

Managed Objects for Internal Application Control

Managed objects are fundamental to the software architecture of the HP E5200A broadband service analyzer. Typically used to control remote network elements, managed objects are also used internally by the service analyzer's application to control application objects.

by John P. Nakulski

A managed object is a software abstraction that acts as a proxy for a real object. It is used by remote clients, such as objects in other threads of execution, or on remote hosts. Managed objects are often used in network management applications to enable control of remote network elements. For example, a router or switch might offer a managed object interface to enable remote configuration and monitoring from a central network management system.

In the HP E5200A broadband service analyzer, managed objects are used in a different way. The service analyzer's application uses managed objects internally to control application objects. For example, a CellLossTestMO managed object is used to control the service analyzer's CellLossTest object. An interface to the same managed objects is also available to external users.

Besides being fundamental to the service analyzer's software architecture, managed objects have proved useful in other ways, including:

- In the decoupling of the user interface
- As the foundation for a command line interface
- In the development of a macro recording facility
- For backward compatibility with a programmer's interface
- In the unit and regression testing of software components
- For the simultaneous control of several remote service analyzers for distributed testing of broadband networks.

Remote Object Communication

The service analyzer contains three Intel i960 microprocessors running VxWorks—a lightweight, multithreaded operating system. Each processor runs several threads of execution. A layer of software called *object transport* provides an interface for streaming *elemental types* between threads of execution on the same processor, between processors on the same host, and between hosts on the same network (see Fig. 1). Elemental types include integers, floating-point numbers, characters, strings, and a few classes such as PDU (protocol data unit).

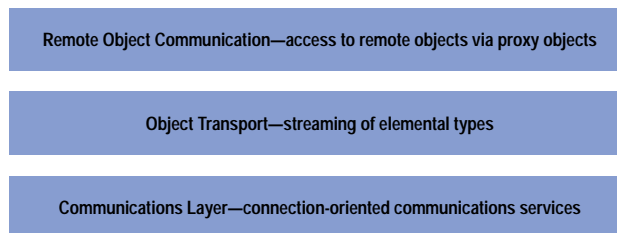


Fig. 1. Layered communication architecture of the HP E5200A broadband service analyzer.

Remote object communication is a software layer built upon object transport that makes object services available to remote clients. Using a proxy object, a client object can invoke the member functions of an object in another thread or on another host almost as easily as calling member functions of objects local to its own thread (see Fig. 2). Of course, each argument of the member functions must be an elemental type.

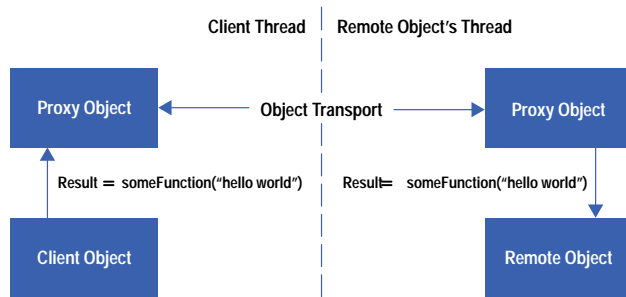


Fig. 2. Remote object communication.

Objects, Attributes, Agents, and Managers

A managed object software layer builds upon the remote object communication layer to provide a dynamically discoverable software interface. The interface consists of named managed objects, each containing a set of named attributes. While it is running, an application can be *browsed* through this interface for a list of its managed objects and their attributes, each of which can be accessed by name.

A managed object's executable attributes, which can be invoked by name, enable remote execution of an object's functions. For example, the service analyzer's CellLossTest managed object contains an attribute named start which runs a test that measures loss of ATM (Asynchronous Transfer Mode) cells. A managed object's data attributes, which can be read and written by name, typically mirror the state of the object. For example, the CellLossTest managed object has an attribute named cellLossCountThreshold, which represents the number of cells that can be lost before the cell loss test fails.

Each thread that contains managed objects also contains a *managed object agent*, which is responsible for dispatching inward communications by calling the desired function on the specified managed object. Each thread that uses managed objects contains a *managed object manager*, which is responsible for finding and communicating with managed objects in remote threads and on remote hosts. These distributed managers and agents cooperate to provide clients with a single, unified view of managed objects (see Fig. 3).

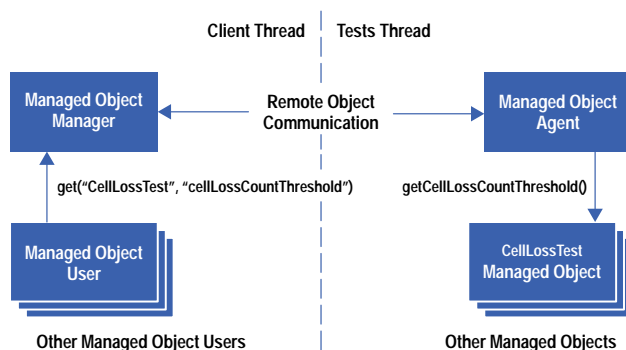


Fig. 3. Managed object communication.

Automatic Code Generation

Each of an application's managed objects is specified in a language-independent *managed object definition* (MOD) file. A MOD file specifies and documents the public interface to a managed object. An example of a MOD file is shown in Fig. 4.

After a managed object is specified, the developer uses a customized Interface Description Language (IDL) compiler to parse the MOD file. This produces C++ code for the *front-end class* (used by the client of the managed object to encapsulate the behavior of the remote managed object) and skeleton C++ code for the *back-end class* (the managed object itself), which the developer must complete.

For example, to use the CellLossTest object from another thread or another host, the client would construct an object of the compiler-generated CellLossTestFE (front-end) class. This offers member functions for invoking executable attributes (such as start() to invoke the start attribute) and member functions for reading and writing data attributes (such as getCellLossCountThreshold() and setCellLossCountThreshold() to read and write the cellLossCountThreshold attribute). The front-end class implements all such functions as operations on the managed object manager.

The thread containing the CellLossTest object itself also contains an instance of the compiler-generated CellLossTestMO (managed object) class. The managed object agent passes the incoming operation to the managed object. The developer needs to *connect* the CellLossTestMO class with the CellLossTest class so that incoming requests such as getCellLossCountThreshold() are

```

#include <elementalTypes.idl>

interface CellLossTest {
    attribute AtoUInt16T transmitVpi;
    attribute AtoUInt16T receiveVpi;
    attribute AtoUInt16T transmitVci;
    attribute AtoUInt16T receiveVci;
    attribute AtoUInt32T durationThreshold;
    attribute AtoFloat64T cellLossRatioThreshold;
    attribute AtoUInt64T cellLossCountThreshold;
    AtoStatusT start();
    AtoStatusT stop();
    AtoStatusT getResultSet(
        out CtsPassFailResultE passFailResult,
        out AtoStringT testReason,
        out AtoUInt32T elapsedTime,
        out AtoUInt64T transmitAverageBandwidth,
        out AtoUInt64T receiveAverageBandwidth,
        out AtoUInt64T cellLossCount,
        out AtoFloat64T cellLossRatio);
    ...
}

```

Fig. 4. An example of a managed object definition (MOD) file.

delegated to the CellLossTest object (see Fig. 5). In this simple example, the CellLossTestMO::getCellLossCountThreshold() function body is filled in by the developer with a call to the CellLossTest::getCellLossCountThreshold() function.

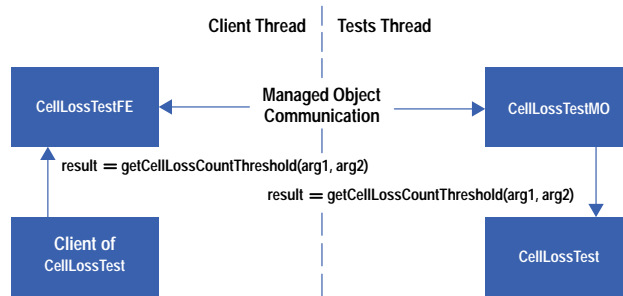


Fig. 5. Interaction of machine-generated objects. (FE = front end. MO = managed object.)

Managed object operations are generally *synchronous*, meaning that the front-end function does not return until the back-end object has completed its execution and has a result available to return. The managed object notification operation, however, is *asynchronous*. Clients *subscribe* to a managed object attribute, supplying a callback function. When the attribute is *triggered*, a notification message is sent and the supplied callback function is invoked. An attribute is generally triggered when it changes or is executed. For example, the testStatus attribute of the CellLossTest managed object is triggered when the test is started or stopped.

Each managed object data attribute must be an elemental type. Each elemental type corresponds to either a C++ built-in type (such as char) or a low-level class (such as Pdu). Developers of a front-end application using another language need to port only those low-level classes and types that are not native to the language.

Reactive Graphical User Interface

The service analyzer's graphical user interface (GUI) is said to be fully *decoupled* from the rest of the service analyzer. The GUI is a separate subsystem (module) and runs in a separate thread, and can even run on a separate host. Other subsystems are not dependent on the GUI; the rest of the service analyzer can run without it.

Further decoupling is achieved through the use of managed objects. Every request from the GUI to another object is a managed object operation. The GUI uses the notification mechanism to learn of and react to the service analyzer's state changes and other events. For example, if an ATM alarm occurs, the GUI is notified and updates the state of its window. This also allows the GUI to react automatically to changes caused by other managed object users (such as the command line interface).

Armed with only the MOD files, a specification of the elemental types, and a specification of the remote object communication and object transport protocols, a developer can write a new user interface in a different language to run on a different host.

Command Line Interface

The design of the service analyzer's command line interface (CLI) exploits the fact that the service analyzer's functionality is available through its managed objects. The CLI *plugs into* the service analyzer at the managed object interface. Each command is interpreted as an action on a managed object, enabling CLI users to control the service analyzer without a GUI.

Tool Command Language (TCL), a public-domain script language, is used as the foundation for the CLI. TCL is embedded in the service analyzer and has been extended with a few simple but powerful functions to enable access to managed objects.

The `moexec` command enables executable attributes to be invoked, while `moget` and `moset` allow the user to read and write managed object data attributes. The `molst` and `moattrlst` commands retrieve the set of available managed objects and the set of attributes for a specified managed object, enabling the user to browse the service analyzer's interface dynamically.

If access to the GUI is not available, the user can log in to the service analyzer remotely using a telnet session to run the CLI.

Macro Recording and Playback

By tapping the communications between the GUI and the rest of the service analyzer at the managed object interface, it is possible to watch the GUI's interaction with the underlying service analyzer. This is exactly how the service analyzer's macro feature works (see Fig. 6).

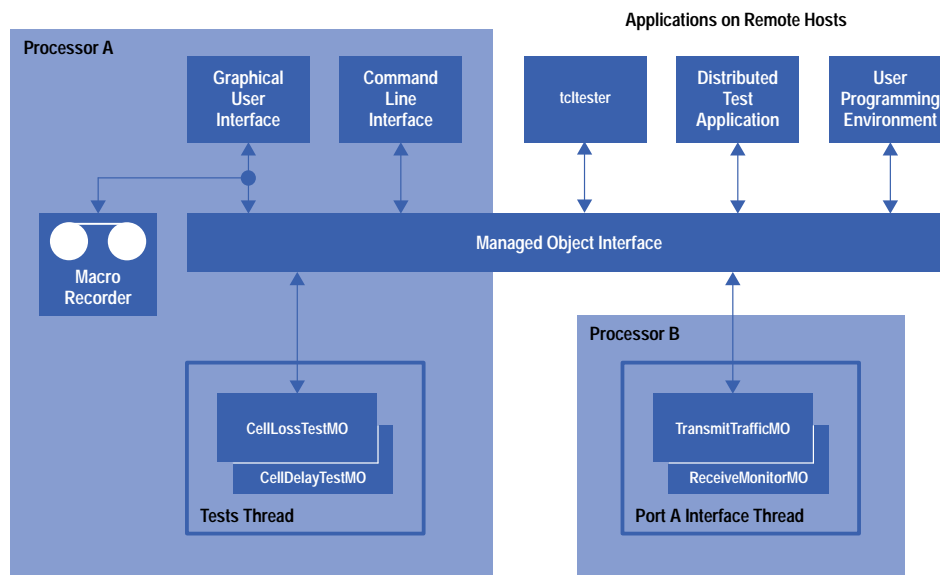


Fig. 6. Users of the managed object interface.

When the macro recording facility is used, a subset of the GUI's managed object operations is converted into CLI commands and is recorded into a macro file. For example, a user may want to record a frequently used `cellLossTest` configuration. When the macro is played back, it is executed in much the same way as a CLI session. If the GUI is active during the playback of the macro, it will respond to service analyzer state changes, allowing the user to watch the effects of the macro as it is running.

Since the macro language is identical to the CLI language, the experienced user can edit and customize a recorded macro or create a new macro. For example, a cell loss test can be executed ten times in succession with increasing transmit bit rates simply by wrapping a for loop around a recorded macro, using a TCL variable to hold the transmit bit rate (see "**Macros**."

User Programming Environments

Backward compatibility with the HP broadband series test set (BSTS) user programming environment (UPE) is also achieved using managed objects. UPE programs, written in C, can be used to control the service analyzer from an HP-UX* workstation or from a PC by calling functions published in an application programming interface (API).

UPE programs are recompiled on the target host and linked with a new set of supplied libraries. The new libraries implement each API function by invoking one or more of the service analyzer's managed object operations via a managed

object manager. An added bonus for users of the new service analyzer is the ability to control several service analyzers from within a single UPE program.

Testing the Service Analyzer

A further bonus gained from extending TCL with managed object commands was the development of `tcldtester`, a tool used by developers to test the service analyzer. `tcldtester` wraps the CLI into a standalone application that can be executed on a host remote from the service analyzer under test. The developer can enter CLI commands interactively to test and debug the service analyzer or execute whole CLI scripts for regression testing.

Distributed Test Applications

Because the service analyzer's managed object communication layer enables communication not only between threads on the same processor but also between hosts on the same network, it is possible for one service analyzer to control one or both ports of a second, remote service analyzer. A CLI user or a standalone PC application can control several remote service analyzers simultaneously. For example, traffic can be injected into different points in a network, several network nodes can be monitored simultaneously, and performance parameters can be measured and collated across an entire network.

One of HP's goals is to develop useful tests for the installation and commissioning of broadband networks. As we gather experience in using the service analyzer for broadband testing, we expect to develop new tests to add to the test suites.

Although we have been closely involved with users during the development of the service analyzer, it is likely that users will have requirements for distributed tests unimagined by our developers. By providing an open managed object interface, users can develop their own broadband test and measurement applications.

Acknowledgments

I wish to acknowledge the contributions of the many individuals who participated in the design and development of software for the HP E5200A broadband service analyzer, especially Earl Chew, who had the foresight to provide an open software interface, and Chris De Souza, who embedded TCL, developed the macro capability, and fostered the architectural vision in the face of tight deadlines. This investment has already enabled the creation of several new applications. In addition, I would like to thank Dan Smith for helping develop the example macro and Sonja Draeger, Gail Hodgson, and Debbie Augusteyn for their helpful suggestions and thorough review of this paper.

HP-UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

-
- ▶ [Go To Subarticle 11a](#)
 - ▶ [Go To Next Article](#)
 - ▶ [Go To Table of Contents](#)
 - ▶ [Go To HP Journal Home Page](#)