HP OpenView Agent Tester Toolkit

In developing HP OpenView agents, a major challenge is to develop and test both the agent and the manager simultaneously. To fill this need, the HP OpenView Agent Tester Toolkit generates tests and allows the developer to execute these tests individually or as a set.

by Paul A. Stoecker

HP OpenView agents can be created by telecommunications network management developers either by using tools or by writing the code directly. The tools available include the *GDMO* Modeling Toolset* (see ***Article 4***), which helps in the design and specification of network management objects using the GDMO language, and the *HP Managed Object Toolkit* (see ***Article 6***), which accepts GDMO documents and produces C++ code to implement a default agent that meets those specifications. Whether the developer builds an agent using these tools or writes the code by hand, one of the major challenges is to develop and test both ends of the communications link simultaneously—the *agent* controlling the managed device and the *manager* that sends requests to the agent and receives the responses. To fill this need, the new HP OpenView Agent Tester Toolkit generates tests and allows the developer to execute these tests individually or as a set.

The Role of an Agent

An agent program enables other programs, called managers, to control physical and logical resources. Examples of resources that are controlled by agents are telephone switching equipment and phone service databases. From a centralized location, a telephone service provider can use automated processes to monitor the performance of the communications lines, reroute traffic as necessary, and maintain the business and accounting records. Because the communication protocol between managers and agents has been standardized, a wide area network of multivendor equipment can be efficiently controlled from a small number of central locations.

The resources being monitored and controlled are modeled as objects called *managed object classes*. Managed object classes are logical groupings of the attributes, events, and actions associated with a resource. A GDMO specification defines the various managed object classes that make up the interface to the resource. Instances of these classes are called into existence by sending a create request. The attribute values for an instance are accessed by issuing set and get requests to change or retrieve the attribute values, respectively. Other message types remove an object instance, allow the agent to notify interested parties of an asynchronous change, or cause the agent to perform some agreed-upon activity.

A collection of managed object instances and their relationships is called the *containment tree*. Subobjects are logically contained or grouped within other objects. Fig. 1 depicts a portion of a containment tree. Each of the boxes in Fig. 1 represents an object instance. The label in each box identifies the object class of that instance. For example, in Fig. 1, a fiber-optic network is composed of two network elements. In one of those network elements, the regenerator and multiplexer sections are shown.

One of the attributes within each of the contained object instances is designated as the *distinguishing attribute*, and the value of this attribute is used to uniquely distinguish that instance from all of its siblings. The containment tree is used to uniquely identify, or *name*, an object instance. An object instance anywhere in the containment tree is identified by specifying the distinguishing attribute and its value from the top of the tree down to the desired instance. The concatenation of all of the distinguishing attributes along this naming path is called the *fully distinguished name*. In Fig. 1, the fully distinguished name for the multiplexer section would consist of the sequence networkId = “net1”; elementId = 5; muxId = 56.

Agent Development

The Managed Object Toolkit saves an enormous amount of work by handling all of the overhead of decoding and validating incoming requests, locating the selected object instance within the containment tree, and invoking an appropriate C++ method on the selected object. However, the attribute values that are set or retrieved by the initial Managed Object Toolkit output are only internal representations. The developer is responsible for filling in empty C++ stubs to make the internal attribute values reflect the state of external physical devices. During this coding process, it is helpful to simulate the requests that will eventually be sent by a manager.

The Agent Tester Toolkit performs this task in two steps. First, it creates test requests from the GDMO specification. Second, it transmits these requests over the network to the agent and receives the responses. During the development phase, these test files can be sent individually and the responses viewed interactively. As each agent operation is implemented,

* GDMO is the ISO (International Standards Organization) *Guidelines for the Definition of Managed Objects*.

the associated test requests can be added to a test suite. The accumulated tests can then be run in a batch mode to check that previously implemented functionality still works properly. Fig. 2 depicts the sequence of steps needed to generate and send the test requests, and shows how the Agent Tester Toolkit relates to other development tools.

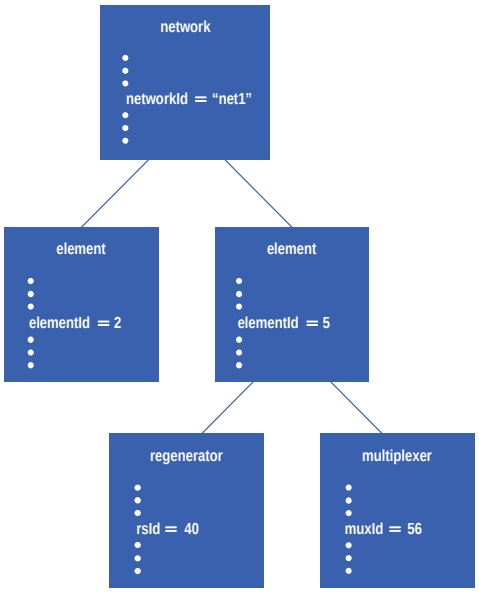

Fig. 1. *A containment tree.*

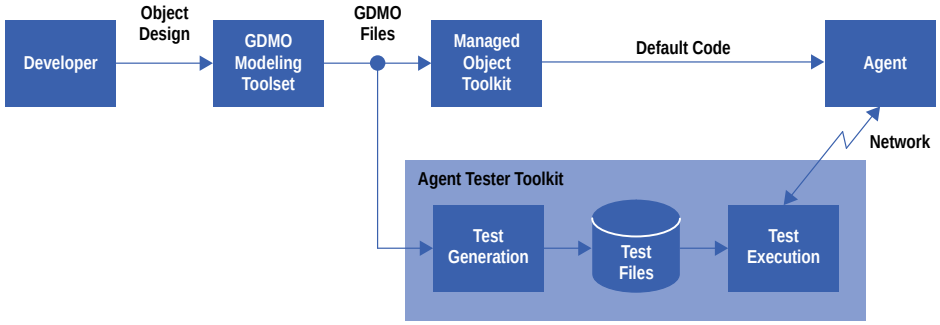

Fig. 2. *Agent development and testing tools.*

Running the Agent Tester

The components of the Agent Tester Toolkit are command-line tools that are invoked in a straightforward way. For example, ovatgen -t /tests gdmo.mib reads the GDMO description in the file gdmo.mib and generates a set of test requests stored in files under the directory /tests. Next, tests for a particular object instance can be sent to the agent as follows:

```
$ ovatrun -i
>create
...
>getall
...
>mytest
...
>delete
...
```

The -i option to ovatrun specifies the interactive mode, in which the user can type the name of a test file in response to the > prompt and the response from the agent is displayed immediately (shown by the dots above).

The test files represent CMIS (Common Management Information Service, ISO/IEC 9595) operations, such as create, set, get, and so on, and are stored in a directory layout that mirrors the organization of the agent's containment tree, with each directory named by its associated managed object class name. At each level in the containment tree, test files are generated that create an object instance, get all attributes, and delete the instance. If there are changeable attributes, tests are also generated that set those attributes to new values and retrieve the changed attributes. In addition, files are generated that test

attribute groups and actions. Documentation files describe the object identifiers used in the tests and optional features called *conditional packages*.

Each test request is written in a format called ASN.1 value notation, which is a standardized format described in ISO and ITU-T documents (8824 and X.208, respectively). ASN.1 (Abstract Syntax Notation One) is a notation for expressing the types of the attributes and operations. For example, a test file that contains a get request to retrieve the current values of several attributes might appear as:

```
GetArgument {
  -- passwordEntryManagedObjectClass
  baseManagedObjectClass {1 3 6 1 4 1 11 9 81},
  baseManagedObjectInstance distinguishedName : {
    {
      {
          -- passwordRootName
          attributeType {1 3 6 1 4 1 11 9 29},
          attributeValue Mod.RootSyntax 0
      }
    },
    {
      {
          -- loginName
          attributeType {1 3 6 1 4 1 11 9 21},
          attributeValue Mod.LoginSyntax "paul"
      }
    }
  },
  attributeIdList {
    -- password
    {1 3 6 1 4 1 11 9 22},
    -- userID
    {1 3 6 1 4 1 11 9 23}
  }
}
```

In this example, the first word, GetArgument, announces the ASN.1 type whose value follows. A GetArgument is a structured type, and in this example its fields are baseManagedObjectClass, baseManagedObjectInstance, and attribute-IdList. Lines beginning with -- are comments inserted by the Agent Tester Toolkit generator to help the reader identify the various *object identifiers* (OIDs), which are strings of digits (e.g., {1 3 6 1 4 1 11 9 23}) that uniquely identify attributes, classes, and other fields. Returning to the GetArgument request, when sent by the Agent Tester Toolkit it asks the agent to return the current value of the password and user ID attributes of an object of class passwordEntryManagedObjectClass. The particular instance is identified by an object instance passwordRootName = 0, which in turn contains the desired subobject loginName = paul. A typical response would be:

```
GetResult {
  managedObjectClass {1 3 6 1 4 1 11 9 81},
  managedObjectInstance distinguishedName : {
    {
      {
          attributeType {1 3 6 1 4 1 11 9 29},
          attributeValue Mod.RootSyntax 0
      }
    },
    {
      {
          attributeType {1 3 6 1 4 1 11 9 21},
          attributeValue Mod.LoginSyntax "paul"
      }
    }
  },
  currentTime "19960327145135",
  attributeList {
    {
      attributeId {1 3 6 1 4 1 11 9 22},
      attributeValue Mod.PasswordSyntax "secret"
    },
```

```
      {
        attributeId {1 3 6 1 4 1 11 9 23},
        attributeValue Mod.UserIDSyntax 4463
      }
    }
  }
```

This response returns the requested class and instance information, and reports that the values of the two requested attributes password and userID were secret and 4463, respectively.

It is also useful to gather as much information as possible when error conditions exist. For example, if we try to query an object that doesn't exist, an error is returned, letting us know what aspect of the request was rejected:

```
$ ovatrun –i
>getbad
–– Error: No such object instance
ObjectInstance distinguishedName : {
  {
    {
      attributeType {1 3 6 1 4 1 11 9 29},
      attributeValue Mod.RootSyntax 0
    }
  },
  {
    {
      attributeType {1 3 6 1 4 1 11 9 21},
      attributeValue Mod.LoginSyntax ”joe”
    }
  }
}
```

Test files are ordinary text files, and customized tests can be crafted using the generated tests as guides. Several supporting tools are included in the Agent Tester Toolkit.

Batch Testing

After portions of the agent have been developed and the tests are working individually, it is good practice to run the tests and check the results in an automated fashion. This is useful to monitor existing behavior of an agent as new code is added, or to be able to repeat the testing process as new versions of the agent are developed or the agent is ported to new hardware platforms. To this end, the Agent Tester Toolkit's run program can execute a sequence of tests in succession. The command is ovatrun without the -i option:

```
$ cd /tests
$ ovatrun
```

This causes the list of tests in a default test director file, batch_list, to be run and the responses stored. After all tests have been run, the responses are compared against a set of known-good results, and summary statistics are prepared in a log file, reporting the number of tests run, passed, and failed. The known-good result files are generally prepared by copying actual response files that have been manually verified. A utility tool is provided that copies result files into place as known-good comparison files. As part of the copying process, this utility removes lines that contain the current time, since this would needlessly cause comparison failures in future test suite runs.

The test director file in its simplest form contains the names of the test files and the order in which they are to be sent. Optional commands in this file allow for more complex situations. For example, ISO standard 10164-1 identifies situations (object creation, object deletion, and attribute value change) in which the agent should emit an event so that all interested managers can maintain a synchronized view of the agent's state. To alert the Agent Tester Toolkit to expect both a response to one of its own create, delete, or set requests and the resulting event emitted by the agent, the pair command can be used. For example, the command pair password/create sends the request command contained in the file password/create and then receives both the confirmation of the request and a notification that the creation has occurred. Similarly, if an isolated event is expected, the event command can precede the name of a file with which the arriving event will be compared. Other commands, such as a shell escape to execute any user command, allow customized testing. For example, a shell escape allows the test designer to send a signal to the agent process to trigger some behavior, such as the sending of an event. This simulates the behavior of the agent in actual operation where some asynchronous condition might cause the event, while still allowing the test process to receive a predictable stream of responses from the agent. Other commands allow finer control over the testing process. For example, a timeout value can be set that controls how long the tester will wait for a response before aborting any single test. An example of a test director file with some of these commands included is as follows:

```
# Comments begin with the '#' character
# The following files are regular tests to get
# the attributes in the already-created
# Root Managed Object Class
root/passFileMOC/getall
# Some of the next tests expect both a response
# and an event
pair root/passFileMOC/passEntryMOC/create
root/passFileMOC/passEntryMOC/getall
pair root/passFileMOC/passEntryMOC/set
root/passFileMOC/passEntryMOC/get
pair root/passFileMOC/passEntryMOC/delete

# Set the timeout to 30 seconds
timeout 30

# Send a UNIX signal that triggers an event
! kill SIGINT $(AGENT_PID)
# Receive the event
event root/passFileMOC/passEntryMOC/event1
```

Finer Control of the Generation Process

A powerful feature of the object-oriented design methodology is that the standards bodies have invested much energy into constructing managed object class building blocks. A side-effect, however, is that in most cases the standard documents from which specific agents inherit contain far more definitions than are needed for that agent. In the case of the Managed Object Toolkit, this causes needless code to be generated, producing a larger agent than is required. To counteract this effect, the Managed Object Toolkit allows developers to specify a subset of the managed object classes, so that code is generated only for that subset. The Agent Tester Toolkit accepts the same subset specifier, and tests are generated only for that subset.

In some cases, greater control over the nature of the generated tests is needed than simply selecting a subset of managed object classes. An example is the containment tree example given in Fig. 1 that began with a network object as the root node. The GDMO description of a network class might allow (as it does in ITU-T Recommendation M.3100) that network to be decomposed into subnetworks and subsubnetworks, and so on. To allow the test designers to specify how many levels of decomposition the agent is expecting, a containment tree specification file can be provided to the test generator. This specification file is formatted like an outline, with the level of indentation indicating how deeply under the root node each class is contained. For example, the containment tree in Fig. 1 would be depicted:

```
network
> element
> > regenerator
> > multiplexer
```

(Only one of the element nodes is shown. It will be explained later how to include both circuit branches.)

If the agent is expecting the network level to be expanded into network and subnetwork levels, this change can be incorporated in the specification file by introducing another network node and indenting its children by an additional level:

```
network
> network
> > element
> > > regenerator
> > > multiplexer
```

This change adds an element to the distinguished name (corresponding to the subnetwork distinguishing attribute value) in each test file, so such containment changes have far-reaching effects. Making such decisions early in the specification phase saves much work compared to adjusting already generated test files.

As mentioned earlier, the tests are placed in a UNIX directory structure that parallels the containment tree structure, with each level named by its associated managed object class name. In many cases, managed object class names can be lengthy, and a pathname to lower-level test cases composed of a sequence of those names can be unwieldy. For example, names such as trailTerminationPointBidirectional and connectionTerminationPointSource appear in the standards, and when several of these are joined (as is typically done when specifying containment relationships), the combination is hard to read. To populate the directory structure with shorter, meaningful names, a default heuristic is applied that selects a few letters from each segment of a managed object class name. For example, a file deep in the tree described so far might be named netw/netw/elem/mult/create. Alternatively, the test developer can override this heuristic by specifying shorter names in an optional field in the specification file:

```
network (net)
> network (subnet)
> > element (NE)
> > > regenerator (rs)
> > > multiplexer (mux)
```

Finally, the GDMO document doesn't specify actual attribute values, so the containment tree's distinguishing attributes have to be supplied by the test designer. Once again, much work is saved by specifying these early, rather than fixing tests after they are generated. These distinguishing attributes can be assigned in the last optional field of the specification file:

```
network (net) networkId="ftc"
> network (subnet) networkId="Bldg1"
> > element (NE2) elementId=2
> > element (NE5) elementId=5
> > > regenerator (rs) rsId=40
> > > multiplexer (mux) muxID=56
```

Note that by including the distinguishing attribute values, we can differentiate between the two element sibling branches.

A supporting tool called ovatct reads GDMO files and produces a skeleton specification file similar to the one above (using the same subset selection file as the Managed Object Toolkit, if provided). More meaningful abbreviations and attribute values can be noted in the specification file and then used as an input to the test generator to guide the production process:

```
ovatct gdmo.mib > spec_file
ovatgen -t /tests -f spec_file gdmo.mib
```

Summary

Key design goals of the Agent Tester Toolkit include supporting agent developers during the development and maintenance phases, and confirming compliance to the GDMO specifications the agent is to implement. Also important is the ability to generate tests iteratively for evolving designs, without time-consuming configuration changes of the test engine itself. The Agent Tester Toolkit complements other tools in the development life cycle.

- Go to Article 10
- Go to Table of Contents
- Go to HP Journal Home Page