a communication protocol. For example, the HTTP provider enables applications to accept incoming web traffic and respond accordingly, or make HTTP requests to other hosts elsewhere on the network.

Protocol providers resemble a Unix daemon process or a Windows service. They execute within their own virtual process space and facilitate communication between the Application Server and external systems. Providers play a critical role in the operation and execution of the MCE because they are the sole means in which applications may execute **asynchronous** operations and are the only sources for **unsolicited** events.

Protocol providers have two primary functions. First, they must respond to actions received from applications executing within the Metreos Application Server. Secondly, they handle data received from external services and generate events to be handled by applications within the Application Server.

The Provider Framework is a well-defined API that enables third-party developers to build extensions to the MCE. By implementing the interfaces defined within the Provider Framework, developers can extend the Metreos Communications Environment to any external system or protocol that they desire. For more information, see the *Metreos Communications Environment: Advanced Developer's Guide.*

OAM – Operations and Maintenance

As the top subcomponent of the application server on the diagram, the OAM module handles communication with the System Control Panel (SCP) and the Metreos Visual Designer. As a developer, your applications may expose configuration values or manual actions that an administrator can manage through the SCP. These features arise through the OAM.

Using the OAM component, the MCE exposes a rich, well-defined management API that may be consumed using the Simple Object Access Protocol (SOAP). Using the OAM API management of the MCE can be easily integrated into other third-party network management tools. The OAM API is described in more detail in the document *Metreos Communications Environment: Advanced Developer's Guide*.

OAM also comprises the first stage of the Application Server's **application lifecycle pipeline**. When a developer or administrator deploys an application either from the System Control Panel or the Visual Designer, the application is first passed to the OAM. More information on the application lifecycle may be found in *Chapter The Application Lifecycle Pipeline.*

Assembler

The **assembler**, working in conjunction with the **core engine**, prepares new applications for use by the Metreos Application Server by distributing application components across the system. This includes separating out scripts, databases, media resources and the installer from each new application, and performing initial tests for correctness and resolving any versioning issues.

Once the application has been extracted and tested for data integrity it takes each script contained within the application and transforms the XML intermediate language into an in-memory executable. During this step the assembler checks the syntax of the intermediate language to ensure the script is well-formed.

The assembler then signals the core engine that the application is valid. The core engine then prepares any **application databases**. If an **application installer** is present, it is executed, setting up any configuration parameters for the application and setting default values as provided by the developer. Also, the core engine negotiates procurement and provisioning of any **media resources**, so applications which need to play audio through Media Servers will have assured access to these files. Finally, the assembler then passes on the largest and critical component of applications, **scripts**, to the script pool.

Script Pool

Application scripts reside within the **script pool** fully prepared for execution. The script pool is notified by the core engine when it is time to start executing a script. The core engine makes this determination by comparing incoming events from protocol providers with the triggering criteria defined for the installed scripts. Once it is informed that an event has come in for a particular script, the script pool passes a copy of the script to the **virtual machine**. This copy is called a **script instance.**

Virtual Machine

The virtual machine manages the running code of live applications. Within the constraints of system resources, any number of scripts, each with any number of instances, may be executed simultaneously. This ensures that multiple applications may run in parallel, and that multiple users may be accessing any one application at the same time.

The virtual machine also ensures system stability by segmenting applications from one another. This mechanism ensures that an unstable application cannot adversely affect other applications. Once the virtual machine determines that an application is in an unstable state, the application is unloaded and will not trigger again until the problem is resolved and the application is reinstalled.

Core Engine

At the center of the Metreos Application Server lies the **core engine**. All communication between components, as well as subcomponent control and resource management, occurs

within the core engine. In addition, this element manages all the other application components. Most of the architecture of the core engine deals with the intricacies of the Metreos Application Server, and therefore does not pertain to application development.

The Application Lifecycle Pipeline

While an understanding of the components of the Metreos Application Server will be useful to the developer of telephony applications, most developers will concern themselves with the movement of applications through the system. The **application lifecycle pipeline** enumerates each phases in the life of an application. Understanding this movement will be of great benefit to the IP telephony application developer.

Application Lifecycle Pipeline Overview

Similar to other development platforms, applications usually begin their existence in an integrated development environment (IDE). As an application evolves, developers will iteratively test the program by *compiling* and *executing* the application. The Metreos Communications Environment provides a similar structure, which consists of five distinct steps:

1. **Development –** Application is created in the Metreos Visual Designer development environment.
2. **Build –** Metreos Visual Studio prepares application for use by converting diagrams into a proprietary XML-based intermediate language**,** and then compiled code and additional application data (such as media resources, an installer, and any needed databases) are combined into a single .MCA archive file.
3. **Deployment –** The .MCA archive file is uploaded to the Metreos Application Server, either through the System Control Panel or via direct deployment in the Metreos Visual Designer.
4. **Installation** – The archive is unpackaged and resources are allocated to handle all components of the new application. Application scripts are copied into the Application Pool where they wait for events.
5. **Execution** – The Metreos Application Server processes the application generated from the user's initial design.

Naturally, each of these steps occur in different locations throughout the MCE platform, some even involving two or more components. *Figure 8* describes this relationship visually.

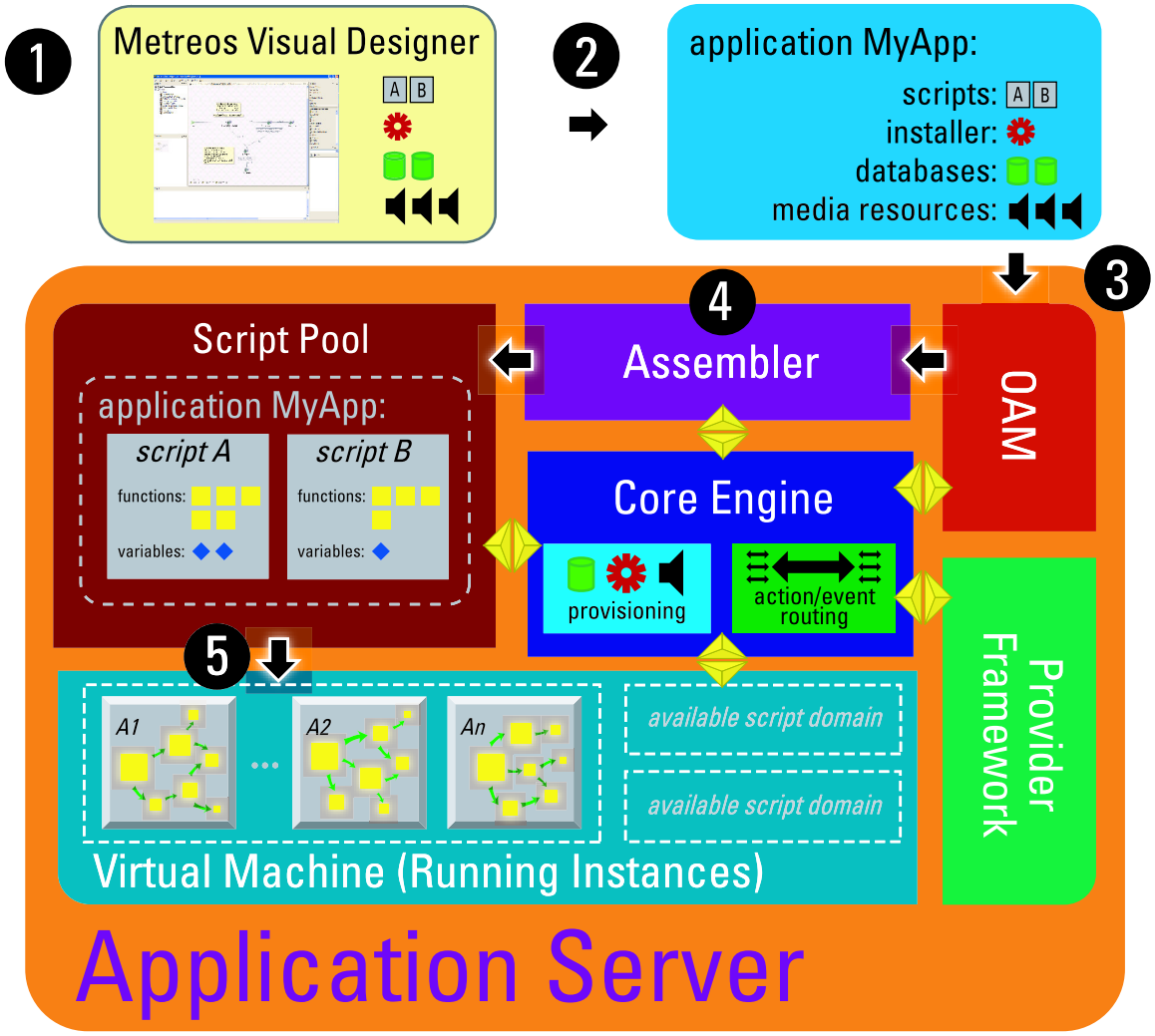

Figure 8: Application Lifecycle Pipeline

Step 1. Development Phase

An application begins its existence in the Metreos Visual Designer. It is at this time that the application logic takes shape. The MVD will assist the programmer by applying certain rules automatically to ensure that applications are well-formed. For example, if a developer drops an event on the canvas which generates callback events, the MVD will automatically generate functions for the developer to use when implementing those event handlers. This and other design features of the Metreos Visual Designer assist programmers in making applications which conform to the powerful architecture of the Metreos Communications Environment.

Step 2. Build Phase

Once the developer has completed the application, or completed enough application components to test some aspect of functionality, they instruct the Metreos Visual Designer to "build" (or compile) their program. This converts the representation of data and logical flow from a visual representation of boxes and arrows to a linear description in an XML-based proprietary intermediate language.

The compiler serves several purposes, but to the programmer the most valuable will be the ability of the compiler to catch mistakes made during development. Functions which lack proper exits, un-initialized variables, or unspecified execution paths will all cause errors to appear during compilation. Developers are encouraged to build their applications frequently to ease the debugging process.

After the application is successfully compiled into XML, all XML code and other application components are packaged into a single unified archive file. This process simplifies transporting applications and related data by rolling everything into a single file, including databases, media resources, installer information, the application, custom code, and related versioning data. These files retain the extension .MCA, for Metreos Communications Archive. Packaging may be accomplished through Metreos Visual Designer, or through the Metreos Packager command line tool "mca.exe". For more information on this latter method, see the document *Metreos Communications Environment: Advanced Developer's Guide.*

Packaging errors can occur when the additional resources such as installers, databases, custom code and media resources appear malformed or contain conflicting information. While the packager cannot catch all possible errors, many types of resource problems can become evident during this phase.

Step 3. Deployment Phase

Packaged archives are installed to the Metreos Application Server (MAS) either through the Metreos Visual Designer or the System Control Panel. The MAS receives the archive through the Operations and Maintenance (OAM) component, which handles all configuration and administrative communications. The OAM in turn routes the complete package to the Assembler, which unpacks the contents and routes individual components throughout the Metreos Application Server.

Step 4. Installation Phase

Installation unpacks the archive file and prepares its elements for provisioning throughout the system. Any databases in the package arrive in the Core Engine, which determines based on versioning data whether or not existing databases from previous installations will affect this install. Then, the Core Engine attempts to create or update the data and schemas defined by the new application. This process could return database creation errors which result from malformed SQL, or, resource allocation problems from overloaded application servers.

The installer for the new application behaves similarly to any included databases. Installers are routed to the Core Engine and immediately processed. Syntax errors or unsupported installer commands generate errors, which are then returned to the user through the OAM.

Media Resources are likewise passed along to the Core Engine, which in turn contacts the Media Servers associated with this Application Server. To ensure optimum performance,

all such media resources are installed on each Media Server. The Core Engine may report problems if the Media Servers lack sufficient resources to store the media files.

Upon successful assembly, the script pool registers each script with the Core Engine in order that appropriate incoming events can initiate script execution. This registration process ends the Installation Phase.

Step 5. Execution Phase

Each time an event fires whose event signature matches a script's triggering event signature, the corresponding script is copied to the Virtual Machine for execution. Any number of instances of the same script may be created, within configuration and resource limitations. Each instance maintains separate memory storage and distinct access to resources.

During execution, any runtime or logic errors which may exist will become apparent. If an application seems unresponsive or behaves unexpectedly, check the Application Server logs for information, or update your code to provide more detailed logging information. When a script instance finishes execution, relevant state information is reset and it returns to the Application Pool awaiting the next triggering event.

Language Architecture

Application Structure

Applications within the Metreos Communications environment contain a wide variety of elements, organized according to strict architectural requirements. This lets programmers organize their development efforts using the common principles of separation and abstraction. Furthermore, components within applications may be easily reused, further enabling rapid application development of converged IP communications services.

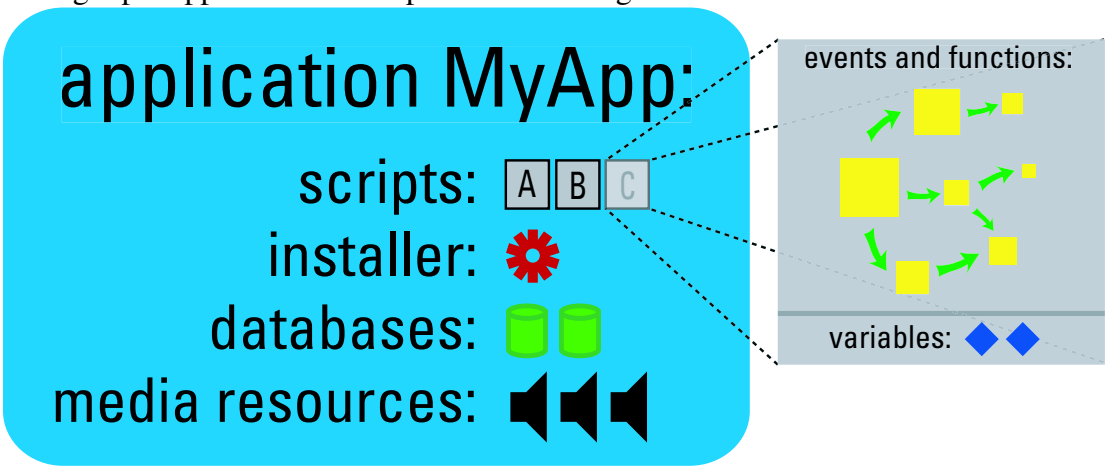

Figure 9: Application Structure

Applications within the Metreos Communications Environment are typically referenced by name. The application name provides basic descriptive information to users and developers of the nature and capabilities of the application service.

Each application consists of up to four types of components: **scripts**, an **installer**, **databases**, and **media resources.** The latter three component types are optional and may be included in any combination to meet individual application needs.

Scripts

All application logic either occurs within scripts, or is directed by scripts. Scripts bear a strong resemblance to threads in a traditional programming environment. Just as in other languages, scripts give programmers the ability to alter execution paths with control structures, store data in script-wide variables, access external resources, and accept input (albeit indirectly) from users.

Installers

As in any operating environment, applications must be installed before use. Application developers may wish to include special instructions to the Metreos Communications Environment regarding the desired setup and deployment of their services. This may include such requirements as dependencies on external protocol providers, configuration settings, and licensing restrictions.

An application can have a single installer. The installer is represented as an XML configuration file, which is automatically prepared by the Metreos Visual Designer upon request. For more information on installers, see the section below entitled "*The Installers and Configuration Parameters.*"

Databases

Telephony applications must often retain data beyond the execution of an individual script. Applications within the Metreos Communications Environment may include any number of associated SQL (Structured Query Language) databases which enable developers to store relational data directly in the application itself. Information stored within databases persists across any downtime periods or other connectivity issues which can occur, making them particularly useful for storing long-term data such as call records, access permissions and other general storage items.

The SQL engine embedded within the Metreos Communications Environment is designed for high-speed retrieval of data for use in high-performance telephony applications. If your application requires report generation, integration with existing databases, or other advanced database features, Metreos recommends connecting to remote data stores through an external ADO.NET data provider in addition to using the embedded database engine.

Media Resources

Most common examples of telephony applications include audio prompts such as pre-recorded greetings, error messages, and status messages. The Metreos Communications Environment lets developers bundle any necessary audio prompts with an application, easing the deployment process further.

Media Resources may be either .vox or .wav files. Any number of files may be included within an application. Such files should be 8 bit/8khz, to maximize performance.

The Action-Event Model

The structure of the Metreos Application Server relies upon a programming paradigm called the **action-event model.** In traditional development environments, programs begin executing a series of instructions once a user has issued the command for the program to start. In the MCE, applications respond to **events** and communicate through **actions**. Developers must fully understand this model to effectively develop IP telephony applications with the Metreos Communications Environment.

Events: Application Development through Reaction

The event-driven programming model has gained recent popularity in a wide variety of development environments, ranging from embedded devices to high-level graphical user interface programming. Instead of polling a resource over and over again for new data, the event architecture automatically executes code associated with the firing of an event.

While these applications certainly qualify as high-level, the structure of the MCE mimics the very low-level x86 interrupt mechanism. For those with experience writing low-level code, writing a program for the MCE may be distantly reminiscent of writing an interrupt-driven I/O handler. The interrupt is equivalent to an event, and actions are similar to output commands to the device which raised the interrupt. Although somewhat analogous in theory, in practice application development for the MCE is orders of magnitude less complex due to the inclusion of well-defined APIs and the high level of abstraction.

The Metreos Communications Environment monitors a variety of events which arise in the outside world. These include telephone calls, HTTP messages, and announcement status information, among others. When these occurrences manifest within the MCE, an event is generated. Examples of events include `CallControl.IncomingCall`, and `Providers.Http.GotRequest`.

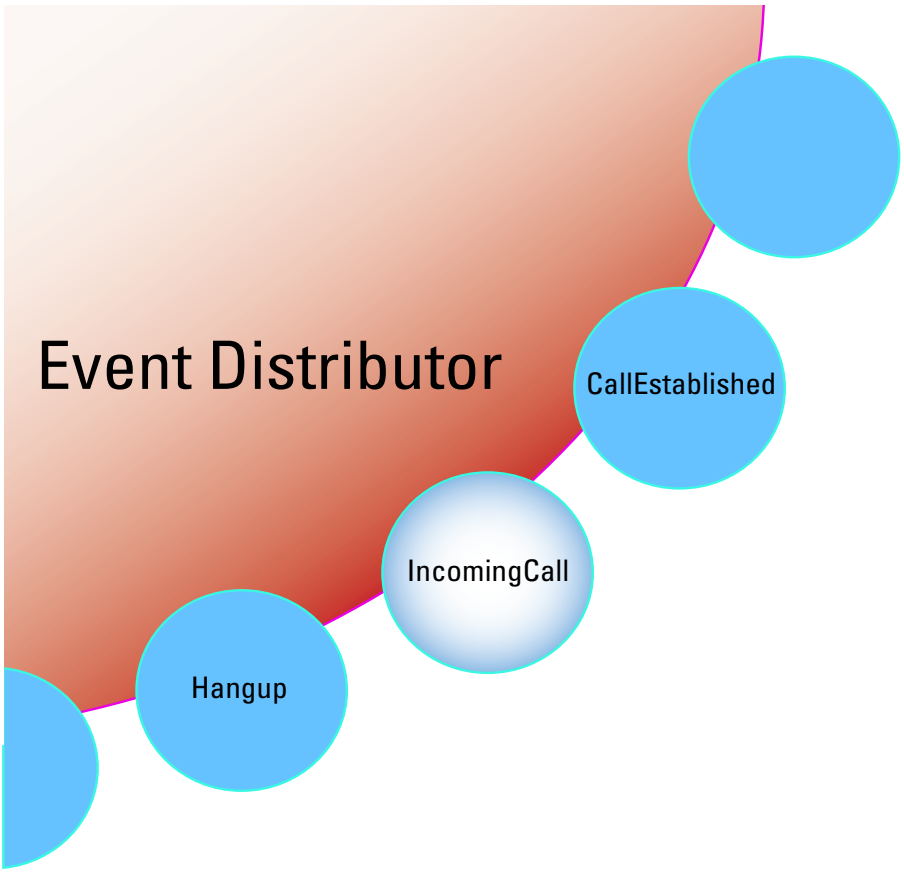

Figure 10: Event Distributor

Events originate from protocol providers and reach applications by way of the **event distributor**, as shown above in *Figure 10: Event Distributor*. Any number of events may fire simultaneously, and of course a single event may fire in rapid succession. Programs within the MCE typically respond to a small subset of the possible events which may be raised, but developers can select any combination of events based on business requirements.

In other development environments, designing applications which respond to a flood of requests presents a significant programming challenge. With the Metreos Communications Environment, however, new **script instances** are automatically created each time a triggering event occurs. Each script instance operates completely independently of other scripts, so script state and script-level variables are automatically preserved. For more information on these scoping requirements, see the section on *Scoping*.

Types of Events

The Metreos Communications Environment presents three distinct types of events to developers, with a fourth hybrid type for technical completeness. Although each event

type occurs under different circumstances, the development requirements are mostly identical for each class of event.

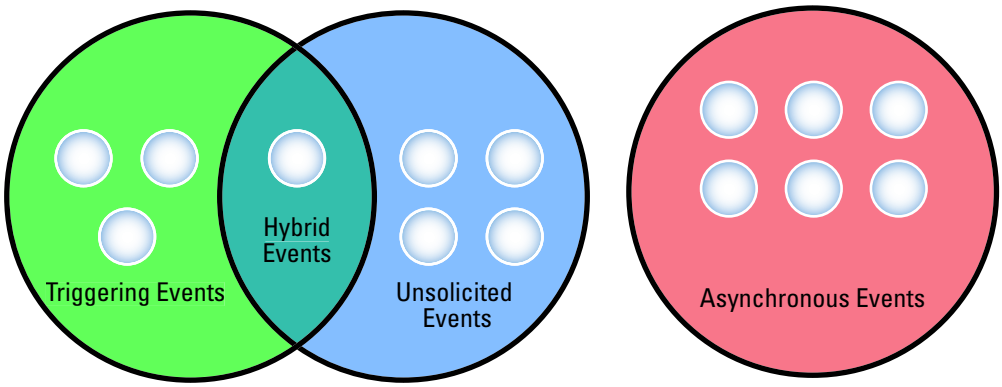

Figure 11: Types of Events

The three fundamental event types are **triggering events**, **unsolicited events,** and **asynchronous events.** Some events may be used as both triggering events and unsolicited events, even within the same application. These are called **hybrid events**. There are no asynchronous events which can occur as triggering or unsolicited events.

Event Signatures

Applications might easily be defined as *groups of scripts which respond to events.* Any programmatic task with the MCE must occur as a result of the firing of an event. Furthermore, code responding to an event can easily define an exact range of parameters of interest. For example, if a script should only begin when calls are placed to a particular set of phone numbers, the firing event can be interrogated to see if it matches the *signature* set of phone numbers needed by the script. .Net regular expressions may also be used to specify event signatures.

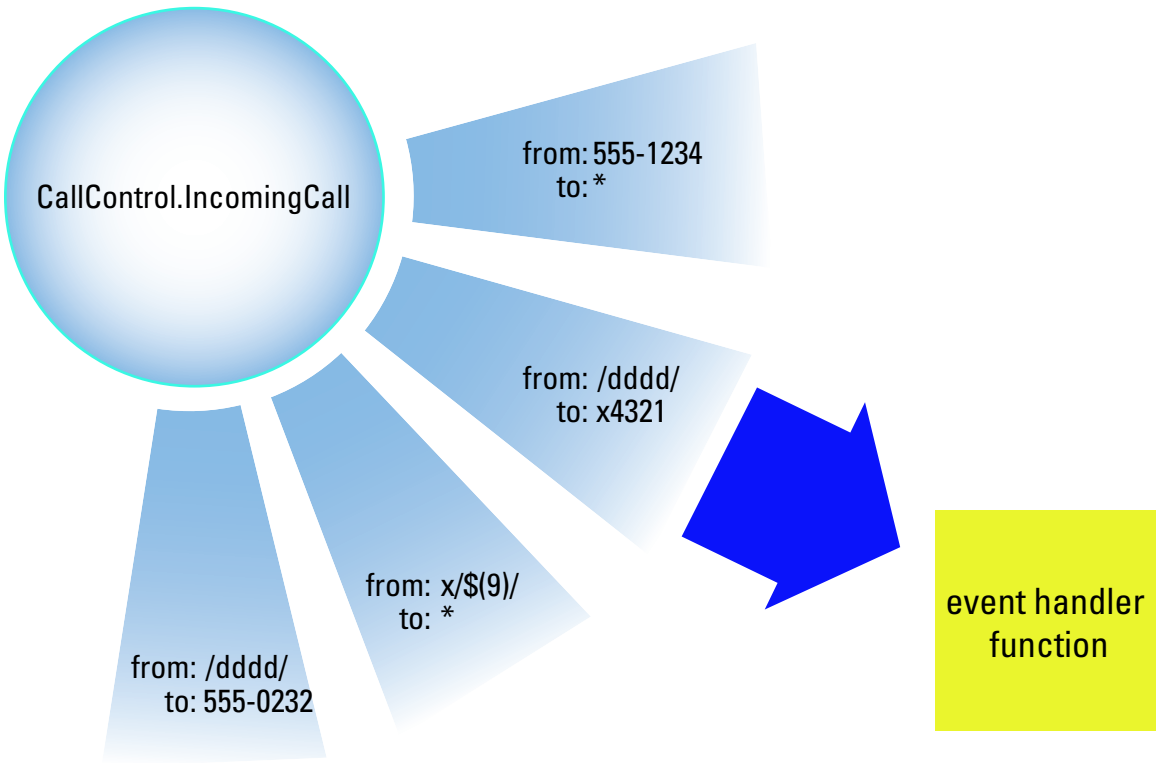

Figure 12: Event Signatures

The unique identification of particular event parameters used to initiate a function is called an **event signature**. Every time an event fires, functions associated with that event are invoked if and only if the corresponding event signature matches. These functions are called **event handlers**. This design serves two important purposes: First, it provides a programmatic way for event routing to occur within the architecture, rather than within applications. This enables the MCE to handle the load-balancing, failover, and performance requirements of a distributed deployment.

Second, the developer is freed from writing detailed analysis code to monitor all incoming events for the appropriate data and discard unwanted events. Developing software components to ignore uninteresting or invalid input is often tedious and error-prone.

Triggering Events

For every script within every application, one particular event serves the unique purpose of starting the script. This event is called the **triggering event**, because it triggers script execution. Each script has exactly one triggering event and thus, exactly one entry point. The signature of the triggering event defined by the application forms an exclusive relationship between event and event handler.

All applications installed on the MCE must declare triggering event signatures which are mutually exclusive. It is important to note that the MCE will not execute an application if it finds more than one application with the exact same triggering event

parameters. Therefore, it is vital that all applications declare triggering event parameters which guarantee that no other applications will trigger off the exact same event.

In the case of overlapping triggering event parameters, the application server will find the application whose triggering event signature is the closest match for the received parameters. For example, if MyApp1 triggers on `Http.GotRequest` with url = `"/myApp"` and MyApp2 triggers on `Http.GotRequest` with `url = "/myApp2"`, and the application server receives a request with url = `"/myApp2"`, the application server will correctly launch MyApp2.

Unsolicited Events

Unsolicited events can occur at any time during script execution. They generally indicate an action which was taken by an external user. For example, a script which triggers on `CallControl.IncomingCall` must be prepared to receive the `CallControl.Hangup` unsolicited event at any time.

However, script designers may anticipate multiple unsolicited events of the same type. For example, if a script places several calls then multiple `CallControl.Hangup` unsolicited events could occur. If each event needs to be handled differently, the developer will have to create branching code to properly interrogate each event signature and follow the necessary course of action. This is the principal use of the `userData` parameter in actions which generate unsolicited events. The MCE automatically propagates this information back to the event handler so developers can tag events using notation relevant to their application.

Asynchronous Callbacks

Asynchronous callbacks differ from unsolicited events in that they can only occur after a corresponding **action** has executed. For example, calling the action `MediaServer.PlayAnnouncement` will attempt to play an audio stream to a specified destination. If the media cannot be played for any reason, within a fraction of a second the `MediaServer.PlayAnnouncement_Failed` event will fire back to the same script which made the request. However, if the announcement was played successful, the `MediaServer.PlayAnnouncement_Complete` event will occur. These two callbacks will not occur before the `MediaServer.PlayAnnouncement` action has executed, and one or the other is guaranteed to occur if the action executed successfully. This differs from unsolicited events which may not ever occur depending on the circumstances.

Events and Protocol Provider Sessions

Another model for understanding events derives from the notion of a **protocol session**. This refers to a grouping of activities all related by one common element, often the same user or user agent. For example, a visit to a website might constitute a session with a clear beginning and end, and data exchanged throughout. Similarly, the collection of TCP/IP packets transmitted to download a single image on one page of that visit might also be considered a session.

Generally speaking, protocol providers within the Metreos Communications Environment present some concept of a session. The start of a new session is typically associated with a triggering event. Events which arise during the course of a session and are handled by applications are either unsolicited events or asynchronous callbacks.

Actions: Making things happen

The complimentary component to events, **actions,** are used to send data to the outside world or carry out specialized application logic. Actions bear a resemblance to method or API function calls in other programming languages.

Application scripts are constructed using conditional logic by linking together one or more **actions**. Actions can be thought of as individual commands that execute on behalf of the application script. There are two main categories of actions: **provider actions** and **native actions**. Understanding the differences between the two action categories is fundamental to understand how application scripts execute within the Metreos Application Server.

Provider Actions

Recall from the previous chapter that a Protocol Provider is a component within the MCE that facilitates communication between external systems and MCE applications. Protocol providers execute within their own virtual process space and are entirely separated from the virtual machine in which application scripts execute.

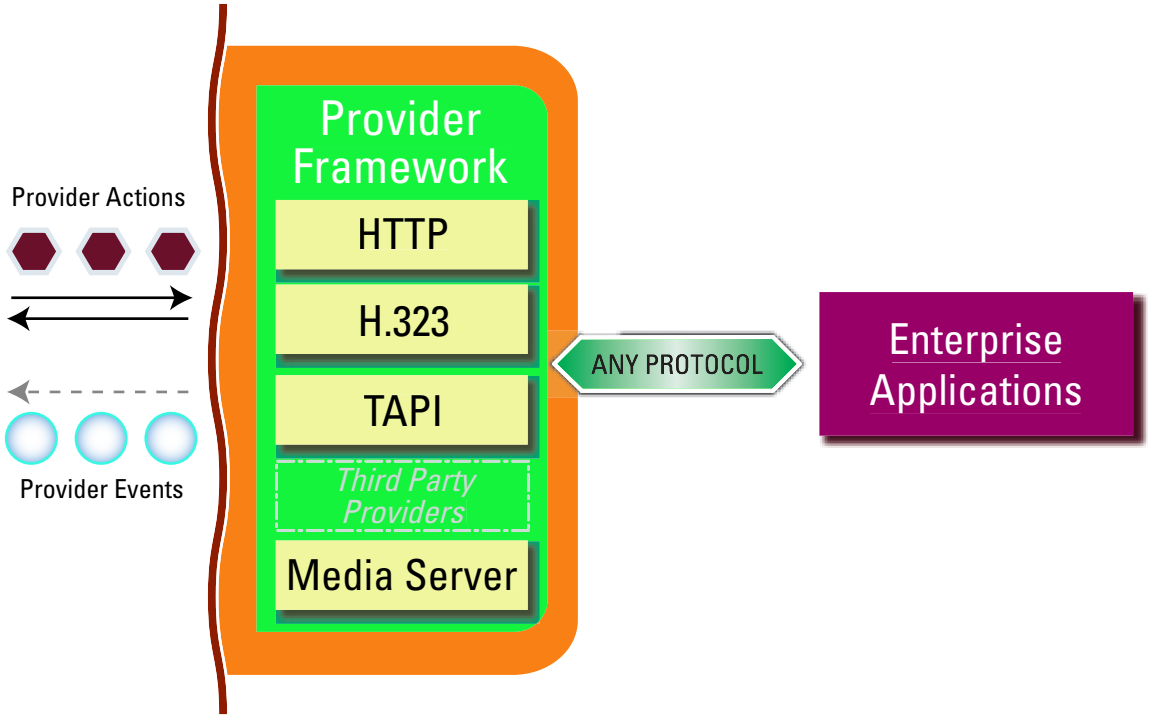

Figure 133: Protocol Provider Framework

As depicted in *Figure 133*, **provider actions** flow out of an application into the provider framework, and **provider events** bring messages from providers back to applications.

Information flows in both directions when a provider action is called by an application script, but only in one direction (from provider to application) when a provider event occurs.

Just like making an API call in a traditional language, calling a provider action blocks script execution until a response is returned. This is indicated by the two execution arrows directly below the provider actions in the diagram. Simple actions which only involve this execution path are called **synchronous actions**. However, some calls to a provider action may only be able to offer a provisional response. In this case, the final response occurs later in the form of asynchronous event. This class of action is called an **asynchronous action**.

Synchronous actions represent a request-response transaction within the application server, as shown in Figure *144*. First the action is called from a particular location within the script. This is called the **request**. When the provider is finished processing the message or an error occurs, an action **response** message is sent back to the specific script instance that sent the original request, and script processing continues.

Figure 144: Synchronous Provider Action

From the time when 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 works well for the majority of actions because the time it takes to process the request is typically very short; however, for some types of actions an asynchronous model is more appropriate.

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 5)* Consider as an example an application script which places a call to some location. The application server does not wait for the call to be answered since it could take an unknown amount of time; instead, the act of answering the call fires as an asynchronous event elsewhere in the application.

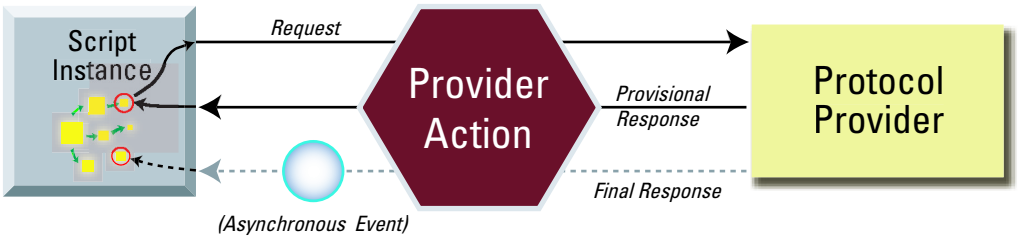

Figure 15: Asynchronous Provider Action

The provisional response indicates that the provider is processing the request but that it may take some indeterminate amount of time. Once the provisional response is received the script instance continues executing normally. Using the example presented above, when a call is placed the provisional response indicates that the initial call request was successful and that the call is proceeding.

When the provider is finished executing the asynchronous action request it constructs an **asynchronous event,** or **callback event,** and sends it to the script instance. The receipt of an asynchronous callback event indicates that the action transaction is complete. Asynchronous callbacks are a special type of unsolicited event in the Metreos Application Server. They contain specific information that allows the virtual machine to map the event to the appropriate script instance and event handler.

In this sense, the Metreos Communications Environment retains application state automatically, by ensuring that actions may call back the scripts from which they were initiated. Asynchronous callback events eliminate the need for polling systems and keep resource utilization to a minimum. Just like a human user, who answers a call as quickly as it arrives, the Metreos Communications Environment can respond to external events as they arise.

Native Actions

The Metreos Application Server provides two methods for extending functionality: the provider protocol framework and support for **native actions**. Native actions essentially allow developers to add custom logic that will execute within the script instance's virtual process space. Like inline assembly in languages like C/C++, native actions provide a technique for executing operations that should reside within the IP telephony application, but are better suited to traditional programming paradigms.

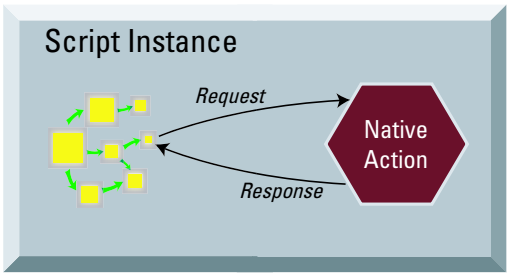

Figure 156: Native Action

A native action by its very nature is always synchronous. They are best used for building logic into an MCE application that does not need to maintain state beyond the lifetime of the script instance and does not need to monitor any external network service. Unlike provider actions, native actions actually cause the virtual machine to execute the action logic within the context script instance. A diagram of the native action execution model is shown in *Figure 156*.

Core Actions

There are some actions that deviate slightly from the model described above. These are referred to as **core actions**. These actions are handled internally by the Metreos Application Server's runtime environment. All of the actions within the `Metreos.ApplicationControl` namespace are categorized as core actions.

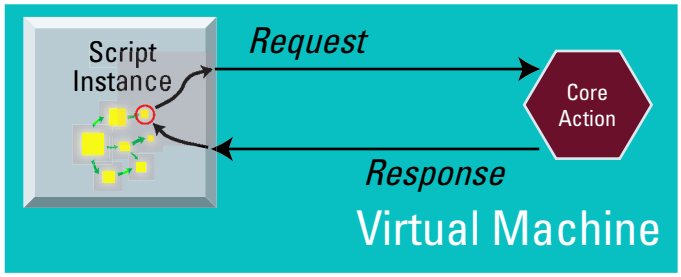

Figure 167: Core Action

The core actions include services such as calling functions, exiting the current function, and ending the application. All core actions are synchronous actions.

Scoping

A considerable design choice in almost every development environment involves the nature of scope. Scoping refers to the accessibility of variables and code segments from within different functional blocks. The Metreos Communications Environment implements a strong scoping mechanism designed to encourage a consistent yet flexible style of coding. Scoping is best understood by reviewing it's 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 has different scoping constraints:

Scripts - All scripts within an application are independent and invisible to one another. There is no way to execute code, access variables, or interrogate the elements of one script from another. Communication between scripts must occur through a database or through an external messaging component such as protocol provider.

The Installer - The configuration settings defined by the installer are globally available from anywhere with the application, but cannot be changed by the application itself. Instead, using the System Control Panel, an administrator can modify configuration settings for an application.

Databases – Every database may be accessed from any script within the application. This permits scripts to communicate with one another, or to share data which will persist even after all script instances have ended.

Media Resources – All media resources may be accessed and played through any script within the application. Multiple scripts may play the same media resource at the same or overlapping times.

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 may be accessed by any function within the same script, but not by other scripts. Functions which need to communicate with one another should use script variables.

Function Level Scope

Functions comprise the lowest level of the Metreos Communications Environment, and thus have access to all other levels above them. Functions may also contain variables which can be set or modified by function elements, but not by other functions, even those within the same script. Function-to-function communication should be performed by using script-level variables, or function parameters and return values.

Application Resources

The Installers and Configuration Parameters

Applications often need systems administrators to set configuration information specific to a particular deployment. Through application installers the MCE enable developers to define the configuration parameters, their types and default values.

An application installer is an XML file similar to the one in Listing 1 below.

```
<?xml version="1.0" encoding="utf-8" ?>
<install xmlns="http://metreos.com/AppInstaller.xsd">
   <configuration>
      <configValue name="CM_Address" description="CallManager address"
                   format="string" diagnostic="false"
                   requiresRestart="true"
                   readOnly="false">192.168.1.250</configValue>

      <configValue name="DialPlan" description="A sample hashtable"
                   format="hashtable" diagnostic="false"
                   requiresRestart="true" readOnly="false" />

      <configValue name="CM_LDAP_Port"
                   description="LDAP port"
                   format="integer" diagnostic="false"
                   requiresRestart="true"
                   readOnly="false">8404</configValue>
   </configuration>
</install>
```

Listing 1: A sample installer.

Application installers contain one or more `configValue` entries that describe the various configurable elements of an application. A configuration entry can specify a default value for string, integer, and boolean types. For example, from the sample listing above the default value for the "CM_Address" configuration entry is "192.168.1.250". You can also specify entries of types hashtable and stringdictionary. Entries of this type cannot have a default value. Key-Value pairs for these entries must be specified directly through the System Control Panel.

Application installers are processed during the installation phase of the application lifecycle pipeline. The Metreos Application Server creates corresponding database entries for each of the configuration values in the installer and links those entries to the application being installed. After this is complete, all of the configuration values specified in the installer are visible and editable through the web via the System Control Panel.

Figure 178 shows a graphical representation of the application installer XML schema:

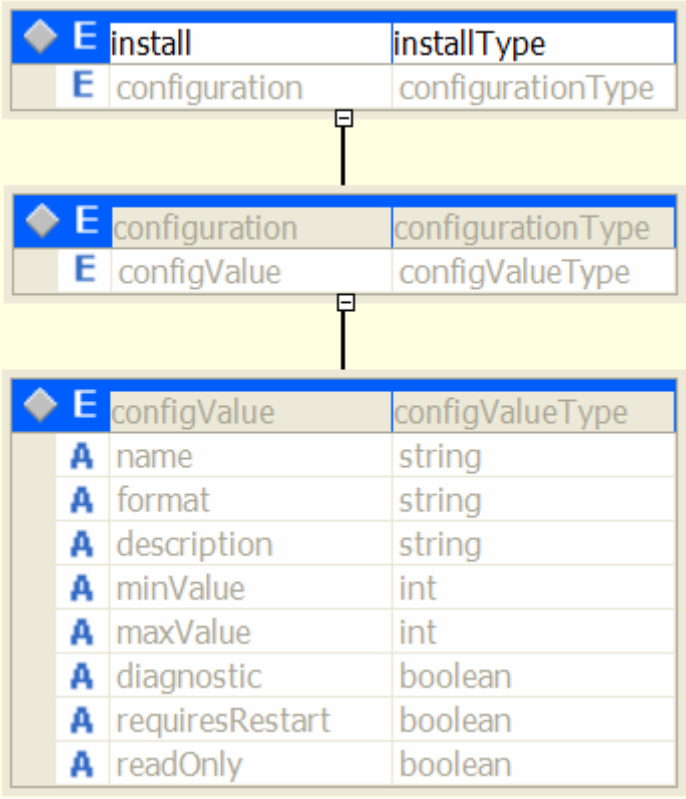

Figure 178: The Application Installer XML Schema

Each configuration entry must specify the `name` and `format` attributes. All others are optional. The `format` attribute indicates the type of configuration value to expect. Valid values for the `format` attribute are: `string`, `integer`, `hashtable`, stringdictionary, and `boolean`.

Database Management

The MCE ships with an embedded SQL database engine that developers may take advantage of when building applications on top of the platform. All basic SQL-92 constructs are supported by the embedded database; however, functionality such as stored procedures is not supported. To utilize database functionality, a schema creation script must be included in the application package. More information on application databases can be found in the document *Metreos Communications Environment: Advanced Developer's Guide.*

More information on the database functionality provided by the MCE can be found in the document *Metreos Communications Environment: Advanced Developer's Guide.*

4. INSIDE MCE APPLICATIONS

Applications within the Metreos Communications Environment do not resemble most other programs. Instead of long blocks of program code, developers see applications as a collection of webs, actions and custom features. Internally, however, applications use a streamlined yet flexible implementation to enable powerful, high-performance telephony services.

Application Structure

As described in the previous chapter, applications consist of four main components: databases, an installer, scripts and media resources. Scripts, in turn, contain global variables and functions, some of which are a special class of function called an event handler.

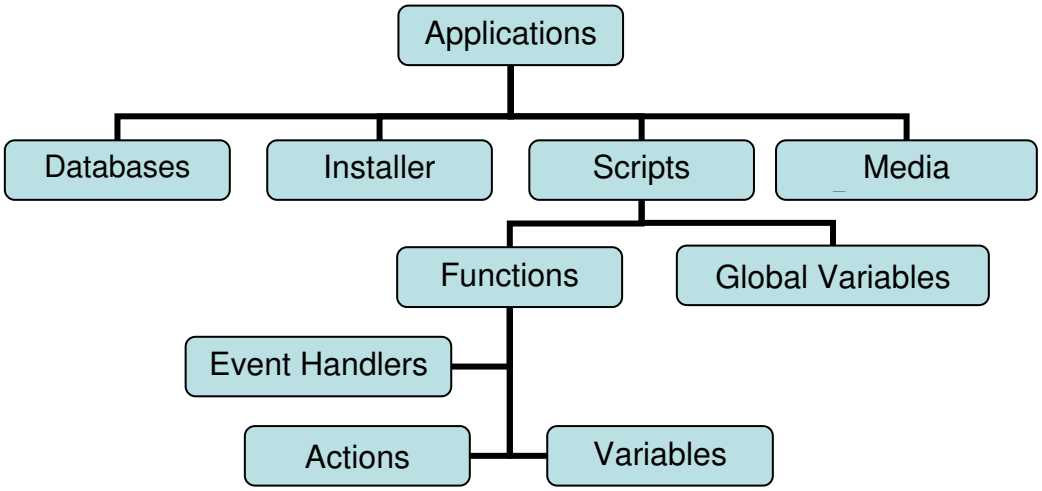

Figure 18: Application Structure

XML-based Implementation

The core language of MCE applications derives from an eXtensible Markup Language (XML). Much in the same way programs like Macromedia Dreamweaver™ and Microsoft FrontPage™ provide a simple interface which generates HTML, the Metreos Visual Designer offers a drag-and-drop application canvas, and compiles nodes and connecting arrows into XML formatted according to Metreos specifications.

However, there are few other similarities between the Metreos Visual Designer and HTML generators. Tools like Dreamweaver™ use a WYSIWYG (what-you-see-is-what-you-get) model for content generation. The Metreos Visual Designer (MVD) canvas displays a conceptual and logical diagram of data flow through a telephony application.

Furthermore, most of the elements apparent on the application canvas of the MVD correspond in a one-to-one relationship with XML elements. It is relatively easy to

compare a particular visual design for an application, consisting of actions, events and routing options, to the compiled XML version of the same application. Using the MVD makes application development simple, but an understanding of the XML behind applications will aid developers in gaining an understanding of the Metreos platform.

Typically WYSIWYG web designers will generate the final product on the fly allowing the developer to switch between a design view and HTML view. The MVD does not utilize this concept. Instead, the MVD stores information locally that contains all of the necessary data to not only generate the application XML, but also to render the application properly on the screen for development. When the MVD compiles the application from the local source code it generates XML formatted according to the Metreos Application Script XML schema. This XML document can be thought of as the **intermediate code** used by the Metreos Application Server for execution. Listing 2 shows the intermediate code for a simple "Hello World" application script for the MCE.

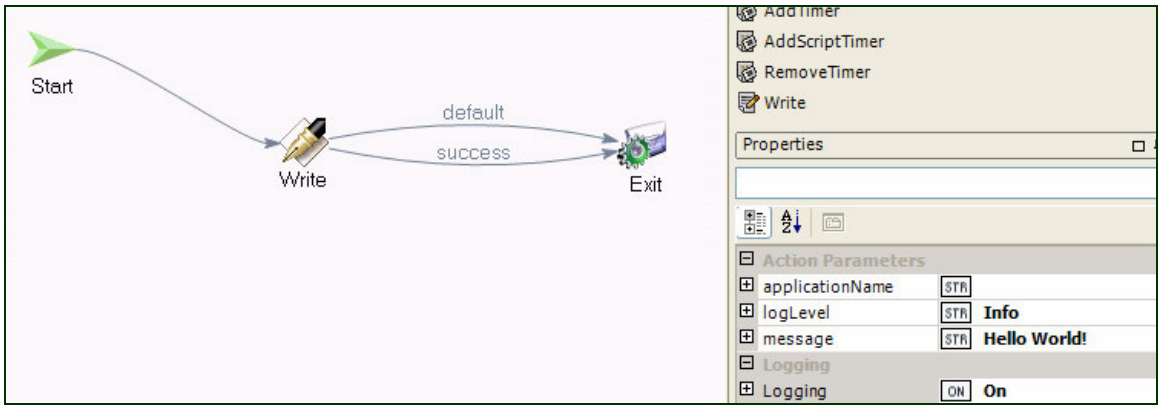

Figure 190: "Hello World" Application Script Screenshot.

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

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

   <trigger>
      <triggerEvent>Metreos.CallControl.IncomingCall</triggerEvent>
      <triggerParam name="to">joe@host.com</triggerParam>
      <triggerHandler>PrintLogMessage</triggerHandler>
   </trigger>

   <function id="PrintLogMessage">
      <variable type="string">myLocalVariable</variable>

      <action id="1" type="native">
         <actionName>Metreos.Native.Log.Write</actionName>
         <actionParam name="message">Hello World!</actionParam>
         <actionParam name="logLevel">Info</actionParam>
         <nextAction returnValue="success">2</nextAction>
         <nextAction returnValue="default">2</nextAction>
      </action>

      <action id="2">
         <actionName>Metreos.ApplicationControl.Exit</actionName>
      </action>
   </function>
</serviceApp>
```

Listing 2: "Hello World" Application Script Intermediate Code

Incidentally, there is no advantage to developing application scripts using the raw intermediate code XML. Since the Metreos Visual Designer simply translates on-screen elements into code, any improvements in efficiency, reliability, or performance could be as easily—if not more easily—found and implemented using the MVD. However, the schema of the XML is readily available in the *Metreos Communications Environment: Advanced Developer's Guide.*

Finally, application storage naturally retains the wide array of benefits provided by XML. Applications may be easily stored on any modern platform, transported across virtually any network infrastructure, and easily parsed due to a well-defined schema. Critics of XML argue that the technology requires significant overhead, which wastes resources and increases the margin for error. The Metreos Communications Environment addresses these issues through the use of an application assembler, which converts bulky XML into streamlined object code, while checking for correctness and rejecting malformed applications.

Execution Model

The basic building block of an MCE application is an **application script,** or simply "script". Each script represents a potential thread of execution for the MCE application.

Scripts begin when the application server receives a **triggering event,** which contains parameters that match the triggering criteria of an application script. When this occurs, a new **script instance** of that script is created and executed.

One might compare a script to an executable file residing on disk, waiting to be started. The triggering event is the equivalent of double-clicking the executable, and the in-memory process that beings execution as a result corresponds to the script instance.

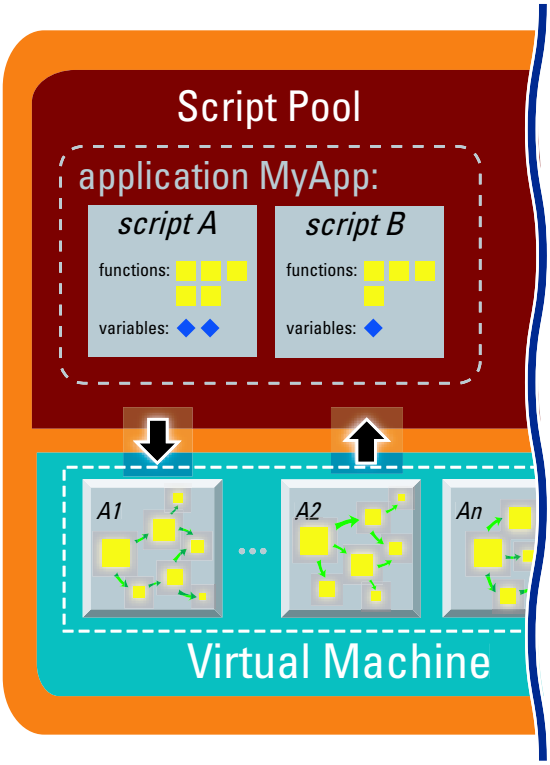

Figure 201: Script Execution and Completion

New script instances are executed within the Metreos Application Server's virtual machine, and each script instance executes in a separate thread. *Figure 201:* Script Execution and Completion shows a segment of the Metreos Application Server architecture. An application called `myApp` contains two scripts, `A` and `B`. Each time a triggering event is received, the MCE places a new instance of script `A` into the virtual machine. Whenever a script instance exits, it is removed from the virtual machine. The difference between a **script** and a **script instance** forms an important conceptual framework for developing IP telephony applications with the Metreos Communications Environment, because designers must account for concurrent instances of the same script.

Application Script Elements

Script processing occurs inside units of code called **functions**. Like many programming languages, all logic occurs within functions (or is controlled by functions), but variables, configuration and other initialization parameters may exist outside of the functions.

While a script may contain many functions, exactly one of those functions serves as the **event handler** for the application script's triggering event. Recall that a new script instance is started each time a triggering event is received. Once received, the triggering event handler function is started and marks the beginning of execution for the new script instance.

Application Script Triggers

As described in the previous chapter, a triggering event is an unsolicited event whose signature corresponds to a script residing in the script pool. Triggers should be thought of as a combination of the type of event and specific event parameters required to start a new instance of an application script type. A trigger must be unique across the entire Metreos Application Server—no two application scripts can have the same triggering criteria.

The trigger used to start the "Hello World" application from the example above is illustrated in *Figure 212* and *Listing 3:* Application Script Trigger Intermediate Code. As shown, a new script instance will begin when an event of type `Metreos.CallControl.IncomingCall` is received and contains a parameter named `to` with a value of "joe@host.com". If the event had a different value for the `to` parameter, then a new script instance would not start.

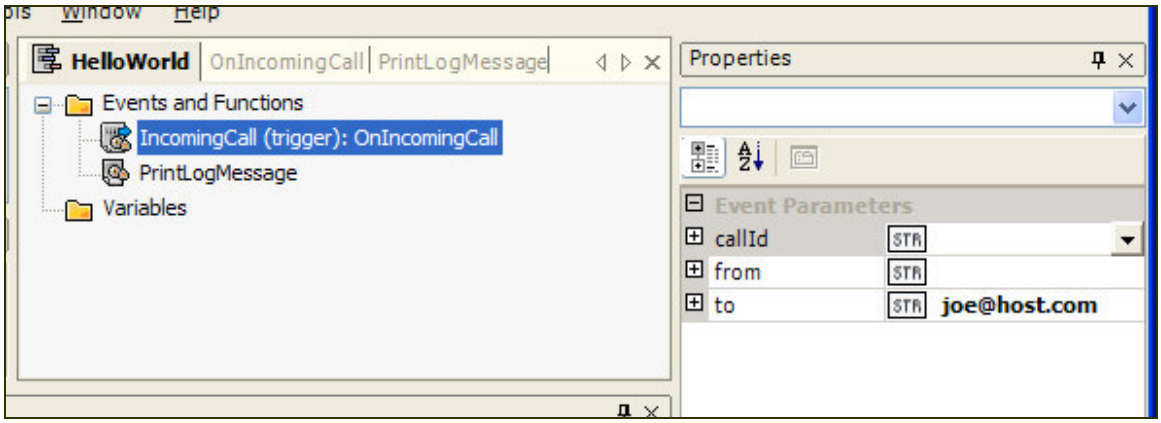

Figure 212: Application Script Trigger Screenshot

```
<trigger>
   <triggerEvent>Metreos.CallControl.IncomingCall</triggerEvent>
   <triggerParam name="to">joe@host.com</triggerParam>
   <triggerHandler>PrintLogMessage</triggerHandler>
</trigger>
```

Listing 3: Application Script Trigger Intermediate Code

Therefore, multiple script types may trigger on the same event type and event the same event parameters; however, the matching criteria for those event parameters must be unique. This is known as the **triggering event signature**.

The triggering event signature must contain at least an event type. It is recommended but not required that the application developer also specify at least one additional parameter to trigger on. Finally, triggering event declarations in the intermediate code must always include the name of the function that should be called to handle the triggering event.

Functions

Like the majority of programming languages that exist today, the MCE allows developers to group application script logic into functions. The Metreos Visual Designer allows application developers to utilize two types of functions: **event handling** and **standalone**. Listing 4 below shows how a standard, standalone function declaration is represented in the intermediate XML code generated by the MVD.

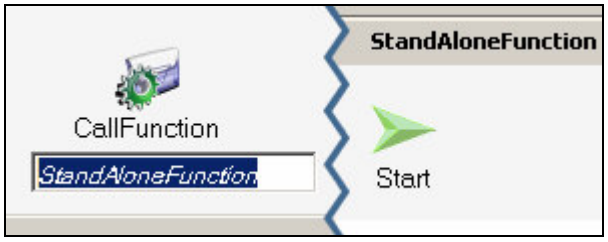

Figure 223: A stand-alone function screenshot from the MVD.

```
<function id="StandaloneFunction">
   <!-- Function Body Removed -->
</function>
```

Listing 4: A typical function declaration.

Event handling functions are only executed in response to an event that is received by the script instance. Listing 5 below shows the declaration of a function that will handle the `Metreos.Providers.Http.GotRequest` event.

```
<function id="HandleHttpRequest">
   <eventHandler>Metreos.Providers.Http.GotRequest</eventHandler>
   <!-- Function Body Removed -->
</function>
```

Listing 5: An event handling function declaration.

The presence of the `<eventHandler>` tag indicates to the virtual machine that the function should start when an HTTP `GotRequest` event is received by the script instance. It is important to note that other `GotRequest` events may be received by the application server and unless they are specifically destined for the script instance they will be treated as potential triggering events. In other words, event handling functions are specifically for handling unsolicited events for currently executing script instances.

Much like triggering event signatures, a function also contains a signature that must be present in the input parameters passed to the function when it is started. Function parameters behave the same when used in both standalone and event handling functions.

To indicate that a function takes an input parameter, a `<variable>` element must be added and that element must include an `initWith` parameter as shown in Listing 6.

```
<function id="FunctionWithInputParameters">
   <variable type="Metreos.Types.String"
             initWith="someParameter">variableName</variable>
   <!-- Function Body Removed -->
</function>
```

Listing 6: A standalone function with one input parameter.

A function may have as many input parameters as the application developer chooses to add. If the function is marked as an event handler and also includes input parameters, then the `<eventParam>` elements must be included to indicate the matching criteria for the event handler as shown in Listing 7. The function name, and optionally, its parameters and type of event that it may handle, constitute the **function signature**.

Function signatures, much like the triggering event signature, must be unique. The primary difference between the two is scoping. Triggering event signatures must be unique across the entire system. In other words each application script type must have a different triggering event signature. In contrast, function signatures must be unique within the application script itself.

For example, given the event handling functions shown in Listing 7 it is apparent that both functions handle the HTTP `GotRequest` event. Furthermore, both functions declare an event parameter that must match against the `url` field in the incoming event. The first function, `HttpEventHandlerA` will be executed when a `GotRequest` event is received that was destined for "/some/url". If the request was sent to "/another/url" then the second event handling function, `HttpEventHandlerB`, will be executed. If neither function matches then the event will not be handled and will be silently ignored. The Metreos Application Server will pick the best match for the specified parameters.

```
<function id="HttpEventHandlerA">
   <eventHandler>Metreos.Providers.Http.GotRequest</eventHandler>
   <eventParam name="url">/some/url</eventParam>
   <variable type="Metreos.Types.String"
             initWith="url">requestUrl</variable>
   <!-- Function Body Removed -->
</function>

<function id="HttpEventHandlerB">
   <eventHandler>Metreos.Providers.Http.GotRequest</eventHandler>
   <eventParam name="url">/another/url</eventParam>
   <variable type="Metreos.Types.String"
             initWith="url">requestUrl</variable>
   <!-- Function Body Removed -->
</function>
```

Listing 7: Two event handlers with differing signatures.

Variables

Variables allow application developers to store and manipulate data within their application scripts. Variables may either be initialized with a default value or initialized using incoming event parameters. Furthermore, variables may be globally or locally scoped. A **global variable** is visible and useable in all functions, whereas a **local variable** is visible and usable only within the function where it is defined.

```
<?xml version="1.0" encoding="utf-8" ?>
<serviceApp name="Conference Application"
   xmlns="http://metreos.com/ServiceApp.xsd"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

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

   <!-- Trigger Removed -->

   <function id="PrintLogMessage">
      <variable type="string">myLocalVariable</variable>
      <!-- Function Body Removed -->
   </function>
</serviceApp>
```

Listing 8: Global and local variables.

Listing 8 above shows how global and local variables are represented in the application script XML intermediate language.

Global and local variables act differently when they contain `initWith` attributes. When global variables are initialized using `initWith` the MAS attempts to retrieve configuration values from the configuration database to store in the global variable. For example take the following global variable declaration:

```
<serviceApp>
   <variable type="Metreos.Types.String"
             initWith="SomeConfigEntry">myGlobalVariable</variable>
</serviceApp>
```

Listing 9: Global variable with initialization.

When a new script instance of this script type starts execution the application server will attempt to initialize `myGlobalVariable` with the value of the `SomeConfigEntry` configuration parameter. If it is unable to do so, the script instance will not start.

If instead of declaring a global variable Listing 9 had declared a local variable within a function, the application server would attempt to initialize it with a value from the incoming event message. The message would be searched for a parameter named `SomeConfigEntry`, and if found, its value would be placed into `myGlobalVariable`.

Actions

The meat of all application scripts is the conditional logic constructed by linking together **actions**. Actions were previously discussed in context of the Metreos Application

Server's execution model. In this section we will explore how actions are represented in the applications intermediate XML code.

```
<function id="PrintLogMessage">
   <action id="1" type="native">
      <actionName>Metreos.Native.Log.Write</actionName>
      <actionParam name="message">Hello World!</actionParam>
      <actionParam name="logLevel">Info</actionParam>
      <nextAction returnValue="success">2</nextAction>
      <nextAction returnValue="default">3</nextAction>
   </action>

   <action id="2" type="native">
      <actionName> Metreos.Native.Log.Write </actionName>
      <actionParam name="message">Hello World, Again!</actionParam>
      <actionParam name="logLevel">Error</actionParam>
      <nextAction returnValue="default">3</nextAction>
   </action>

   <action id="3">
      <actionName>Metreos.ApplicationControl.Exit</actionName>
   </action>
</function>
```

Listing 10: An action map for a simple function.

Listing 10 above shows how a simple action map would be represented in the application script's intermediate XML code. Each action is comprised of two mandatory elements: `actionName` and `nextAction`. Each action may also define zero or more `actionParam` elements.

The flow of the application script is defined by how the actions link to one another, as represented by the `nextAction` elements. Each nextAction link contains a `returnValue` attribute which indicates which path to take, based on what the result of the action is.

Script Instance State Machine

When script instances are executed inside the virtual machine they have a very specific state machine in which they move through. Figure 24 is a state machine diagram that models how application script instances behave when executing.

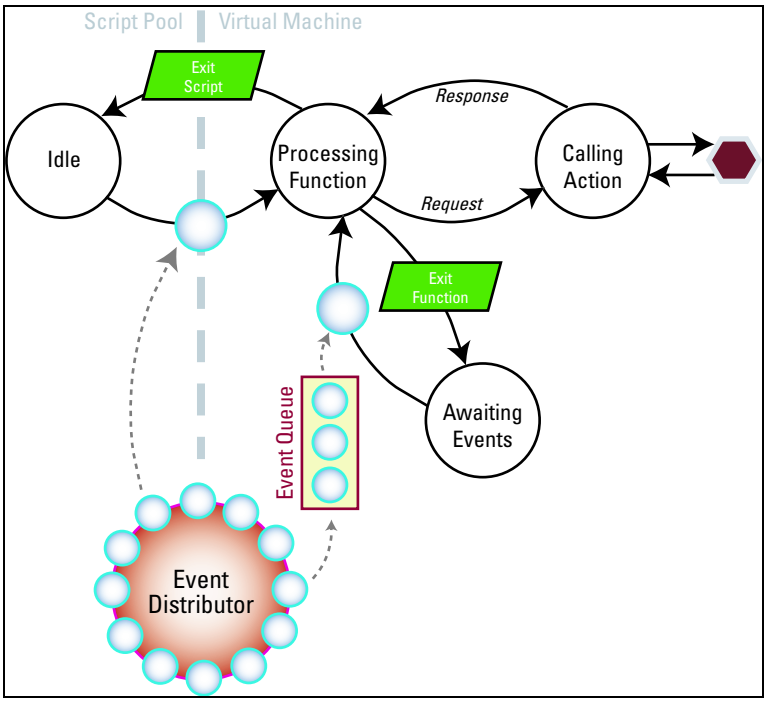

Figure 234: The state machine for application script instances.

Script instances can handle at most a single event at any given time. Therefore, when an event is received and processed by the script, nothing should be done inside the function handling the event to prolong the processing any longer than necessary. Any event destined for the script that is received while processing of a previous event is in progress will be queued up and handled in the order in which it was received.

When a script instance is in the **idle** state it is waiting to be started and resides in the script pool. Once a triggering event that matches that script type is received, the instance is moved from the script pool to the virtual machine and enters the **processing function** state. When inside this state, events are queued up until the script instance executes an `EndFunction` action, placing it in the **awaiting events** state.

5. DEVELOPING WITH METREOS VISUAL DESIGNER

Most of the history of telephony application development required hours of tedious planning, testing and development with cumbersome text-only tools. The Metreos Visual Designer almost completely eliminates the need for a keyboard when creating apps, giving developers the ability to draw applications in much the same manner of assembling a Microsoft PowerPoint™ slide or Microsoft Visio™ diagram.

The graphical interface of the Metreos Visual Designer primarily consists of the application canvas, akin to the code editor window of traditional integrated development environments. Elements of the telephony application appear as "nodes" in a web-like structure, with routing arrows indicating the flow of program execution based on easily defined conditions.

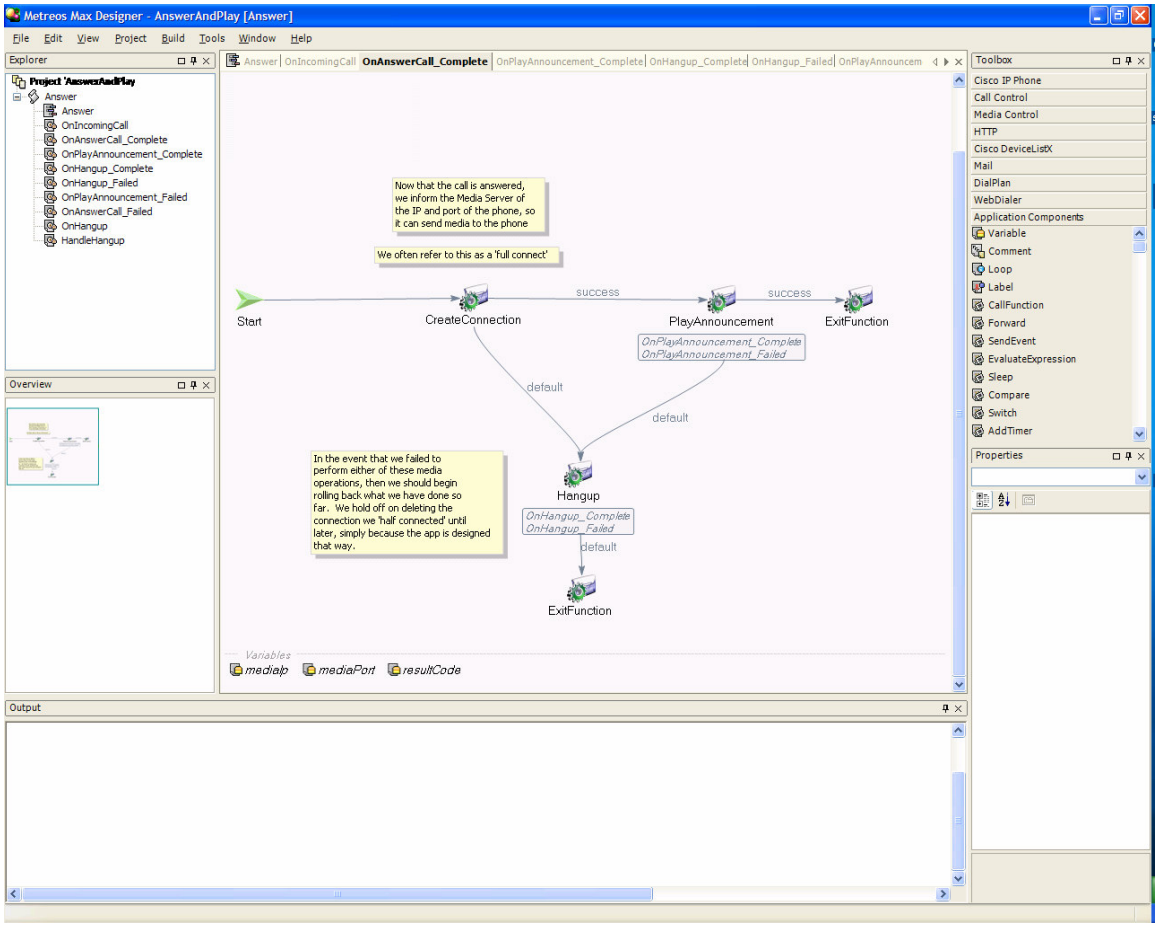

Figure 25: The Metreos Visual Designer

These nodes, which represent either actions called by the application to achieve a desired affect or powerful programmatic control structures, are simply dragged from the toolbar onto the application canvas. The settings of each node appear in the properties grid.

The size of an application poses no limit for the Metreos Visual Designer, which supports scaling and navigation of the application canvas through an overview window, as well as dividing applications into any number of separate functions, each stored on a different tab within the canvas.

The Metreos Visual Designer conveniently manages the applications and developer-defined support files as a single project. With just a few mouse clicks, a programmer may instantly compile, package and install their application to a running MCE system. Packaged applications persist as a single file, and can be easily transported or shared among a community of users.

The Metreos Visual Designer (MVD) enables the competent developer to produce in a few hours what once required weeks or months of intense planning and development. However, telephony application development is still complex, and experience writing software will benefit the programmer considerably, as well as thorough understanding of the structure of both the entire Metreos platform as well as the MVD. This document aims to satisfy that requirement for the curious and devoted telephony application developer.

Getting Started with the MCE

Developing applications for the Metreos Communications Environment (MCE) does not strictly require immediate access to a running installation of the MCE, but it is difficult to ensure that designs fulfill expectations without the ability to perform live testing. For information on purchasing the Metreos Communications Environment for your organization, please contact the Metreos sales division at www.metreos.com.

Once you have obtained an MCE system, installation requires little time or technical expertise. However, to ensure proper setup and facilitate ongoing administration, please refer to the document *Metreos Communications Environment: Administrator's Guide.*
Finally, developers will need to record the IP address and listening port of the Metreos Application Server in their environment. This information will properly configure the Metreos Visual Designer for application installation. Contact your system administrator for this information

Metreos Visual Designer Tour

The Metreos Visual Designer serves as the integrated development environment for telephony applications within the Metreos Communications Environment. Almost all development will occur within this program, so familiarity with each interface element will benefit the programmer immensely.

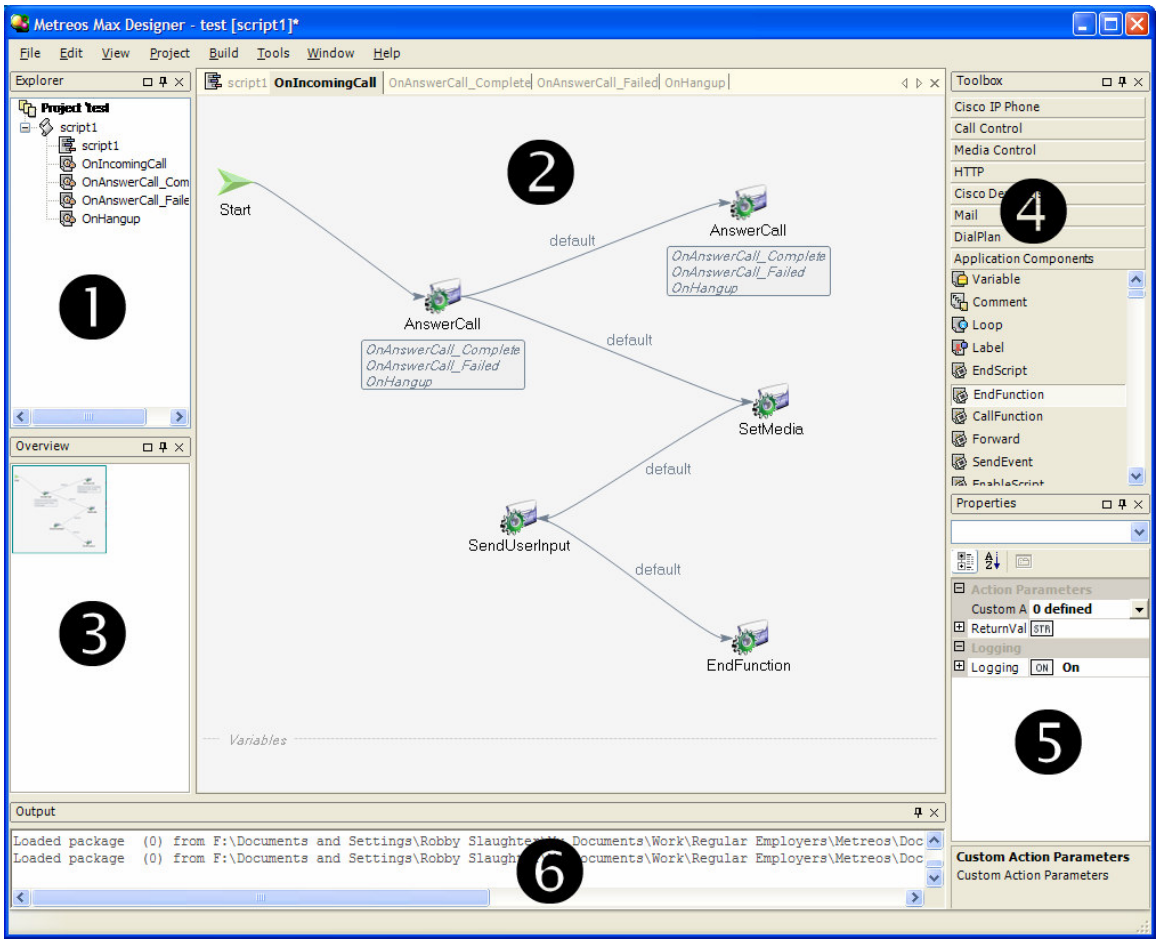

Figure 246: Windows in the Metreos Visual Designer

1. Explorer Window

The **explorer window** lists all chief components of the currently loaded application, grouped by category. Components may include scripts, installers, databases, and media resources. For details on these elements, see the section on MCE application structure. The explorer window acts as the main navigation system for the Metreos Visual Designer, permitting developers to easily switch between each piece of a large application.

2. Application Canvas

The heart of the Metreos Visual Designer is the **application canvas.** The user may create diagrams on the canvas using application elements such as actions, control structures, and variables. Because applications may become much larger than the screen, the canvas may be panned up and down or left and right using scrollbars.

3. Overview Window

Like a map, the **overview window** displays a tiny representation of the current application canvas. The blue rectangle overlaid on the window indicates the area of the canvas currently visible onscreen. Users may simply click and drag a new box to change the viewing area of the application canvas.

4. Toolbox

Application components reside in the **toolbox**. To add an element to an application, click on the category heading listed in the toolbox and then drag and drop the desired component onto the application canvas. The Metreos Visual Designer includes a host of built-in toolbox categories. Third-party components may easily be added to the toolbox by developing new protocol providers. For more information, see the document called *Metreos Communications Environment: Advanced Developer's Guide*.

5. Properties Window

Almost all application components require or support configuration. Developers change these settings using the **properties window**. Simply click on a specific instance of a component on the application canvas, or on the original item in the toolbox. This updates the properties window to contain details for the selected component.

6. Output Window

Any time a user compiles an application for eventual installation to the Metreos Application Server, the compiler generates informational, warning, and error messages. As in many development environments, these items appear in an **output window**.

Projects and Files

The Metreos Visual Designer uses a project metaphor to aid developers in organizing applications and applications and application components into a logical system. Creating a new project within the MVD requires creating a folder on the developer's system or a shared network drive, usually with the same name as the application.

Within this folder, the Metreos Visual Designer stores all the application components. The folder contains files which represent all the scripts used in the application, the files with configuration and installation data for the application, and the files containing SQL code for databases required by the program.

While developers may add additional files to this folder structure for notes, documentation, or other elements, Metreos neither recommends nor supports directly modifying any of the files in the project using tools other than the Metreos Visual Designer. Doing so may produce unexpected results, and will void your warranty and cancel your rights to any and all technical support from Metreos.

Using the Metreos Visual Designer

Before creating your first application with the Metreos Visual Designer, you should learn to operate and manipulate the MVD interface. While this is similar to other graphical environments with which you may be familiar, learning the distinctions and capabilities is a worthwhile endeavor.

Creating Projects

Creating a new application within the Metreos Visual Designer starts with the creation of a new project. Select the *File->New Project* menu option to bring up the New Project dialog. Indicate the name of the new application, and select a location to store the project folder on your computer or an available network drive. You may later move the project to a new location if you desire, simply by relocating this folder.

After creating a new project, you will notice an icon appear in the Explorer Window, adjacent to the name of your new project. This indicates the components which comprise your project. As you can see from the window, your project currently contains four empty folders for each of the application component types.

Creating a Script

The script is the largest and most important grouping within an application. Every application requires at least one script, and may contain any number of additional scripts as required by its design. To create your first script, select *Project->Add Script->New Script* from the main menu.

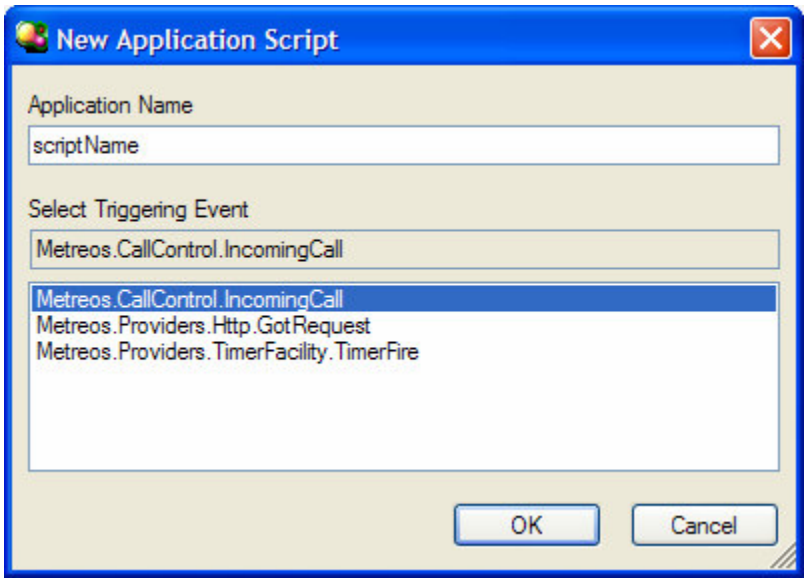

Figure 257: Creating a new script

Scripts require a unique name, and must be triggered by some external event. Provide a name in the box labeled Script Name and select the triggering event from the list below. Click OK to continue.

The Metreos Visual Designer creates the new script using the name you provided. This updates the application canvas, creating a new tab with the title of the script. You can see that the script itself contains elements, including Events and Functions as well as Variables.

Furthermore, you can use the plus and minus icons to expand and collapse the available trees in the Explorer window.

Finally, note that the Metreos Visual Designer automatically created a function based on the trigger you selected, visible in the Events and Functions section as well as an additional tab adjacent the script. We will return to this momentarily.

Additional Scripts and Script Navigation

Projects may contain dozens of scripts, and scripts with useful functionality might be used in multiple different applications. To add additional new scripts, repeat the process described above. You may also add existing scripts using the *Project->Add Script->Existing Script* menu option. Simply use the file browser to select a script file to bring it into the project.

You can easily move between scripts by using the Explorer window. Simply double-click on a script to select it and update the application canvas. If you have modified the script since you last the saved the project, the Metreos Visual Designer will ask you if you want to save the current script before moving on to another. You may not have unsaved changes on multiple scripts simultaneously. This design feature helps developers keep track of where they are making changes and improvements.

Event Handlers and Triggering Events

You may have noticed the appearance of a function in the Events and Functions section of the new script. The Metreos Visual Designer created this function automatically to act as the event handler for the triggering event, which you specified when you originally created this script. This will be the first function executed within this script.

Manipulating Elements on the Application Canvas

Select an automatically created event handler by clicking on a script name in the explorer window followed by second tab atop the application canvas. You will see a light gray background with an icon arrow and the label “Start”.

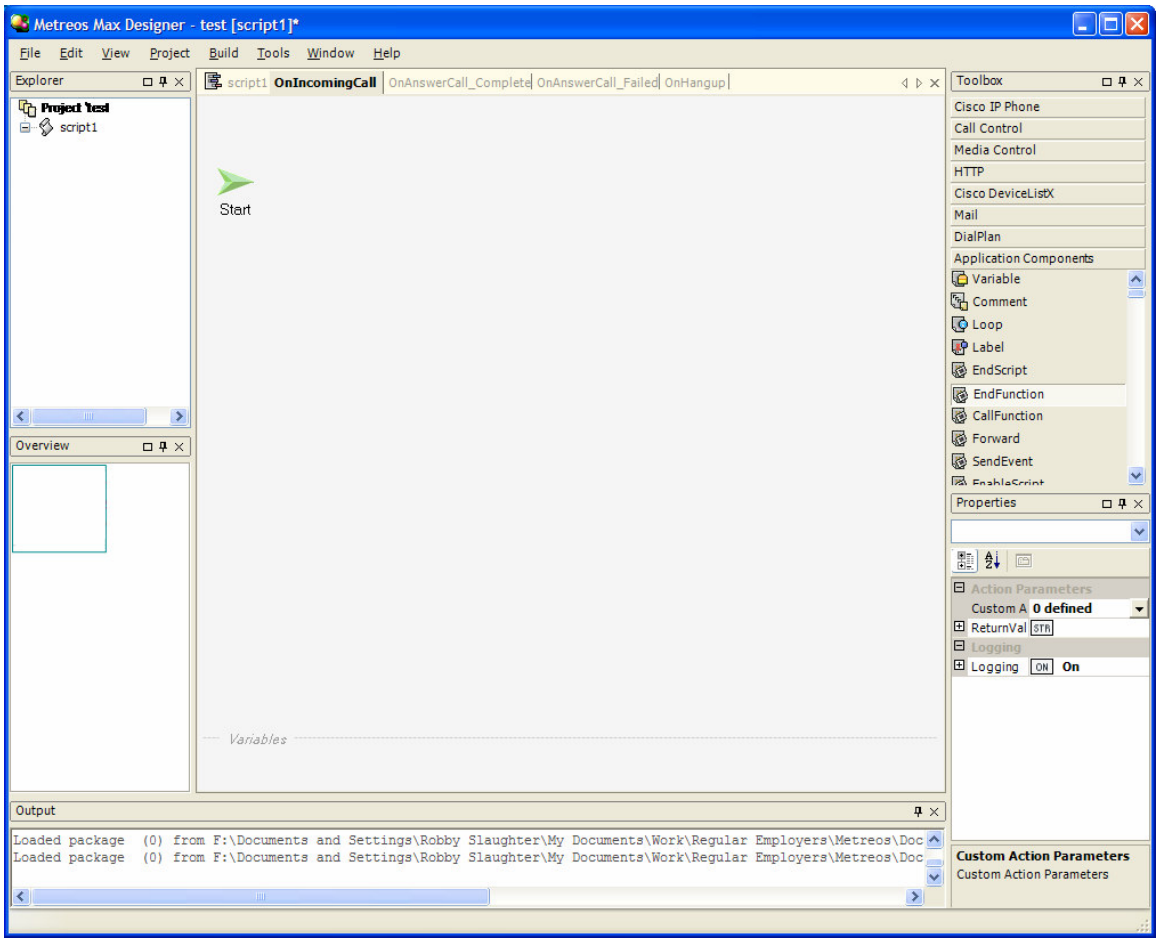

Figure 268: Blank Application Canvas

The Start icon is one of the basic elements on the application canvas: the start node. This is where function execution begins. You can move the start node (and any node) by placing your mouse atop the node name, then clicking and dragging it 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 SendRequest, from the toolbox onto the application canvas.

The Metreos Visual Designer automatically connects the start node to the newly created SendRequest node using an arrow. This indicates that immediately after the function begins executing, control will pass to the SendRequest action.

To delete an arrow, a node, or any other element on the application canvas, simply click once on the title or border of the element and press the delete key. Optionally, you may 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 elements. Then, you can move or delete the elements as a group.

Manipulating Execution Paths (Arrows) on the Application Canvas

On the application canvas, arrows indicate the path of execution from one element to another. Sometimes, the Metreos Visual Designer will be able create these arrows automatically, if there is no option for another path. Often you will need to indicate the execution path yourself by drawing the arrow manually.

To draw an arrow between two elements, place your mouse in the center of the starting element, where the cursor will change to indicate you may draw an arrow. Then, click and drag to the second element to create the node. If you need to reverse the flow of execution, simply click on the arrow and press the delete key, and draw the arrow in the opposite direction.

Execution paths sometimes depend on the result condition of the previous element. In this case, a text description will appear 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 canvases clean, you can change the style of an arrow by right-clicking near the middle of the arrow. You may determine whether the arrow is segmented, meaning formed from a basis of individual straight lines. Also, you can set the link style to be either "curve", "line", or "bevel".

Elements which 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 below the icon.

Figure 29: Example of action that automatically includes event handlers

These items are event handlers, in the same way the original function you created is an event handler. You will notice that these new functions also appear as tabs on the application canvas. In an actual application, you will need to visit these tabs and create functions to resolve the outcomes of these events.

Special Elements: Variables and other Application Components

In addition to the variety of elements available elsewhere in the toolbox, the application components category includes such utilities as loops, comments, and variables. Loops are dragged and dropped onto the screen like other elements, but may be clicked once and resized using small control boxes reminiscent of other programs. Comments, which store

information text indicated to explain steps or direct the programmer may be directly edited and will automatically resize to fit the new text. The last element, variables, store data within a function for easy access, and are created by dragging and dropping the element to the canvas.

Variables, however, reside in a group near the bottom of the application canvas. Variables will only be visible when your mouse is near the bottom of the screen.

To rename a variable, click once on the variable name and type a new name. To delete a variable, click on the variable icon and press the delete key, or right-click on the variable and choose the Delete option.

Element Properties

Almost all elements may be further refined and configured through the use of the Properties Window. After selecting any item with a single click, the properties window displays a grid of options relating to the selected element. These choices are grouped either by category or alphabetically. To switch between these two options, click either the grouping button or the alphabetical button.

Editing most properties simply involves clicking on the box to be modified and changing the value. Some properties may only accept a small range of values --- these appear as dropdown boxes. Finally, some complex properties contain a small plus sign to indicate they expand into multiple sub-elements.

The “Hello World” Application

Practically every book on software development includes the “Hello World” program as its first example. This trivial application usually prints the message “Hello World” to the screen, and then exits. The purpose of the example is to demonstrate the structure and style of the language, and to give the student some exposure to actual code.

For many application development environments, introducing this application makes practical sense as well. Displaying text to the console will certainly be used in future applications, and creating and executing the “Hello World” program demonstrates their development environment is correctly configured.

The Metreos Communications Environment, however, is unlike traditional application development environments. For instance, the MCE does not present a text console as an output device. Indeed, the primary I/O devices in a telephony system are telephones.

Furthermore, traditional applications and MCE applications begin execution under different conditions. In the former case, the user double-clicks an application icon or types a program name at the command line. The Metreos Communications Environment, however, utilizes an event driven architecture, meaning applications begin execution automatically as a result of an external occurrence. Traditional apps are started, but MCE

telephony apps are triggered. The difference may seem subtle, but the implications are profound.

Naturally, you can still use the MCE to send a message like “Hello World” to a predetermined destination, so for the sake of tradition we will include this example. Since there is no “console” for the MCE to display text, we will instead write the string “Hello World” to a log file. And since there is no way to explicit start a particular application, we will select a generic trigger to ensure the program may be executed.

“Hello World” Development steps

Use the following steps to create the “Hello World” application using the Metreos Visual Designer:

Create a new project using the *File->New Pro*ject menu option. Title the project “Hello World” and store it in a logical location on your system.

Add a new script using the *Project->Add Script->New Script* menu item. Name the script “main”, and select the trigger Metreos.CallControl.IncomingCall.

Click on the tab labeled **OnIncomingCall** on the application canvas to bring up this function. This is where we will “write the code” for the program.

Select the Application Components category from the toobox, and drag-and-drop the Write action onto the application canvas. The MVD will automatically connect the start node to this node on your behalf.

Click once on the newly created Write action to make it the active selection. Move to the Properties Window and select the box adjacent to Message. Type the words “Hello World”, without the quotes. Enter the word “Info” into the box adjacent to LogLevel.

Return to the toolbox and select EndScript from the Application Components category. Drag and drop this onto the application canvas.

Place your mouse over the center of the Write action until your cursor changes, and then click-and-drag to attach an execution flow arrow to the EndScript component.

The “Hello World” application is now complete. Don’t forget to save your work using *File->Save*!

“Hello World” Deployment Steps

Once you have created the “Hello World” application, you will probably want to deploy and test this new program. First, you’ll want to confirm that the Metreos Visual Designer knows the location of the Metreos Application Server. Select the *Tools->Options* menu to bring up this dialog.

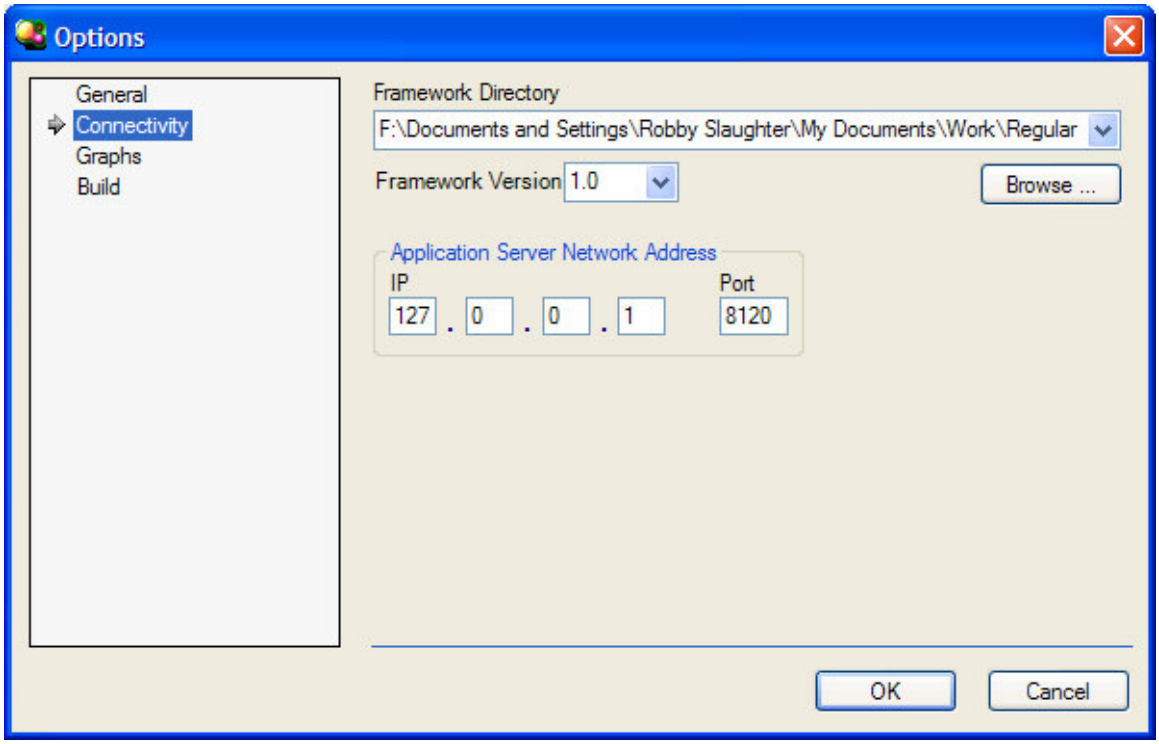

Figure 270: Tools->Options Dialog

Type in the IP address and port of your application server into the "Application Server Network Address" box, and click OK. If you do not have this information, contact your system administrator.

Finally, select the menu option *Build->Deploy* to compile your application and install it on the application server.

Warnings and Error Messages

If you made any mistakes during application development, or if there are other related configuration problems, error messages will be printed in the **output window**.

Testing “Hello World”

To demonstrate the “Hello World” application, place a call which terminates at the address of your application server. Your system administrator can assist you in this process.

Once you hear a ring or other confirmation that the call is being placed, the program has executed. Ask your system administrator for the Application Server log files to see the “Hello World” message.

6. APPENDIX A: ADVANCED TOPICS

Developing Native Actions

The following is an abridged listing of relevant code needed to write 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 ResultData { get { return resultData; } }
            private string resultData;

            public Action () {}

            [Action("Action", true, "Action", "Performs task")]
            public string Execute(LogWriter log,
                  SessionData sessionData,
                  IConfigUtility configUtility)
            {
                  resultData = parameter;
                  return IApp.VALUE_SUCCESS;
            }

            public void Clear()
            {
                  parameter  = null;
                  resultData = null;
            }

            public bool ValidateInputs()
            {
                  return true;
            }
      }
}
```

Listing 11: Simple INativeAction implementation.

All native actions must implement INativeAction. The native action must implement a default public constructor, as well as 3 methods: `Execute`, `Clear`, and `ValidateInputs`. The `Execute` method is called by the runtime environment when the action is invoked. `ValidateInputs` occurs immediately before `Execute`, and `Clear` occurs immediately afterwards.

Like most other programming environments, an action, or method, has inputs and outputs. A native action can support multiple incoming parameters, as well as multiple

outgoing parameters. The action parameters are delineated by decorating a .NET property with an `ActionParamField` attribute. In this attribute, one can specify the name in which it will appear in the Metreos Visual Developer. Also, the developer can specify whether the action parameter is required or not. This, along with the type of the property, is of critical importance at runtime. If a required parameter is missing when the action is invoked, the script will fail and stop operating. The same is true if the incoming parameter is of a different type than the one expected or is not convertible to the required type. For optional parameters, a type mismatch will also cause the script to stop executing, just as with required parameters.

Outgoing parameters are specified by decorating properties with the `ResultDataField` attribute. This attribute is simpler. The developer can optionally specify a description for the result data field.

The developer can be assured that parameters marked as required are present when the `Execute` method begins execution. All parameters are guaranteed to be of the correct type. Beyond that, the `ValidateInputs` exists for the developer to cleanly perform additional validation of the incoming parameters. If the validation fails, a return value of `false` will also stop execution of the script.

In the `Execute` method, the developer can then perform any logic necessary to fulfill the task of the native action. Before exiting the method, be sure to assign the result data fields to whatever it is that they should be: these values will then pass down to the variables specified for that action.

The `Execute` method also returns a `string`. This value determines what branch is taken after this action finishes executing. If the action ultimately failed, this value can be set to `IApp.VALUE_FAILURE ("failure")`. If it succeeded, `IApp.VALUE_SUCCESS ("success")` can be used instead. These constants are located in `Metreos.Interfaces.dll`. Though these return values are not required (any string value is valid to branch on), they then have the benefit of conforming to the same values the framework native actions use, increasing familiarity for the user of the Metreos Visual Designer.

Of additional use to the action developer is the `LogWriter` provided in the `Execute` method. Usage of this object, via the `Write` method, will output text to the console (subject to configured `LogLevel` constraints). Also, alarms can be set and cleared using the `SetAlarm` and `ResetAlarm`, causing emails to be sent to the administrator.

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

The `SessionData` object is provided as a convenience to the developer, since it can be accessed in `UserCode` and c# code snippets, as well as in this `Execute` method.

Two other attributes have thus far gone not discussed in Listing 11.

The `PackageDecl` allows the developer to specify the namespace of the package that this action resides in, and a description of the package. If any other actions exist in this project, it is not necessary to decorate them with the `PackageDecl` attribute. If they are in the same assembly, they are assumed to be in the same package.

`Action` specifies a number of properties for the action itself. In order according to how they appear in the listing: the name of the action; the display name in the Metreos Visual Designer; and a description.

For deployment, the name of the assembly file in which the native action resides must be the exact same name as the action namespace. Do not use `Metreos.*` or any other Metreos namespace. Instead, native actions of a particular type should be grouped in a library together with a namespace and output filename of the following form:

`{company name}``.Native.`**`{category name}.{action name}`**

To use this assembly in the Metreos Visual Designer, execute the Pgen.exe tool, provided with the Metreos framework, on the assembly to generate an XML metadata file. Place the assembly and the XML file (of the same name as each other aside from the file extension) into the same folder. This XML file and assembly together are now ready to be referenced and used by the Metreos Visual Designer.

Developing Native Types

Often times when developing applications new types must be created to encapsulate different forms of data. The Metreos Communications Environment allows developers to extend its type system by building **native types**. A native type is a .NET class that implements certain well defined interfaces exposed by the Application Framework of the MCE.

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

            public Type()
            {
                  Reset();
            }

            public bool Parse(string newValue);

            public void Reset()
            {
                  _value = "Prefix: ";
            }

            public string ExampleMethod(string var)
            {
                  return _value + var;
            }
      }
}
```

Listing 12: Simple IVariable implementation.

All native types must implement the `IVariable` interface.

The two methods to implement in order to meet the criteria of IVariable are `Parse(string)`, and `Reset()`.

`Parse(string)` is required in order for support of initializing with default values in Metreos Visual Designer, since the user can only specify a text string for a default value. It is not necessary for the Parse to actually do anything; default values then have no meaning in MVD for this variable type. A `true` should be returned in every case aside from a critical failure, as the script attempting to initialize will stop execution.

Native types are also assigned to through the result data fields of native or provider actions. If the type being returned by the action is exactly the same type of the variable receiving the value, then a direct assignment is made. If this condition is not met, then `Parse(string)` may be invoked, or any overload of `Parse` which better matches the incoming type from the action. Therefore, various incoming types can be assigned to your native type, based on how many `Parse` overloads you have. Alternatively, you can catch all assignment attempts with `Parse`(object obj), and use `is` statements to check for various types.

In code listing 12, the `ExampleProperty` and `ExampleMethod` are only for tutorial purposes. The idea here is that one can then use these methods, properties, indexers, etc., to access the data contained by this type in actions after this variable has been assigned to, exclusively in UserCode or c# code snippets.

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 which holds the code. In the example above, the code would be written in the Company.Types.Example namespace and would be located in Company.Types.Example.dll.