Open Management Infrastructure (OMI)

Getting Started

Microsoft Corporation
March 2014

Table of Contents

1 Introduction

This manual explains how to get started with OMI. It is by no means a complete reference, but hopefully after reading it you will be able to:

- Build from the source distribution.
- Install the distribution.
- Start and stop the server.
- Use the command-line client.
- Develop and test a simple provider.
- Develop a simple client application.

1.1 What is OMI?

OMI is a software service that runs on managed nodes. It provides the manageability infrastructure for building distributed systems management applications based on DMTF management standards, including:

- CIM Infrastructure Specification (DSP0004).
- CIM Schema (`http://dmtf.org/standards/cim`).
- Generic Operations Specification (DSP0223).
- WS-Management Protocol (DSP0226, DSP0227, DSP0230; see also ISO/IEC 17963:2013).

These standards define:

- A **meta-model** that defines rules for forming classes, properties, methods, and instances.
- A **schema** that defines specific classes for various management domains (*e.g.* storage, networking, operating systems). The schema is expressed using the "Managed Object Format" (DSP0004).
- The **operations** that CIM clients may perform on CIM servers.
- The **protocols** enabling clients and servers to communicate (*e.g.* WS-Management, CIM-XML).

A software service that implements these standards is a **CIM Server**, also known as a **CIM Object Manager (CIMOM)**. For more information on these standards, visit the DMTF (Desktop Management Task Force) web site: `http://dmtf.org`.

Note: WBEM (Web-Based Enterprise Management) comprises several standards, including CIM (Common Information Model) and WS-Management (which is now specified in ISO/IEC 17963:2013). **WBEM** refers to the broader set of standards. For this reason, the server is named **OMI**.

1.2 What does a CIM Server do?

In general, a CIM server enables client applications to perform operations on managed resources, such as CPUs, disks, networks, and processes. Typical operations include:

- Enumerating resource instances.

- Invoking a method on a resource.
- Subscribing to events on a resources.

But CIM servers do not perform these operations on resources directly. Instead servers furnish developers with a framework for building pluggable modules called **providers**. Providers are modules that interact directly with one or more resources. For example, a "process provider" interacts with operating system processes. Providers are packaged as shared libraries with a main entry point (used by the server to initialize the provider).

1.3 Operations

OMI enables clients to perform the following CIM/WBEM operations on providers:

- **GetInstance** - gets a single instance from the server.
- **EnumerateInstances** - enumerates instances of a given CIM class.
- **CreateInstance** - creates an instance of a CIM class.
- **DeleteInstance** - deletes an instance.
- **ModifyInstance** - modifies the properties of an instance.
- **Associators** - finds instances associated with a given instance.
- **References** - finds references that refer to a given instance.
- **Invoke** - invokes a method on a given instance or class.
- **Subscribe** - subscribes to an Indication class or group of classes.

OMI clients initiate these operations through these protocols:

- The WS-Management protocol
- The local Binary protocol.
- The CIM-XML protocol (not supported yet).

The server accepts client requests and routes them to the appropriate provider. Provider responses are routed back to the requesting client.

1.4 License

OMI software is currently freely available for use by anyone under the terms of the Apache 2.0 license (`http://www.apache.org/licenses/LICENSE-2.0.txt`).

1.5 Supported Platforms

OMI supports the following platforms.

- HP-UX 11i v2 and v3 (PA-RISC and IA64)
- Oracle/Sun Solaris 9 (SPARC), Solaris 10 (SPARC/x86) and 11 (SPARC/x86)
- Red Hat Enterprise Linux Server 4 (x86/x64), 5 (x86/x64), and 6 (x86/x64)
- Novell SUSE Linux Enterprise Server 9 (x86), 10 (x86/x64), and 11 (x86/x64)
- CentOS Linux 5 (x86/x64) and 6 (x86/x64)
- Debian Linux 5 (x86/x64)
- IBM AIX v5.3, v.6.1 and v7.1 (POWER)

OMI also builds on Windows with a few functional limitations.

1.6 Server Footprint

OMI was expressly designed to work on very small systems. Conventional CIM servers are too large for embedded and mobile operating systems, but OMI will fit easily on these systems. Memory consumption is low, and the object size of the server, built using the `--favorsize` flag, is less than 350 kilobytes for a 32-bit target, and less than 425 kilobytes for a 64-bit target. You can achieve even smaller footprints by disabling features you don't need.

2 Building and Installing

This chapter explains how to build and install OMI. It assumes you have obtained the OMI source distribution (`omi-1.0.0.tar`).

Note: Throughout this manual, we use `omi-1.0.0` to represent the OMI version that you are using. In any of the examples provided, you should replace `omi-1.0.0` with your actual OMI version.

2.1 Prerequisites

OMI depends on the following software. Be sure these are installed on your system before building.

- GNU make
- Native C and C++ compiler
- OpenSSL headers and libraries
- PAM headers and libraries

2.2 Overview

The following commands build and install OMI (these steps are expounded in the sections below). Note again that you should replace `omi-1.0.0` with the release version of OMI that you are using.

```
# tar xf omi-1.0.0.tar
# cd omi-1.0.0
# ./configure
# make
# make install
```

The `make install` command installs all files under the `/opt/omi-1.0.0` directory.

2.3 Unpacking the source distribution

The OMI source distribution is a `tar` file. Unpack the distribution with the `tar` utility as follows:

```
# tar xf omi-1.0.0.tar
```

This command creates a directory named `omi-1.0.0`, which contains the source distribution.

2.4 Configuring the build

To configure the build, run the `./configure` script from the root of the source distribution. Type the following to print a help message explaining how to use the script.

```
# ./configure --help
```

This will also list any new options that have been added in recent releases of OMI.

The options allow you to change where components are installed. For example, to install the programs under `/usr/local/bin` and everything else under `/opt/omi`, configure as follows.

```
# ./configure --prefix=/opt/omi --bindir=/usr/local/bin
```

The default `prefix` is `/usr/omi-1.0.0`. After installing, you will find all OMI programs (with a `omi` prefix) under `/usr/local/bin`.

2.5 Configuring the build outside the source distribution

OMI 1.0.6 added support for configuring the build outside of the source distribution. To do this, create a build directory outside of the source distribution and then invoke the `configure` script from that build directory using a relative path. For example (assuming version 1.0.6):

```
# tar xvf omi-1.0.6.tar
# mkdir build
# cd build
# ../omi-1.0.6/configure
# make
```

2.6 Building the distribution

After configuring, build by typing `make`, where `make` refers to GNU make. For example:

```
# make
```

This builds all components.

2.7 Installing the distribution

After building the source distribution, install by typing:

```
# make install
```

You may configure and build as any user. But you must install as root since the install script creates files under root-owned directories. Even if the `--prefix` option specifies a non-root owned directory, the PAM authentication file (`omi.pam`) must be copied to a root-owned directory.

2.8 Uninstalling the distribution

Note that wherever OMI is installed, you will find a script called `omiuninstall`. This script removes all installed components, but leaves any third-party components (*e.g.* providers and registration files) intact.

2.9 Installing under DESTDIR

Sometimes it is useful to install all the components under a "DESTDIR" for the purposes of building an RPM or deploying the binaries to a target machine. For example, consider these steps:

```
$ ./configure --prefix=/opt/abc
$ make
$ make install DESTDIR=/tmp/destdir
```

So instead of installing under:

```
/opt/abc
```

OMI is installed under here instead:

```
/tmp/destdir/opt/abc
```

This procedure isolates all of the installable files for packaging or for building an install manifest.

Following the example above, a binary distribution for a given platform can be created as follows:

```
$ cd /tmp/destdir/
$ tar cvf omi-1.0.0-linux-x86.tar opt
```

Later this can be installed by simply un-tar-ing the package in the root directory of a Linux system.

2.10 Installation layout

After installing, you will find the installed files in the locations specified by the `./configure` options. For example, if you configured with `./configure --prefix=/opt/omi`, you will find the following files after installing.

```
/opt/omi/bin/omicli
/opt/omi/bin/omigen
/opt/omi/bin/omireg
/opt/omi/bin/omiserver
/opt/omi/bin/omiagent
/opt/omi/etc/ssl/certs/omi.pem
/opt/omi/etc/ssl/certs/omikey.pem
/opt/omi/etc/omicli.conf
/opt/omi/etc/omiregister/root-omi/omiidentify.reg
/opt/omi/etc/omigen.conf
/opt/omi/etc/omiserver.conf
/opt/omi/lib/libmicxx.so
/opt/omi/lib/libomiclient.so
/opt/omi/lib/libomiidentify.so
/opt/omi/share/omischema/CIM_Schema.mof
...
/opt/omi/share/omi.mak
/opt/omi/include/MI.h
/opt/omi/include/omiclient/handler.h
/opt/omi/include/omiclient/linkage.h
/opt/omi/include/omiclient/client.h
/opt/omi/include/micxx/atomic.h
/opt/omi/include/micxx/propertyset.h
/opt/omi/include/micxx/dinstance.h
/opt/omi/include/micxx/instance.h
/opt/omi/include/micxx/field.h
/opt/omi/include/micxx/context.h
/opt/omi/include/micxx/datetime.h
/opt/omi/include/micxx/micxx.h
```

```
/opt/omi/include/micxx/types.h
/opt/omi/include/micxx/arraytraits.h
/opt/omi/include/micxx/array.h
/opt/omi/include/micxx/linkage.h
/opt/omi/include/micxx/string.h
```

In addition to these files, the installer also copies a PAM (Pluggable Authentication Module) file called `omi.pam` under the `/etc/pam` directory.

The following sections discuss these installed files.

2.10.1 'bin'

The `bin` directory contains all OMI programs, including:

- `omiserver` – the server program.
- `omiagent` – the provider agent program.
- `omigen` – the provider generation tool.
- `omireg` – the provider registration tool.
- `omicli` – the command-line client tool.

2.10.2 'etc'

The `etc` directory contains system-wide configuration files used by various programs, including:

- `omicli.conf` – configuration file for omicli program.
- `omigen.conf` – configuration file for omigen program.
- `omiserver.conf` – configuration file for omiserver program.

The `omicli` and `omigen` programs look first for configuration files named `.omiclirc` and `.omigenrc` in the current and home directories (in which case the system-wide configuration file is ignored).

2.10.3 'lib'

The `lib` directory contains libraries. These include:

- `libmi.so` – The C-language MI API support library. `libmicxx.so` – the C++ provider support library (*Note that this library has been deprecated*).
- `libomiclient.so` – the C++ binary protocol client library.
- `libomiidentify.so` – the identify provider (OMI_Identify class).

2.10.4 'include'

The `include` directory contains C and C++ header files required for provider and client application development. These include:

- `MI.h` – C provider header file.
- `micxx/micxx.h` – main C++ provider header file (*Note that this has been deprecated*).
- `omiclient/client.h` – main C++ client header file.

2.10.5 'omischema'

The `omischema` directory contains MOF files that define the CIM schema. These files are used by the provider generator tool (`omigen`) while generating provider sources. The directory contains hundreds of MOF files. The main MOF file is called `CIM_Schema.mof` (which includes all others).

As of omi-1.0.8, the CIM schema version being used is CIM-2.32.0.

2.10.6 'etc/ssl/certs'

The `etc/ssl/certs` directory contains PEM-formatted certificates for SSL (private and public). These include:

- `omi.pem` – the public certificate.
- `omikey.pem` – the private certificate/key.

2.10.7 'etc/omiregister'

The `etc/omiregister` directory contains a *namespace directory* for each CIM namespace. Each namespace directory has the same name as the corresponding CIM namespace, except `/` characters are translated to `-` characters. For example, for the CIM namespace `root/cimv2`, there is a directory named `root-cimv2`. The server scans the `etc/omiregister` directory during startup to obtain a list of supported namespaces.

Each namespace directory contains provider registration files (with a `.reg` extension). Each registration file corresponds to a single provider library. These files are created by the `omireg` utility. The following registration file (named `omiidentify.reg`) registers a provider that implements the `OMI_Identify` class.

```
LIBRARY=omiidentify
CLASS=OMI_Identify
```

Placing this file in the `etc/omiregister/root-omi` directory, registers the provider for that namespace. The server scans all namespace directories to discover provider registrations during startup.

2.10.8 'share'

The `share` directory contains the `omi.mak` file. This file is included by provider makefiles generated by the `omigen` tool.

3 Using the server

This chapter explains how to use the server program. It explains how to start, validate, and stop the server. It also explains various options and where to find the log files.

3.1 Setting up your path

You may run each program by specifying its fully-qualified path, or for convenience, you may wish to add the `bin` directory to your path. The examples below assume you have done so.

3.2 Getting help

To get help with server options, type the following.

```
# omiserver -h
```

This prints a help message that explains the usage, arguments, and options.

3.3 Starting the server

To start the server in the foreground, type this.

```
# omiserver
```

To start the server in the background, use the `-d` (daemonize) option.

```
# omiserver -d
```

Multiple instances of the server may run on the same host subject to the following constraints:

- Each server is built with a distinct installation prefix, so that each server has a unique PID file and socket file paths.
- Each server binds to a distinct port. The port may be set with the `--port` command-line option or `port` configuration file option.

If these constraints are not met, attempting to run a second server results in an "already running" message.

3.4 Validating the server

To validate that the server is working correctly, use the `omicli` tool to send it a request. Type `omicli -h` for help with this tool. When initially installed, the server only has one provider, which provides the `OMI_Identify` class. To enumerate all instances of this class, type the following command (`id` is short for `identify`).

```
# omicli id
```

If the server is working properly, this command should print a single instance to standard output. For example (if the OMI version were 1.0.0):

```
instance of OMI_Identify
{
    [Key] InstanceID=2FDB5542-5896-45D5-9BE9-DC04430AAABE
    SystemName=linux
    ProductName=OMI 1.0.0
```

```
        ProductVendor=Microsoft
        ProductVersionMajor=1
        ProductVersionMinor=0
        ProductVersionRevision=0
        ProductVersionString=1.0.0
        Platform=LINUX_IX86_GNU
        OperatingSystem=LINUX
        Architecture=IX86
        Compiler=GNU
        ConfigPrefix=/tmp/omi
        ConfigLibDir=/tmp/omi/lib
        ConfigBinDir=/tmp/omi/bin
        ConfigIncludeDir=/tmp/omi/include
        ConfigDataDir=/tmp/omi/share
        ConfigLocalStateDir=/tmp/omi/var
        ConfigSysConfDir=/tmp/omi/etc
        ConfigProviderDir=/tmp/omi/etc
        ConfigLogFile=/tmp/omi/var/log/omiserver.log
        ConfigPIDFile=/tmp/omi/var/run/omiserver.pid
        ConfigRegisterDir=/tmp/omi/etc/omiregister
        ConfigSchemaDir=/tmp/omi/share/omischema
        ConfigNameSpaces={root-omi, interop, root-cimv2}
    }
```

This instance identifies various characteristics of the server and system.

3.5 Stopping the server

To stop the server, type the following.

```
# omiserver -s
```

This stops the server by sending a signal to the process whose process id (pid) is contained in `var/run/omiserver.pid`. The server removes this file when it shuts down.

3.6 Server and Agent Logs

To enable logging, start `omiserver` with the option: `-loglevel <level number>`. To generate HTTP `trc` files, start `omiserver` with the option: `-httptrace`. Without these options, logging and `trc` files will not be enabled.

Server log messages are directed to `var/log/omiserver.log`. The server spawns agent processes (`omiagent`) in order to run providers as specified users (determined by the provider hosting model). Log messages from agents are written to files whose name has the form:

```
var/log/omiagent.<UID>.<GID>.log
```

`<UID>` and `<GID>` are the user id and group id of the agent process's owner. To browse logs, look for files under `var/log` whose name matches `omi*`.

3.7 Server file and directory locations

Sometimes it is helpful to know where the server expects to find various files. Where is the server log file? Where is the provider registration directory? To obtain a list of server and file locations, type the following command.

```
# omiserver -p
```

Running this on a system where OMI was installed under `/opt/omi` prints the following.

```
prefix=/opt/omi
libdir=/opt/omi/lib
bindir=/opt/omi/bin
localstatedir=/opt/omi/var
sysconfdir=/opt/omi/etc
providerdir=/opt/omi/lib
certsdir=/opt/omi/etc/ssl/certs
datadir=/opt/omi/share
rundir=/opt/omi/var/run
logdir=/opt/omi/var/log
schemadir=/opt/omi/share/omischema
schemafile=/opt/omi/share/omischema/CIM_Schema.mof
pidfile=/opt/omi/var/run/omiserver.pid
logfile=/opt/omi/var/log/omiserver.log
registerdir=/opt/omi/etc/omiregister
pemfile=/opt/omi/etc/ssl/certs/omi.pem
keyfile=/opt/omi/etc/ssl/certs/omikey.pem
agentprogram=/opt/omi/bin/omiagent
serverprogram=/opt/omi/bin/omiserver
includedir=/opt/omi/include
configfile=/opt/omi/etc/omiserver.conf
socketfile=/opt/omi/var/omiserver.sock
```

4 Using the command-line client (‘omicli’)

This chapter explains how to use the command-line client tool. This tool sends requests to the local CIM server and prints the responses to standard output. For example, `omicli ei root/omi OMI_Identify` sends the `ei` request (`enumerate-instances`) to the server and then prints the following on standard output (assuming, here, that the OMI version is 1.0.0):

```
instance of OMI_Identify
{
    [Key] InstanceID=2FDB5542-5896-45D5-9BE9-DC04430AAABE
    SystemName=linux
    ProductName=OMI 1.0.0
    ProductVendor=Microsoft
    ProductVersionMajor=1
    ProductVersionMinor=0
    ProductVersionRevision=0
    ProductVersionString=1.0.0
    Platform=LINUX_IX86_GNU
    OperatingSystem=LINUX
    Architecture=IX86
    Compiler=GNU
    ConfigPrefix=/tmp/omi
    ConfigLibDir=/tmp/omi/lib
    ConfigBinDir=/tmp/omi/bin
    ConfigIncludeDir=/tmp/omi/include
    ConfigDataDir=/tmp/omi/share
    ConfigLocalStateDir=/tmp/omi/var
    ConfigSysConfDir=/tmp/omi/etc
    ConfigProviderDir=/tmp/omi/etc
    ConfigLogFile=/tmp/omi/var/log/omiserver.log
    ConfigPIDFile=/tmp/omi/var/run/omiserver.pid
    ConfigRegisterDir=/tmp/omi/etc/omiregister
    ConfigSchemaDir=/tmp/omi/share/omischema
    ConfigNameSpaces={root-omi, interop, root-cimv2}
}
```

Each `instance of { ... }` construct represents an instance of a CIM class. The braces contain properties and their values. Key properties are annotated with the `Key` qualifier.

4.1 General usage

The general usage of the tool is as follows:

```
# ./output/bin/omicli [OPTIONS] COMMAND ...
```

The available `OPTIONS` are:

```
-h, --help          Print a help message.
-q                  Operate quietly.
-t                  Enable diagnostic tracing.
-R N                Repeat this command N times.
```

```
-shallow            Use shallow inheritance (see 'ei' command).
-synchronous        Executes command in synchronous mode.
-ac CLASSNAME       Association class (see 'a' and 'r' commands).
-rc CLASSNAME       Result class (see 'a' command).
-r ROLE             Role (see 'a' and 'r' commands).
-rr ROLE            Result role (see 'a' command).
-n                  Show null properties.
-u USERNAME         Username.
-p PASSWORD         User's password.
-id                 Send identify request.
--socketfile PATH   Talk to the server server whose socket file resides
                    at the location given by the path argument.
--httpport          Connect on this port instead of default.
--httpsport         Connect on this secure port instead of default.
--querylang         Query language (for 'ei', 'sub' command).
--queryexpr         Query expression (for 'ei', 'sub' command).
```

COMMAND is one of the following:

```
noop
     Perform a no-op operation.
gi NAMESPACE INSTANCENAME
     Peform a CIM [g]et [i]nstance operation.
ci NAMESPACE NEWINSTANCE
     Peform a CIM [c]reate [i]nstance operation.
mi NAMESPACE MODIFIEDINSTANCE
     Peform a CIM [m]odify [i]nstance operation.
di NAMESPACE INSTANCENAME
     Peform a CIM [d]elete [i]nstance operation.
ei [-shallow] NAMESPACE CLASSNAME
     Peform a CIM [e]numerate [i]nstances operation.
iv NAMESPACE INSTANCENAME METHODNAME PARAMETERS
     Peform a CIM extrinisic method [i]nvocation operation.
a [-ac -rc -r -rr ] NAMESPACE INSTANCENAME
     Perform a CIM [a]ssociator instances operation.
r [-rc -r] NAMESPACE INSTANCENAME (references)
     Perform a CIM [r]eference instances operation.
gc NAMESPACE CLASSENAME
     Peform a CIM [g]et [c]lass operation.
enc INSTANCE
     Attempt to encode and print the given instance representation.
wql NAMESPACE WQLQUERY
     Peform a WQL query operation.
cql NAMESPACE CQLQUERY
     Peform a CQL query operation.
sub NAMESPACE
     Peform a subscribe to indication operation.
```

In a command, the INSTANCENAME and PARAMETERS formats are:

```
{ class_name property_name property_value property_name property_value }
```

Here, `property_value` can be a string value, an `INSTANCENAME`, or an array in the form:

```
[ property_value property_value ]
```

The following sections provide some examples of how to use the command-line client.

4.2 Getting help

To print a help message that lists the options and commands, type the following:

```
# ./output/bin/omicli -h
```

4.3 The socket file

By default, when `omicli` and `omiserver` are built together (with the same prefix), the `omicli` program contains the location of the server's socket file. But when they are built separately, or if you want to communicate with multiple instances of the server, you must specify the socket file location using the `--socketfile` option. The socket file is located here: `*/var/run/omiserver.sock`, where `*` is the prefix the server was built with.

4.4 The No-Op request

The following command sends a no-op request to the server.

```
# ./output/bin/omicli noop
```

This tests whether the server is running and responsive. If so, it prints a message indicating success. If the server is not responsive, the command will time out.

4.5 Enumerating Instances

The following command enumerates instances of `OMI_Identify` within the `root/omi` namespace.

```
# ./output/bin/omicli ei root/omi OMI_Identify
```

4.6 Getting an Instance

The following command gets a single instance of the `OMI_Identify` class from the `root/omi` namespace.

```
# ./output/bin/omicli gi root/omi \
  { OMI_Identify InstanceID 2FDB5542-5896-45D5-9BE9-DC04430AAABE }
```

The expression in braces represents the **instance name** for the given instance. This instance name has a single key, although an instance name may have multiple keys. Consider, for example, the following class.

```
class MyClass
{
    [Key] String  Key1;
    [Key] Uint32  Key2;
    [Key] Boolean Key3;
    ...
};
```

And now consider the following instance of that class.

```
instance of MyClass
{
    Key1=XYZ
    Key2=123
    Key3=false
    ...
};
```

The instance name for this instance is expressed as follows on the command line.

```
{ MyClass Key1 XYZ Key2 123 Key3 false }
```

In general, instance names are expressed as a class name followed by name-value pairs, all enclosed in brackets. Failing to specify values for a complete set of keys results in an error.

4.7 Invoking a method

To invoke an extrinsic method, use the `iv` command, whose usage is:

```
# ./output/bin/omicli iv NAMESPACE INSTANCENAME METHODNAME PARAMETERS
```

For example, consider the `SetState` extrinsic method defined by the following CIM class.

```
class OMI_Frog
{
    [Key] Uint32 Key;

    Uint32 SetState(
        [In] String NewState,
        [In(false), Out] String OutState);
};
```

The following command invokes the `SetState` method on the instance of `OMI_Frog` named `{ OMI_Frog Key 123 }`.

```
# ./output/bin/omicli iv root/omi \
  { OMI_Frog Key 123 } SetState { NewState Hopping }
```

This command prints any output parameters to standard output. For example, the above command might print this:

```
{ OldState Sitting }
```

4.8 Subscribing to an Indication

Use the `queryexpr="select"` statement to subscribe to an indication, as illustrated below.

Note: On single threaded systems running in-proc providers and performing a multi-class subscription, the response may be somewhat misleading. If the first indication class takes longer to subscribe than the `OperationTimeout`, the client always receives an `OperationTimeout` error. If the first indication class's subscription succeeds, however, then the client receives a response indicating success even if the subscriptions of other classes time out and fail thereafter.

4.8.1 Subscribing to an alert Indication

An alert indication class derives from the standard CIM_Indication class. For instance:

```
class XYZ_DiskFault : CIM_Indication
{
  string detailmessage;
};
```

In order to subscribe to this XYZ_DiskFault indication class, run:

```
# ./output/bin/omicli sub root/sample \
  -queryexpr="select * from XYZ_DiskFault"
```

4.8.2 Subscribing to a lifecycle Indication

Consider as an example a class derived from CIM_Process:

```
class XYZ_Process : CIM_Process
{
  uint32 runningTime;

  [static]
  uint32 Create( [in] string imageName, [out] CIM_Process REF process );
  uint32 GetRunTime( [in] uint32 pid, [out] uint32 runningTime );
};
```

To subscribe to the lifecycle creation indication for instances of the XYZ_Process indication class, run:

```
# ./output/bin/omicli sub root/sample \
  -queryexpr="select * from CIM_InstCreation \
              where SourceInstance ISA XYZ_Process"
```

Note: A lifecycle subscription query must contain one and only one "ISA" operator specifying the target class name of the subscription filter. In addition, OMI does not support the use of an OR operator as an ancestor node of the ISA operator in the WHERE clause of a query, because this would create too complex a filter.

5 Using the client library 'omiclient'

The omiclient client library defines a C++ API that enables client applications to connect to and send requests to the CIM server. It uses a local binary protocol for communicating with the CIM server. As a result, the client application must reside on the same host as the CIM server. This chapter explains how to get started using this library.

Please note, however, that the omiclient client library is deprecated and may not be supported in future releases.

5.1 Client library source examples

The following source files (under the source distribution) illustrate how to use the client library.

```
./omiclient/tests/test_client.cpp
./cli/cli.cpp
```

The first is the unit test for the omiclient library. The second is the main source file of the omicli tool discussed above. The examples below show how to do simple things with the client library. For more detail, see these examples.

5.2 The 'omiclient' library

The base name of client library is omiclient. The full name is platform dependent. For example, on Linux the full name is libomiclient.so. This library resides in the lib directory (selected when the distribution was built). The client application must be linked with this library.

5.3 The '<omiclient/client.h>' header

Client applications must include <omiclient/client.h>. This header file defines the full client interface. It resides in the include directory (selected when the distribution was built).

5.4 Connecting to the local server

The first step is to connect to the local CIM server, illustrated by the following program.

```
01  #include <omiclient/client.h>
02
03  using namespace std;
04
05  int main()
06  {
07      const Uint64 timeout = 30000000;
08
09      Client c;
10      String locator;
11      String username;
12      String password;
13
14      if (!c.Connect(locator, username, password, timeout))
```

```
15      {
16          // Error!
17      }
18
29      return 0;
20  }
```

Line 1 includes `<omiclient/client.h>`, the main header file for the client interface. This header is located under the installation `include` directory. Consult this file to more details about the interface.

Line 9 instantiates an instance of the `Client` class. This function takes an optional `Handler` instance, which is required when calling the asynchronous member functions. The examples below use the synchronous methods and so no handler is needed.

Line 14 establishes a synchronous connection with the local CIM server. The `Client::Connect` function takes four arguments: `locator`, `username`, `password`, and `timeout`. The `locator` is a string that specifies the Unix domain socket file used to connect to the server. If the `locator` is empty, the client uses the default socket file, located under the `run` directory with the name `omiserver.sock`.

To connect to a server whose socket file is not in the default location, set the location parameter to the the full path of that socket file, such as `/opt/omi/var/run/omiserver.sock`.

The `username` and `password` parameters are used to authenticate the user with the server. There are two kinds of authentication: explicit and implicit. With **explicit** authentication, the `username` and `password` parameters hold the log on credentials for a given user. With **implicit** authentication, these parameters are empty, in which case the user is authenticated using the identity of the current user (obtained with the `getuid` system call).

The `timeout` parameter specifies how long to wait (in microseconds) before failing. In this example, the timeout is 30 seconds (30,000,000 microseconds).

Developers may call the `Client::Disconnect()` method to explicitly disconnect from the server. Otherwise, the connection is closed implicitly by the `Client` destructor.

5.5 Enumerating instances

The code fragment below enumerates instances of the `OMI_Identify` class.

```
01      const String nameSpace = "root/omi";
02      const String className = "OMI_Identify";
03      const bool deepInheritance = true;
04      const Uint64 timeout = 2000000;
05      Array<DInstance> instances;
06      MI_Result result;
07
08      if (!c.EnumerateInstances(nameSpace, className, deepInheritance,
09          timeout, instances, result))
10      {
11          // Error!
12      }
13
```

```
14      if (result != MI_RESULT_OK)
15      {
16          // Error!
17      }
18
19      for (Uint32 i = 0; i < instances.GetSize(); i++)
20          instances[i].Print();
```

Line 8 performs an enumeration request. It obtains instances of the given class from the given namespace. The resulting instances are in the `instances` parameter upon return.

The `deepInheritance` parameter specifies whether to return instances of classes derived from `OMI_Identify` (if true) or to return instances of `OMI_Identify` only (if false).

Line 19 through 20 print the resulting instances.

Use `EnumerateInstances` with regard for memory usage. It places all instances into memory at once, which may exhaust available memory when there are many thousands of instances. To avoid memory exhaustion, use the asynchronous form, called `EnumerateInstancesAsync`. All asynchronous functions use the `Handler` class, which defines virtual functions for delivering instances one at a time. See Appendix B for a fully asynchronous example.

5.6 Getting a single instance

The code fragment below shows how to get a single instance from the server.

```
01  // Construct an instance name:
02  const String className = "OMI_Identify";
03  DInstance instanceName(className, DInstance::CLASS);
04
05  // Add a key property:
06  const String propertyName = "InstanceID";
07  const String instanceID = "2FDB5542-5896-45D5-9BE9-DC04430AAABE";
08  const String isNull = false;
09  const String isKey = true;
10  instanceName.AddUint32(propertyName, instanceID, isNull, isKey);
11
12  // Perform get instance:
13  const String nameSpace = "root/omi";
14  const Uint64 TIMEOUT = 2000000;
15  DInstance instance;
16  MI_Result result;
17  if (!c.GetInstance(nameSpace, instanceName, TIMEOUT, instance,
18      result))
19  {
20      // Error!
21  }
22
23  if (result != MI_RESULT_OK)
24  {
```

```
25      // Error!
26  }
27
28  instance.Print();
```

Lines 1 through 10 build an instance name (using the `DInstance` class). Lines 2 through 3 construct a `DInstance` whose class name is `OMI_Identify`. Lines 6 through 10 add a key property to the instance name called `InstanceID`.

Lines 13 through 18 perform the get instance request. Upon return, the `instance` parameter contains the result.

Finally, Line 28 prints the resulting instance to standard output.

5.7 Invoking an extrinsic method

This section shows how to invoke an extrinsic method. Recall the definition of the `OMI_Frog` class.

```
class OMI_Frog
{
    [Key] Uint32 Key;

    Uint32 SetState(
        [In] String NewState,
        [In(false), Out] String OutState);
};
```

The code fragment shows how to invoke the `OMI_Frog.SetState` method.

```
01  // Initialize instance name:
02  DInstance instanceName("OMI_Frog", DInstance::CLASS);
03  instanceName.AddUint32("Key", 123, false, true);
04
05  // Initialize input parameters:
06  DInstance in(T("SetState"), DInstance::METHOD);
07  in.AddString(T("NewState"), "Hopping", false, false);
08
09  // Invoke method:
10  const Uint64 TIMEOUT = 5000000;
11  DInstance out;
12  MI_Result result;
13
14  if (!c.Invoke(
15      "root/omi",
16      instanceName,
17      "SetState",
18      in,
19      TIMEOUT,
20      out,
21      result))
22  {
```

```
23      // Error!
24  }
25
26  if (result != MI_RESULT_OK)
27  {
28      // Error!
29  }
30
31  out.Print();
```

Lines 2 through 3 initialize the instance name, which identifies the instance of `OMI_Frog` whose method will be called. This specifies a single key named `Key` with the value `123`.

Lines 6 through 7 initialize the input parameters for the method. In this example, there is a single parameter named `NewState` with the value `Hopping`.

Lines 14 through 21 invoke the `SetState` method. Upon return, `out` holds any output parameters (the `OldState` parameter in this case). The `out` parameter always has a parameter named `ReturnValue`, which contains the return value of the function (the return value of `OMI_Frog.SetState` in this example). In CIM, all functions are required to return a value.

Note: *Breaking change in OMI 1.0.8*: **'MIReturn' is now 'ReturnValue'**. This affects all clients (including providers) which invoke a method and read the return value from the invocation result instance. In OMI 1.0.7 and before, a client may invoke a method using the `libmicxx.so` library, and read the return value in the `MIReturn` property of the result instance. In OMI 1.0.8 and future releases, the client must read the return value in the `ReturnValue` property instead.

6 Using MI 'miapi' and/or 'libmi.so'

6.1 Introduction to the MI Client Library

MI on Windows is the Windows Management Infrastructure, also known as WMIv2. It is documented online in the MSDN Library, at http://msdn.microsoft.com/en-us/library/jj152383.aspx.

For OMI, the miapi client library can be used by including MI.h in your C source file:

```
#include <MI.h>
```

and by adding the following flags in the GNUmakefile:

```
-L$Omi/lib -lmi
```

6.1.1 CDXML

MI introduces CDXML (cmdlet-definition XML), an XML schema for mapping Windows PowerShell cmdlets to CIM class operations or methods. PowerShell cmdlet developers use CDXML files to define cmdlets that call a CIM Object Manager (CIMOM) server such as WMI in Windows to manage the server. Cmdlets defined in CDXML communicate with remote CIM servers using the WsMan protocol, implemented by the WinRM service in Windows. This enables management of end-points that don't have PowerShell installed on them, such as a computer running Windows Server with PowerShell remoting disabled, or a computer running a CIM server like OMI or other CIM server software.

For documentation of CDXML, see http://msdn.microsoft.com/en-us/library/jj542522.aspx.

6.1.2 The MI Application Programming Interface (API)

The Management Infrastructure API contains the interfaces, enumerations, structures, and unions used to developer native WMI providers and clients. It is documented online at http://msdn.microsoft.com/en-us/library/hh404805(v=vs.85).aspx.

6.2 Samples illustrating MI library operations

Examples of MI operations performed using the OMI libraries are located in the OMI distribution in the samples/MIAPI folder. In general, these examples show how to perform each operation both synchronously and asynchronously.

The synchronous example generally calls MI_Operation_GetInstance in a loop to iterate through the results of the operation, whereas the asynchronous example initializes a MI_OperationCallbacks structure with the event and callbacks needed to process the results asynchronously.

The following operations are illustrated:

- **GetInstance** - The get.c file shows how to use MI_Session_GetInstance.
- **EnumerateInstances** - The enumerate.c file shows how to use MI_Session_EnumerateInstances.
- **CreateInstance** - The create.c file shows how to use MI_Session_CreateInstance.

- **DeleteInstance** - The `delete.c` file shows how to use `MI_Session_DeleteInstance`.
- **ModifyInstance** - The `modify.c` file shows how to use `MI_Session_ModifyInstance`.
- **Associators** - The `association.c` file shows how to use `MI_Session_AssociatorInstances`.
- **Invoke** - The `invoke.c` file shows how to use `MI_Session_Invoke`.
- **References** - The `reference.c` file shows how to use `MI_Session_ReferenceInstances`.
- **Subscribe** - The `subscribe.c` file shows how to use `MI_Session_Subscribe`.

7 Developing a provider in 5 minutes

This chapter provides a very quick overview of the provider development process. It shows the minimum steps for building a simple instance provider. For a more complete discussion of provider development, see the next chapter.

Appendix A contains a complete listing of all file in this example. These files are also included in the source distribution under `omi-1.0.0/doc/omi/samples/frog` (assuming OMI version 1.0.0).

7.1 Defining 'schema.mof'

First we define the class schema shown in the `schema.mof` file below.

```
class XYZ_Frog
{
    [Key] String Name;
    Uint32 Weight;
    String Color;
};
```

7.2 Generating the provider sources

Next we generate the provider sources and the makefile using the command below.

```
someuser@linux:~/gadget> omigen -m frog schema.mof XYZ_Frog
Creating XYZ_Frog.h
Creating XYZ_Frog.c
Checking XYZ_Frog.c
Creating schema.c
Creating module.c
Creating GNUmakefile
```

7.3 Implementing the 'EnumerateInstances' stub

Next we implement the `EnumerateInstances` stub. The generated stub looks like this:

```
void MI_CALL XYZ_Frog_EnumerateInstances(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

The implementation below provides two frogs.

```
void MI_CALL XYZ_Frog_EnumerateInstances(
    XYZ_Frog_Self* self,
```

```
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    XYZ_Frog frog1;
    XYZ_Frog_Construct(&frog1, context);
    XYZ_Frog_Set_Name(&frog1, MI_T("Fred"));
    XYZ_Frog_Set_Weight(&frog1, 55);
    XYZ_Frog_Set_Color(&frog1, MI_T("Blue"));
    XYZ_Frog_Post(&frog1, context);
    XYZ_Frog_Destruct(&frog1);

    XYZ_Frog frog2;
    XYZ_Frog_Construct(&frog2, context);
    XYZ_Frog_Set_Name(&frog2, MI_T("Sam"));
    XYZ_Frog_Set_Weight(&frog2, 65);
    XYZ_Frog_Set_Color(&frog2, MI_T("Blue"));
    XYZ_Frog_Post(&frog2, context);
    XYZ_Frog_Destruct(&frog2);

    MI_Context_PostResult(context, MI_RESULT_OK);
}
```

Note: In CIM, a property either has a value or is null (has no value). The `XYZ_Frog->Name.exists` property is true if the `Frog.Name` property has a value or false if the property is null (has no value). If the property has a value, one may use the `XYZ_Frog->Name.value` function to obtain it.

7.4 Registering the provider

Next we register the provider as follows:

```
# make reg
///usr/local/bin/omireg /root/sampleprov/libfrog.so
    Created /opt/omi/lib/libfrog.so
    Created /opt/omi/etc/omiregister/root-cimv2/frog.reg
```

This creates `frog.reg` under the registration directory for the default namespace `root/cimv2` and it copies the provider to the installed directory.

7.5 Testing the provider

To test the provider, send an enumerate request to the provider as shown below.

```
# omicli ei root/cimv2 XYZ_Frog
instance of XYZ_Frog
```

```
{
    [Key] Name=Fred
    Weight=55
    Color=Green
}
instance of XYZ_Frog
{
    [Key] Name=Sam
    Weight=65
    Color=Blue
}
```

7.6 Going further

While this chapter has given a brief overview of the provider development process, the next chapter goes into provider development in more detail.

8 Developing providers

This chapter shows how to develop providers, a process consisting of 6 stages:

- Defining the MOF schema
- Generating the provider sources
- Implementing the provider operations
- Building the provider
- Registering the provider
- Validating the provider

We discuss each stage, showing how to develop providers that implement the following operations:

- **get-instance**
- **enumerate-instances**
- **associator-names**
- **reference-names**
- **invoke-method**

OMI supports two provider language bindings: C and C++. The C++ binding is now deprecated, and this chapter only shows how to use the C binding. Whether you build C or C++ providers, the development stages are the same, although the details of the interface vary.

8.1 Defining the MOF schema

The first stage is to define the MOF classes comprising your schema. You may extend an existing CIM class like this:

```
class XYZ_MyComputerSystem : CIM_ComputerSystem
{
    ...
};
```

Or you may define a new root class (with no super class):

```
class MyClass
{
    ...
};
```

The MOF language is defined in the **CIM Infrastructure Specification (DSP0004)**, which may be found at (`http://dmtf.org/standards/cim`).

The provider developed below implements the following class definitions (which are placed in a file called `schema.mof`).

```
// schema.mof

class XYZ_Widget
{
    [Key] Uint32 Key;
```

```
    Uint32 ModelNumber;
    String Color;
};

class XYZ_Gadget
{
    [Key] Uint32 Key;
    Uint32 ModelNumber;
    Uint32 Size;

    Uint32 ChangeState(
        [In] Uint32 NewState,
        [In(False), Out] Uint32 OldState);
};

[Association]
class XYZ_Connector
{
    [Key] XYZ_Widget REF Left;
    [Key] XYZ_Gadget REF Right;
};
```

Notice that all classes define above have the `XYZ_` prefix. Similarly, all classes in the CIM schema begin have the `CIM_` prefix. All classes should have a suitable prefix but for brevity, this prefix is omitted henceforth.

The `Connector` class is an association, as indicated by the `Associator` qualifier. Each instance of `Connector`, connects one instance of `Widget` with one instance of `Gadget`.

8.2 Generating the provider sources

The second stage involves generating the provider sources. The following command generates provider sources from the `schema.mof` file defined above.

```
omigen -m xyzconnector schema.mof XYZ_Gadget=Gadget \
       XYZ_Widget=Widget XYZ_Connector=Connector
Creating Gadget.h
Creating Gadget.c
Patching Gadget.c
Creating Widget.h
Creating Widget.c
Checking Widget.c
Creating Connector.h
Creating Connector.c
Checking Connector.c
Creating schema.c
Creating module.c
Creating GNUmakefile
```

The `-m xyzconnector` option creates `GNUmakefile` with rules for building, regenerating, and registering the provider. This makefile creates a library whose base name is given by the `-m` option (`xyzconnector`).

> **Tip:** You may regenerate sources by typing `make gen` or by retyping the command above. The generator will never overwrite editable files. Instead it attempts to patch them. Some files are non-editable and are regenerated completely.

The generator creates sources for classes `XYZ_Gadget`, `XYZ_Widget`, and `XYZ_Connector`. The command above defines aliases for each class name, allowing shorter names to be used throughout the source code. For example, `XYZ_Gadget=Gadget` causes `Gadget.h` to be generated instead of `XYZ_Gadget.h`. Alias can be used to completely rename a class. For example: `CIM_ComputerSystem=CompSys`.

To learn more about the `omigen` options, type `omigen -h` to print a help message.

The purpose of each generated file is given below.

- `Gadget.h` – defines C declarations and interfaces for the CIM `Gadget` class.
- `Widget.h` – defines C declarations and interfaces for the CIM `Widget` class.
- `Connector.h` – defines C declarations and interfaces for the CIM `Connector` class.
- `Gadget.c` – defines the implementation for the `CIM_Gadget` class.
- `Widget.c` – defines the implementation for the `CIM_Widget` class.
- `Connector.c` – defines the implementation for the `CIM_Connector` class.
- `schema.c` – internal definitions.
- `module.c` – defines `MI_Main` library entry point and provider load/unload implementation.
- `GNUmakefile` – defines rules for building the provider library.

Many of these files are not intended to be edited. Developer edits may be made to the following files.

- `Gadget.c`
- `Widget.c`
- `Connector.c`
- `module.c`

For example, to implement the `get-instances` operation for the `Gadget` class, modify the `Gadget.c` file.

8.3 Implementing the provider operations

This section shows how to implement the following provider operations:

- **get-instance**
- **enumerate-instances**
- **associator-names**
- **reference-names**
- **invoke-method**

8.3.1 Implementing enumerate-instances

This section implement the enumerate-instances operation for the `Gadget` class. This implementation provides the following instances (shown in MOF format).

```
instance of XYZ_Gadget
{
    Key = 1003;
    ModelNumber = 3;
    Size = 33;
};

instance of XYZ_Gadget
{
    Key = 1004;
    ModelNumber = 4;
    Size = 43;
};
```

To implement the enumerate-instance operation for the `Gadget` class, start by examining the generated stub (see `Gadget.c`).

```
void MI_CALL Gadget_EnumerateInstances(
    Gadget_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

This function is invoked by the CIM server. The implementer may respond on the same thread or create a new thread if the request is long running. The lifetime of the request is bound to the lifetime of the `context` parameter. All parameters remain in scope until the provider calls `MI_Context_PostResult`. The provider may create a new thread to handle the request, in which case the parameters may live beyond the invocation of `EnumerateInstances`.

We provide an implementation that provides two instances of the `Gadget` class. The following implementation handles the request on the calling thread.

```
void MI_CALL Gadget_EnumerateInstances(
    Gadget_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
```

```
{
    // Gadget.Key=1003:
    Gadget gadget1;
    Gadget_Construct(&gadget1, context);
    Gadget_Set_Key(&gadget1, 1003);
    Gadget_Set_ModelNumber(&gadget1, 3);
    Gadget_Set_Size(&gadget1, 33);
    Gadget_Post(&gadget1, context);
    Gadget_Destruct(&gadget1);

    // Gadget.Key=1004:
    Gadget gadget2;
    Gadget_Construct(&gadget2, context);
    Gadget_Set_Key(&gadget2, 1004);
    Gadget_Set_ModelNumber(&gadget2, 4);
    Gadget_Set_Size(&gadget2, 43);
    Gadget_Post(&gadget2, context);
    Gadget_Destruct(&gadget2);

    MI_Context_PostResult(context, MI_RESULT_OK);
}
```

This function constructs instances of the `Gadget` class and passes them to the `Gadget_Post` function. When all instances have been posted, the provider passes the result status to the `MI_Context_PostResult` function. This finalizes the request.

Tip: By passing the `--nogi CLASSNAME` option (no get-instance) to the generator tool, the server uses the `enumerate-instances` implementation to satisfy all `get-instance` requests. Only use this technique if the number of instances is small, otherwise the provider will be very slow.

8.3.2 Implementing get-instance

Next we show how to implement the `get-instance` operation for the `Gadget` class. Here is the generated stub for the `get-instance` request.

```
void MI_CALL Gadget_GetInstance(
    Gadget_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Gadget* instanceName,
    const MI_PropertySet* propertySet)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

The `instanceName` parameter holds the keys for the target instance. Here is the full implementation of this function.

```
void MI_CALL Gadget_GetInstance(
    Gadget_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Gadget* instanceName,
    const MI_PropertySet* propertySet)
{
    MI_Result r = MI_RESULT_OK;
    Gadget g;
    Gadget_Construct(&g, context);
    if (instanceName->Key.value == 1003)
    {
        // Gadget.Key=1003:
        Gadget_Set_Key(&g, 1003);
        Gadget_Set_ModelNumber(&g, 3);
        Gadget_Set_Size(&g, 33);
        Gadget_Post(&g, context);
    }
    else if (instanceName->Key.value == 1004)
    {
        // Gadget.Key=1004:
        Gadget_Set_Key(&g, 1004);
        Gadget_Set_ModelNumber(&g, 4);
        Gadget_Set_Size(&g, 43);
        Gadget_Post(&g, context);
    }
    else
    {
        r = MI_RESULT_NOT_FOUND;
    }

    Gadget_Destruct(&g);
    MI_Context_PostResult(context, r);
}
```

We examine the key and return the matching instance. If neither condition matches, we post the `MI_RESULT_NOT_FOUND` result.

8.3.3 Implementing an extrinsic method

This section implements the `ChangeState` extrinsic method. The generator produces the following stub.

```
void MI_CALL Gadget_Invoke_ChangeState(
    Gadget_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
```

```
    const MI_Char* methodName,
    const Gadget* instanceName,
    const Gadget_ChangeState* in)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

The `instanceName` parameter is the instance whose `ChangeState` method has been invoked (this parameter is omitted for static methods). The `in` parameter contains the input parameters. The implementation should perform the following tasks:

- Read the input parameters.
- Perform the desired action.
- Build the output parameters.
- Set the return value.
- Post the output parameters to the server.
- Return a successful status.

The following implementation performs each of these steps.

```
void MI_CALL Gadget_Invoke_ChangeState(
    Gadget_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_Char* methodName,
    const Gadget* instanceName,
    const Gadget_ChangeState* in)
{
    Gadget_ChangeState out;
    Gadget_ChangeState_Construct(&out, context);

    // Print the input parameter:
    if (in->NewState.exists)
    {
        printf("NewState=%u\n", in->NewState.value);
    }

    // Perform desired action here
    // ...

    // Set the output parameter
    Gadget_ChangeState_Set_OldState(&out, 2);

    // Set the return value
    Gadget_ChangeState_Set_MIReturn(&out, MI_RESULT_OK);

    // Post the output parameters
    Gadget_ChangeState_Post(&out, context);
```

```
    Gadget_ChangeState_Destruct(&out);

    MI_Context_PostResult(context, MI_RESULT_OK);
}
```

8.3.4 Implementing enumerate-instances for an association provider

This section shows how to implement the enumerate-instances operation for the `Connector` association class. This operation produces the following instances (shown in MOF format).

```
instance of XYZ_Connector
{
    Left = "XYZ_Widget.Key=1001";
    Right = "XYZ_Gadget.Key=1003";
};

instance of XYZ_Connector
{
    Left = "XYZ_Widget.Key=1002";
    Right = "XYZ_Gadget.Key=1004";
};
```

Here is the implementation.

```
void MI_CALL Connector_EnumerateInstances(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    Widget widget;
    Gadget gadget;
    Connector connector;

    Widget_Construct(&widget, context);
    Gadget_Construct(&gadget, context);
    Connector_Construct(&connector, context);

    // Connector.Left="Widget.Key=1001",Right="Gadget.Key=1003"
    Widget_Set_Key(&widget, 1001);
    Gadget_Set_Key(&gadget, 1003);
    Connector_Set_Left(&connector, &widget);
    Connector_Set_Right(&connector, &gadget);
    Connector_Post(&connector, context);
```

```
    // Connector.Left="Widget.Key=1002",Right="Gadget.Key=1004"
    Widget_Set_Key(&widget, 1002);
    Gadget_Set_Key(&gadget, 1004);
    Connector_Set_Left(&connector, &widget);
    Connector_Set_Right(&connector, &gadget);
    Connector_Post(&connector, context);

    Widget_Destruct(&widget);
    Gadget_Destruct(&gadget);
    Connector_Destruct(&connector);

    MI_Context_PostResult(context, MI_RESULT_OK);
}
```

8.3.5 Implementing get-instances for an association class

The following example implements `get-instance` for the `Connector` association class.

```
void MI_CALL Connector_GetInstance(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Connector* instanceName,
    const MI_PropertySet* propertySet)
{
    MI_Result r = MI_RESULT_OK;
    Widget widget;
    Gadget gadget;
    Connector connector;

    Widget_Construct(&widget, context);
    Gadget_Construct(&gadget, context);
    Connector_Construct(&connector, context);

    if (instanceName->Left.value->Key.value == 1001 &&
        instanceName->Right.value->Key.value == 1003)
    {
        // Connector.Left="Widget.Key=1001",Right="Gadget.Key=1003"
        Widget_Set_Key(&widget, 1001);
        Gadget_Set_Key(&gadget, 1003);
        Connector_Set_Left(&connector, &widget);
        Connector_Set_Right(&connector, &gadget);
        Connector_Post(&connector, context);
    }
    else if (instanceName->Left.value->Key.value == 1002 &&
        instanceName->Right.value->Key.value == 1004)
    {
```

```
            // Connector.Left="Widget.Key=1001",Right="Gadget.Key=1003"
            Widget_Set_Key(&widget, 1002);
            Gadget_Set_Key(&gadget, 1004);
            Connector_Set_Left(&connector, &widget);
            Connector_Set_Right(&connector, &gadget);
            Connector_Post(&connector, context);
        }
        else
        {
            r = MI_RESULT_NOT_FOUND;
        }

        Widget_Destruct(&widget);
        Gadget_Destruct(&gadget);
        Connector_Destruct(&connector);

        MI_Context_PostResult(context, r);
    }
```

Note that `instanceName->Left.value` returns an instance of type `Gadget` (see MOF definitions). And so `instanceName->Left.value->Key.value` returns the `Key` property of the associated `Gadget` instance.

8.3.6 Implementing the associator-instances operation

This section shows how to implement the associator-instances operation. This operation finds the other end of an association. For example, it might find the `Gadget` instances that are associated with a single `Widget` instance (through a `Connector` instance). For example, consider the following MOF definitions.

```
instance of XYZ_Connector
{
    Left  = "XYZ_Widget.Key=1001";
    Right = "XYZ_Gadget.Key=1003";
};

instance of XYZ_Connector
{
    Left  = "XYZ_Widget.Key=1002";
    Right = "XYZ_Gadget.Key=1004";
};
```

The associator-instances operation starts with the instance name of an instance and finds associated instances. For example, the associator-instances of:

```
XYZ_Widget.Key=1001
```

include:

```
XYZ_Gadget.Key=1003
```

The generator produces two stubs to handle associator-instances requests. The first yields associators in which the `instanceName` parameter matches the first reference property

(`Connector.Left`). The second yields associations in which the `instanceName` parameter matches the second reference property (`Connector.Right`). The stubs are defined as follows.

```
void MI_CALL Connector_AssociatorInstancesLeft(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Widget* instanceName,
    const MI_Char* resultClass,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}

void MI_CALL Connector_AssociatorInstancesRight(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Gadget* instanceName,
    const MI_Char* resultClass,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

The following implementation yields associators of `Widget` instances.

```
void MI_CALL Connector_AssociatorInstancesLeft(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Widget* instanceName,
    const MI_Char* resultClass,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Result r = MI_RESULT_OK;
    Gadget gadget;
    Gadget_Construct(&gadget, context);
```

```
    if (instanceName->Key.value == 1001)
    {
        // Gadget.Key=1003
        Gadget_Set_Key(&gadget, 1003);
        Gadget_Set_ModelNumber(&gadget, 3);
        Gadget_Set_Size(&gadget, 33);
        Gadget_Post(&gadget, context);
    }
    else if (instanceName->Key.value == 1002)
    {
        // Gadget.Key=1004
        Gadget_Set_Key(&gadget, 1004);
        Gadget_Set_ModelNumber(&gadget, 4);
        Gadget_Set_Size(&gadget, 43);
        Gadget_Post(&gadget, context);
    }
    else
    {
        r = MI_RESULT_NOT_FOUND;
    }

    Gadget_Destruct(&gadget);
    MI_Context_PostResult(context, r);
}
```

8.3.7 Implementing the reference-instances operation

The reference-instances operation takes an instance name and finds the reference instances that refer to it. Consider the following MOF definitions.

```
instance of XYZ_Connector
{
    Left  = "XYZ_Widget.Key=1001";
    Right = "XYZ_Gadget.Key=1003";
};

instance of XYZ_Connector
{
    Left  = "XYZ_Widget.Key=1002";
    Right = "XYZ_Gadget.Key=1004";
};
```

For example, the reference-instance of `XYZ_Widget.Key=1001` includes the first MOF instance shown above. As with associator-instances, the generator produces two stubs:

```
void MI_CALL Connector_ReferenceInstancesLeft(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
```

```
    const Widget* instanceName,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}

void MI_CALL Connector_ReferenceInstancesRight(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Gadget* instanceName,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

The implementation of `ReferenceInstancesLeft` is shown below.

```
void MI_CALL Connector_ReferenceInstancesLeft(
    Connector_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const Widget* instanceName,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    MI_Result r = MI_RESULT_OK;
    Widget widget;
    Gadget gadget;
    Connector connector;

    Widget_Construct(&widget, context);
    Gadget_Construct(&gadget, context);
    Connector_Construct(&connector, context);

    if (instanceName->Key.value == 1001)
    {
        // Connector.Left="Widget.Key=1001",Right="Gadget.Key=1003"
        Widget_Set_Key(&widget, 1001);
        Gadget_Set_Key(&gadget, 1003);
        Connector_Set_Left(&connector, &widget);
```

```
        Connector_Set_Right(&connector, &gadget);
        Connector_Post(&connector, context);
    }
    else if (instanceName->Key.value == 1002)
    {
        // Connector.Left="Widget.Key=1002",Right="Gadget.Key=1004"
        Widget_Set_Key(&widget, 1002);
        Gadget_Set_Key(&gadget, 1004);
        Connector_Set_Left(&connector, &widget);
        Connector_Set_Right(&connector, &gadget);
        Connector_Post(&connector, context);
    }
    else
    {
        r = MI_RESULT_NOT_FOUND;
    }

    Widget_Destruct(&widget);
    Gadget_Destruct(&gadget);
    Connector_Destruct(&connector);

    MI_Context_PostResult(context, r);
}
```

8.3.8 Implementing indication operations

Please note that indications should only be posted from a separate thread within the provider (a "background thread"). They should not be posted using the `omiserver` thread during a call into a provider. For example, during an `EnumerateInstances` call, the provider should directly post its instances and final result. It should not directly post indications within that function call. Instead, it should post indications on a separate background thread. Violation of this policy may trigger unexpected behavior in `omiserver`. The following examples illustrate appropriate techniques for posting alert and lifecycle indications.

8.3.8.1 Implementing an alert indication

As an example, consider the following MOF schema:

```
/* Indication class derived from the CIM standard indication class */
class XYZ_DiskFault : CIM_Indication
{
  string detailmessage;
};
```

The generated code snippet looks like this:

```
void MI_CALL XYZ_DiskFault_EnableIndications(
        XYZ_DiskFault_Self* self,
        MI_Context* indicationsContext,
        const MI_Char* nameSpace,
        const MI_Char* className )
```

```
{
  /* TODO: store indicationsContext for posting indication usage */
  /* NOTE: Call one of following functions if and ONLY if a termination
           error occurs, to finalize the indicationsContext,
           and terminate all active subscriptions to current class:

             MI_Context_PostResult
             MI_Context_PostError
             MI_Context_PostCimError      */
}

void MI_CALL XYZ_DiskFault_DisableIndications(
        XYZ_DiskFault_Self* self,
        MI_Context* indicationsContext,
        const MI_Char* nameSpace,
        const MI_Char* className )
{
  /* TODO: stop background thread that monitors subscriptions */
  MI_PostResult( indicationsContext, MI_RESULT_OK );
}

void MI_CALL XYZ_DiskFault_Subscribe(
        XYZ_DiskFault_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const MI_Filter* filter,
        const MI_Char* bookmark,
        MI_Uint64  subscriptionID,
        void** subscriptionSelf )
{
  /* NOTE: This function indicates that a new subscription occurred */
}

void MI_CALL XYZ_DiskFault_Unsubscribe(
        XYZ_DiskFault_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        MI_Uint64  subscriptionID,
        void* subscriptionSelf )
{
  /* NOTE: This function indicates that a subscription was cancelled */
  MI_PostResult( context, MI_RESULT_OK );
}
```

The implementation of this indication class is shown below (see the sample code in the `Indication/Alert` subdirectory under the samples directory for the complete implementation):

```
void MI_CALL XYZ_DiskFault_EnableIndications(
        XYZ_DiskFault_Self* self,
        MI_Context* indicationsContext,
        const MI_Char* nameSpace,
        const MI_Char* className )
{
  /* TODO: store indicationsContext for posting indication usage */
  /* NOTE: Call one of following functions if and ONLY if a termination
           error occurs, to finalize the indicationsContext,
           and terminate all active subscriptions to current class:

             MI_Context_PostResult
             MI_Context_PostError
             MI_Context_PostCimError      */

  int code;
  memset( self, 0, sizeof(XYZ_DiskFault_Self) );
  self->self.context = indicationsContext;
  self->self.postindication = TriggerIndication;
  code = Thread_Create(&self->self.thread, fireIndication, (void*)self);
  if ( code != 0 )
  {
    /* Failed to create thread */
    MI_Context_PostError( indicationsContext, MI_RESULT_FAILED,
                          MI_T("MI"), MI_T("Failed to create thread") );
  }
}

void MI_CALL XYZ_DiskFault_DisableIndications(
        XYZ_DiskFault_Self* self,
        MI_Context* indicationsContext,
        const MI_Char* nameSpace,
        const MI_Char* className )
{
  /* TODO: stop background thread that monitors subscriptions */
  MI_Uint32 retValue;
  self->self.disabling = MI_TRUE;
  Thread_Join( & self->self.thread, &retValue );
  MI_PostResult( indicationsContext, MI_RESULT_OK );
}
```

8.3.8.2 Implementing a lifecycle indication

This section describes how to implement lifecycle indication support in a normal class.

Consider a common process class as an example:

```
/* A class derived from CIM_Process */
class XYZ_Process : CIM_Process
{
  uint32 runningTime;

  [static]
  uint32 Create( [in] string imageName, [out] CIM_Process REF process );
  uint32 GetRunTime( [in] uint32 pid, [out] uint32 runningTime );
};
```

The generated code snippet looks like this:

```
void MI_CALL XYZ_Process_EnumerateInstances(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const MI_PropertySet* propertySet,
        MI_Boolean keysOnly,
        const MI_Filter* filter )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_GetInstance(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const XYZ_Process* instanceName,
        const MI_PropertySet* propertySet )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_CreateInstance(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const XYZ_Process* newInstance )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_ModifyInstance(
        XYZ_Process_Self* self,
```

```
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const XYZ_Process* modifiedInstance,
        const MI_PropertySet* propertySet )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_DeleteInstance(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const XYZ_Process* instanceName )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_Invoke_RequestStateChange(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const MI_Char* methodName,
        const XYZ_Process* instanceName,
        const XYZ_Process_RequestStateChange* in )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_Invoke_Create(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const MI_Char* methodName,
        const XYZ_Process* instanceName,
        const XYZ_Process_Create* in )
{
  MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}

void MI_CALL XYZ_Process_Invoke_GetRunTime(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
```

```
        const MI_Char* className,
        const MI_Char* methodName,
        const XYZ_Process* instanceName,
        const XYZ_Process_GetRunTime* in )
{
    MI_PostResult( context, MI_RESULT_NOT_SUPPORTED );
}
```

Implementation of lifecycle indications within this class takes a form like the following:

```
void MI_CALL XYZ_Process_Load(
        XYZ_Process_Self** self,
        MI_Module_Self* selfModule,
        MI_Context* context )
{
  MI_Result r;
  *self = &g_self;
  memset(&g_self, 0, sizeof(g_self));

  /* get lifecycle context and store the context for
   * posting indication usage */
  r = MI_Context_GetLifecycleIndicationContext(
      context, &(*self)->context);
  CHECKR_POST_RETURN_VOID(context, r);

  /* register a callback, which will be invoked if subscription
   * changed, i.e., client cancelled a subscription or create a new
   * subscription */
  r = MI_LifecycleIndicationContext_RegisterCallback(
      (*self)->context, _LifecycleIndicationCallback,
      (void*)(ptrdiff_t)(*self));
  CHECKR_POST_RETURN_VOID(context, r);

  /* notify server that what types of lifecycle indication
   * supported by current class */
  r = MI_LifecycleIndicationContext_SetSupportedTypes(
      (*self)->context, MI_LIFECYCLE_INDICATION_CREATE);
  CHECKR_POST_RETURN_VOID(context, r);

  /* intialize global data */
  r = _Initialize(context, *self);
  if (r != MI_RESULT_OK)
  {
    _Finalize(*self);
    *self = NULL;
  }
  else
  {
```

```
    int code;
    (*self)->self.context = (*self)->context;
    (*self)->self.postindication = TriggerIndication;

    /*
     * Please note that lifecycle indications should only be posted
     * from a background thread.
     *
     * This XYZ_Process example spawns a background thread that will
     * periodically fire CIM_InstCreation indications to simulate
     * process creation events.
     */
    code = Thread_Create(&(*self)->self.thread, fireIndication,
        (void*)(*self));
    if ( code != 0 )
    {
        /* Failed to create thread */
        r = MI_RESULT_FAILED;
        _Finalize(*self);
    }
  }
  MI_PostResult(context, r); *self = &g_self;
}

void MI_CALL XYZ_Process_Unload(
        XYZ_Process_Self* self,
        MI_Context* context)
{
  MI_Uint32 retValue;

  /* Shutdown the spawned thread */
  self->self.disabling = MI_TRUE;
  Thread_Join( & self->self.thread, &retValue );

  /* cleanup */
  _Finalize( self );
  MI_LifecycleIndicationContext_PostResult(self->context,MI_RESULT_OK);
  MI_PostResult( context, MI_RESULT_OK );       _Finalize(self);
}

void MI_CALL XYZ_Process_EnumerateInstances(
        XYZ_Process_Self* self,
        MI_Context* context,
        const MI_Char* nameSpace,
        const MI_Char* className,
        const MI_PropertySet* propertySet,
        MI_Boolean keysOnly,
```

```
        const MI_Filter* filter )
{
  MI_Uint32 i;
  MI_Result r;
  for( i = 0; i < PROCESS_COUNT; i++ )
  {
    r = XYZ_Process_Post( &self->processes[i], context );
    if( r != MI_RESULT_OK )
      break;
  }
  MI_PostResult( context, r );
}

/*
 * Callback function to trigger lifecycle indication (CIM_InstCreation)
 * for XYZ_Process.
 * This function simulates the process creation events by posting a
 * CIM_InstCreation indication.
 *
 * In reality, the provider may utilize the system API to monitor the
 * system-wide process creation event, and generate a corresponding
 * CIM_InstCreation indication via the lifecycle context; the same goes
 * for CIM_InstDeletion, CIM_InstModification, CIM_InstRead,
 * CIM_InstMethodCall, etc.
 */
MI_Uint32 MI_CALL TriggerIndication(
    _In_ void* callbackdata)
{
  XYZ_Process_Self* self = (XYZ_Process_Self*)callbackdata;
  MI_LifecycleIndicationContext* context = self->context;
  MI_Uint32 seqno = self->seqno++;
  MI_Uint32 index = seqno % PROCESS_COUNT;

  /*
   * When practical, the provider developer needs to get the process
   * creation event, for example by monitoring a system event or
   * responding to a system API, and then trigger a Cim_InstCreation
   * indication.
   *
   * In order to trigger a CIM_InstCreation indication, the XYZ_Process
   * instance needs to be created first.
   * After that, call MI_LifecycleIndicationContext_PostCreate function.
   *
   * The following sample code shows how to create a XYZ_Process
   * instance and trigger a CIM_InstCreation indication of XYZ_Process.
   */
  if( self->types & MI_LIFECYCLE_INDICATION_CREATE )
```

```
  {
    XYZ_Process process;
    MI_Result r;
    r = MI_LifecycleIndicationContext_ConstructInstance(
      context, &XYZ_Process_rtti, &process.__instance);
    if (r == MI_RESULT_OK)
    {
      r = _SetPropertyValue(index, &process);
      if ( r == MI_RESULT_OK )
        /* Post CIM_InstCreation indication from background thread */
        MI_LifecycleIndicationContext_PostCreate(context,
            &process.__instance);

      /* Destroy the instance */
      MI_Instance_Destruct(&process.__instance);
    }
  }
  return 1;
}

/*========================================================================
**
** Thread proc to post the indication periodically
**
**======================================================================*/
MI_Uint32 THREAD_API fireIndication(void* param)
{
  SelfStruct* self = (SelfStruct*)param;
  MI_Uint32 exitvalue = 1;

  if ( !self || !self->context || !self->postindication )
  {
    exitvalue = 0;
    goto EXIT;
  }

  /* initialize random seed: */
  SRAND();

  /* produce and post indication */
  while( MI_TRUE != self->disabling )
  {
    if ( 0 == self->postindication(self) )
      break;

    /* randomly sleep */
    Sleep_Milliseconds(rand() % 1000 + 500);
```

```
  }

EXIT:
  Thread_Exit(exitvalue);
  return exitvalue;
}
```

An alternative way to generate a lifecycle indication is to define an indication class that derives from the lifecycle indication class. For example, consider the `ABC_Process` class below. `ABC_ProcessCreation` is defined so as to generate a CIM_InstCreation indication for the `ABC_Process` class:

```
class ABC_Process : CIM_Process
{
  uint32 runningTime;

  [static]
  uint32 Create( [in] string imageName, [out] CIM_Process REF process );
};

class ABC_ProcessCreation : CIM_InstCreation
{
};
```

8.3.8.3 More sample code for indications

For more detailed examples of indication implementations, please see the Indication subdirectory under the samples directory.

8.4 Building the provider

To build the provider with the generated `GNUmakefile` just type `make`. This creates a shared library, containing the provider. On Linux, this file will be named `libxyzconnector.so`.

8.5 Registering the provider

To register the provider, use the `omireg` tool, which creates a registration file (`xyzconnector.reg`) under the registration directory and copies the provider to the `lib` directory. For example:

```
$ omireg libxyzconnector.so
Created /opt/omi/lib/libxyzconnector.so
Created /opt/omi/etc/omiregister/root-cimv2/xyzconnector.reg
```

By default the provider is hosted in the same process as the server. To host the provider in a separate process see the `--hosting` option.

Also by default the provider services the `root/cimv2` namespace. To change the namespace, see the `--namespace` option.

For more help with the `omireg` tool, use the `-h` option.

The contents of the `xyzconnector.reg` file are listed below.

```
# omireg /home/someuser/connector/libxyzconnector.so
HOSTING=@inproc@
LIBRARY=xyzconnector
CLASS=XYZ_Connector{XYZ_Widget,XYZ_Gadget}
CLASS=XYZ_Gadget
CLASS=XYZ_Widget
```

It is better to use `omireg` to re-generate these files rather than editing them directly. If you do edit them, you should only need to change the hosting model. The supported hosting models include:

- **@inproc@** – provider runs in same process as server.
- **@requestor@** – provider runs in separate process as the requester's user (the client's authenticated user).
- **USERNAME** – provider runs as this specified user.

8.6 Validating the provider

To validate the provider, use the `omicli` tool to send requests to the new providers. The following command enumerates instances of the new `Widget` provider.

```
# omicli ei root/cimv2 XYZ_Widget
```

Appendix A Frog Provider Sources

A.1 'schema.mof'

```
class XYZ_Frog
{
    [Key] String Name;
    Uint32 Weight;
    String Color;
};
```

A.2 'XYZ_Frog.h'

```
/* @migen@ */
/*
**==============================================================================
**
** WARNING: THIS FILE WAS AUTOMATICALLY GENERATED. PLEASE DO NOT EDIT.
**
**==============================================================================
*/
#ifndef _XYZ_Frog_h
#define _XYZ_Frog_h

#include <MI.h>

/*
**==============================================================
**
** XYZ_Frog [XYZ_Frog]
**
** Keys:
**    Name
**
**==============================================================
*/

typedef struct _XYZ_Frog
{
    MI_Instance __instance;
    /* XYZ_Frog properties */
    /*KEY*/ MI_ConstStringField Name;
    MI_ConstUint32Field Weight;
    MI_ConstStringField Color;
}
XYZ_Frog;
```

```
typedef struct _XYZ_Frog_Ref
{
    XYZ_Frog* value;
    MI_Boolean exists;
    MI_Uint8 flags;
}
XYZ_Frog_Ref;

typedef struct _XYZ_Frog_ConstRef
{
    MI_CONST XYZ_Frog* value;
    MI_Boolean exists;
    MI_Uint8 flags;
}
XYZ_Frog_ConstRef;

typedef struct _XYZ_Frog_Array
{
    struct _XYZ_Frog** data;
    MI_Uint32 size;
}
XYZ_Frog_Array;

typedef struct _XYZ_Frog_ConstArray
{
    struct _XYZ_Frog MI_CONST* MI_CONST* data;
    MI_Uint32 size;
}
XYZ_Frog_ConstArray;

typedef struct _XYZ_Frog_ArrayRef
{
    XYZ_Frog_Array value;
    MI_Boolean exists;
    MI_Uint8 flags;
}
XYZ_Frog_ArrayRef;

typedef struct _XYZ_Frog_ConstArrayRef
{
    XYZ_Frog_ConstArray value;
    MI_Boolean exists;
    MI_Uint8 flags;
}
XYZ_Frog_ConstArrayRef;

MI_EXTERN_C MI_CONST MI_ClassDecl XYZ_Frog_rtti;
```

```
MI_INLINE MI_Result MI_CALL XYZ_Frog_Construct(
    XYZ_Frog* self,
    MI_Context* context)
{
    return MI_ConstructInstance(context, &XYZ_Frog_rtti,
        (MI_Instance*)&self->__instance);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Clone(
    const XYZ_Frog* self,
    XYZ_Frog** newInstance)
{
    return MI_Instance_Clone(
        &self->__instance, (MI_Instance**)newInstance);
}

MI_INLINE MI_Boolean MI_CALL XYZ_Frog_IsA(
    const MI_Instance* self)
{
    MI_Boolean res = MI_FALSE;
    return MI_Instance_IsA(self, &XYZ_Frog_rtti, &res) ==
           MI_RESULT_OK && res;
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Destruct(XYZ_Frog* self)
{
    return MI_Instance_Destruct(&self->__instance);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Delete(XYZ_Frog* self)
{
    return MI_Instance_Delete(&self->__instance);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Post(
    const XYZ_Frog* self,
    MI_Context* context)
{
    return MI_PostInstance(context, &self->__instance);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Set_Name(
    XYZ_Frog* self,
    const MI_Char* str)
{
    return self->__instance.ft->SetElementAt(
```

```
        (MI_Instance*)&self->__instance,
        0,
        (MI_Value*)&str,
        MI_STRING,
        0);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_SetPtr_Name(
    XYZ_Frog* self,
    const MI_Char* str)
{
    return self->__instance.ft->SetElementAt(
        (MI_Instance*)&self->__instance,
        0,
        (MI_Value*)&str,
        MI_STRING,
        MI_FLAG_BORROW);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Clear_Name(
    XYZ_Frog* self)
{
    return self->__instance.ft->ClearElementAt(
        (MI_Instance*)&self->__instance,
        0);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Set_Weight(
    XYZ_Frog* self,
    MI_Uint32 x)
{
    ((MI_Uint32Field*)&self->Weight)->value = x;
    ((MI_Uint32Field*)&self->Weight)->exists = 1;
    return MI_RESULT_OK;
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Clear_Weight(
    XYZ_Frog* self)
{
    memset((void*)&self->Weight, 0, sizeof(self->Weight));
    return MI_RESULT_OK;
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Set_Color(
    XYZ_Frog* self,
    const MI_Char* str)
{
```

```
    return self->__instance.ft->SetElementAt(
        (MI_Instance*)&self->__instance,
        2,
        (MI_Value*)&str,
        MI_STRING,
        0);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_SetPtr_Color(
    XYZ_Frog* self,
    const MI_Char* str)
{
    return self->__instance.ft->SetElementAt(
        (MI_Instance*)&self->__instance,
        2,
        (MI_Value*)&str,
        MI_STRING,
        MI_FLAG_BORROW);
}

MI_INLINE MI_Result MI_CALL XYZ_Frog_Clear_Color(
    XYZ_Frog* self)
{
    return self->__instance.ft->ClearElementAt(
        (MI_Instance*)&self->__instance,
        2);
}

/*
**==============================================================================
**
** XYZ_Frog provider function prototypes
**
**==============================================================================
*/

/* The developer may optionally define this structure */
typedef struct _XYZ_Frog_Self XYZ_Frog_Self;

MI_EXTERN_C void MI_CALL XYZ_Frog_Load(
    XYZ_Frog_Self** self,
    MI_Module_Self* selfModule,
    MI_Context* context);

MI_EXTERN_C void MI_CALL XYZ_Frog_Unload(
    XYZ_Frog_Self* self,
    MI_Context* context);
```

```
MI_EXTERN_C void MI_CALL XYZ_Frog_EnumerateInstances(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter);

MI_EXTERN_C void MI_CALL XYZ_Frog_GetInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* instanceName,
    const MI_PropertySet* propertySet);

MI_EXTERN_C void MI_CALL XYZ_Frog_CreateInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* newInstance);

MI_EXTERN_C void MI_CALL XYZ_Frog_ModifyInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* modifiedInstance,
    const MI_PropertySet* propertySet);

MI_EXTERN_C void MI_CALL XYZ_Frog_DeleteInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* instanceName);


#endif /* _XYZ_Frog_h */
```

A.3 'XYZ_Frog.c'

```
/* @migen@ */
#include <MI.h>
```

```
#include "XYZ_Frog.h"

void MI_CALL XYZ_Frog_Load(
    XYZ_Frog_Self** self,
    MI_Module_Self* selfModule,
    MI_Context* context)
{
    *self = NULL;
    MI_Context_PostResult(context, MI_RESULT_OK);
}

void MI_CALL XYZ_Frog_Unload(
    XYZ_Frog_Self* self,
    MI_Context* context)
{
    MI_Context_PostResult(context, MI_RESULT_OK);
}

void MI_CALL XYZ_Frog_EnumerateInstances(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const MI_PropertySet* propertySet,
    MI_Boolean keysOnly,
    const MI_Filter* filter)
{
    XYZ_Frog frog1;
    XYZ_Frog_Construct(&frog1, context);
    XYZ_Frog_Set_Name(&frog1, MI_T("Fred"));
    XYZ_Frog_Set_Weight(&frog1, 55);
    XYZ_Frog_Set_Color(&frog1, MI_T("Blue"));
    XYZ_Frog_Post(&frog1, context);
    XYZ_Frog_Destruct(&frog1);

    XYZ_Frog frog2;
    XYZ_Frog_Construct(&frog2, context);
    XYZ_Frog_Set_Name(&frog2, MI_T("Sam"));
    XYZ_Frog_Set_Weight(&frog2, 55);
    XYZ_Frog_Set_Color(&frog2, MI_T("Blue"));
    XYZ_Frog_Post(&frog2, context);
    XYZ_Frog_Destruct(&frog2);

    MI_Context_PostResult(context, MI_RESULT_OK);
}

void MI_CALL XYZ_Frog_GetInstance(
```

```
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* instanceName,
    const MI_PropertySet* propertySet)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}

void MI_CALL XYZ_Frog_CreateInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* newInstance)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}

void MI_CALL XYZ_Frog_ModifyInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* modifiedInstance,
    const MI_PropertySet* propertySet)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}

void MI_CALL XYZ_Frog_DeleteInstance(
    XYZ_Frog_Self* self,
    MI_Context* context,
    const MI_Char* nameSpace,
    const MI_Char* className,
    const XYZ_Frog* instanceName)
{
    MI_Context_PostResult(context, MI_RESULT_NOT_SUPPORTED);
}
```

A.4 ‘module.c’

```
/* @migen@ */
#include <MI.h>

MI_EXTERN_C MI_SchemaDecl schemaDecl;
```

```
void MI_CALL Load(MI_Module_Self** self, struct _MI_Context* context)
{
    *self = NULL;
    MI_Context_PostResult(context, MI_RESULT_OK);
}

void MI_CALL Unload(MI_Module_Self* self, struct _MI_Context* context)
{
    MI_Context_PostResult(context, MI_RESULT_OK);
}

MI_EXTERN_C MI_EXPORT MI_Module* MI_MAIN_CALL MI_Main(MI_Server* server)
/*  WARNING: THIS FUNCTION AUTOMATICALLY GENERATED.
    PLEASE DO NOT EDIT. */
{
    extern MI_Server* __mi_server;
    static MI_Module module;
    __mi_server = server;
    module.flags |= MI_MODULE_FLAG_STANDARD_QUALIFIERS;
    module.charSize = sizeof(MI_Char);
    module.version = MI_VERSION;
    module.generatorVersion = MI_MAKE_VERSION(1,0,8);
    module.schemaDecl = &schemaDecl;
    module.Load = Load;
    module.Unload = Unload;
    return &module;
}
```

A.5 ‘schema.c’

```
/* @migen@ */
/*
**==============================================================================
**
** WARNING: THIS FILE WAS AUTOMATICALLY GENERATED. PLEASE DO NOT EDIT.
**
**==============================================================================
*/
#include <ctype.h>
#include <MI.h>
#include "XYZ_Frog.h"

/*
**==============================================================================
**
** Schema Declaration
```

```
**
**==============================================================================
*/

extern MI_SchemaDecl schemaDecl;

/*
**==============================================================================
**
** Qualifier declarations
**
**==============================================================================
*/

/*
**==============================================================================
**
** XYZ_Frog
**
**==============================================================================
*/

/* property XYZ_Frog.Name */
static MI_CONST MI_PropertyDecl XYZ_Frog_Name_prop =
{
    MI_FLAG_PROPERTY|MI_FLAG_KEY, /* flags */
    0x006E6504, /* code */
    MI_T("Name"), /* name */
    NULL, /* qualifiers */
    0, /* numQualifiers */
    MI_STRING, /* type */
    NULL, /* className */
    0, /* subscript */
    offsetof(XYZ_Frog, Name), /* offset */
    MI_T("XYZ_Frog"), /* origin */
    MI_T("XYZ_Frog"), /* propagator */
    NULL,
};

/* property XYZ_Frog.Weight */
static MI_CONST MI_PropertyDecl XYZ_Frog_Weight_prop =
{
    MI_FLAG_PROPERTY, /* flags */
    0x00777406, /* code */
    MI_T("Weight"), /* name */
    NULL, /* qualifiers */
    0, /* numQualifiers */
```

```
    MI_UINT32, /* type */
    NULL, /* className */
    0, /* subscript */
    offsetof(XYZ_Frog, Weight), /* offset */
    MI_T("XYZ_Frog"), /* origin */
    MI_T("XYZ_Frog"), /* propagator */
    NULL,
};

/* property XYZ_Frog.Color */
static MI_CONST MI_PropertyDecl XYZ_Frog_Color_prop =
{
    MI_FLAG_PROPERTY, /* flags */
    0x00637205, /* code */
    MI_T("Color"), /* name */
    NULL, /* qualifiers */
    0, /* numQualifiers */
    MI_STRING, /* type */
    NULL, /* className */
    0, /* subscript */
    offsetof(XYZ_Frog, Color), /* offset */
    MI_T("XYZ_Frog"), /* origin */
    MI_T("XYZ_Frog"), /* propagator */
    NULL,
};

static MI_PropertyDecl MI_CONST* MI_CONST XYZ_Frog_props[] =
{
    &XYZ_Frog_Name_prop,
    &XYZ_Frog_Weight_prop,
    &XYZ_Frog_Color_prop,
};

static MI_CONST MI_ProviderFT XYZ_Frog_funcs =
{
  (MI_ProviderFT_Load)XYZ_Frog_Load,
  (MI_ProviderFT_Unload)XYZ_Frog_Unload,
  (MI_ProviderFT_GetInstance)XYZ_Frog_GetInstance,
  (MI_ProviderFT_EnumerateInstances)XYZ_Frog_EnumerateInstances,
  (MI_ProviderFT_CreateInstance)XYZ_Frog_CreateInstance,
  (MI_ProviderFT_ModifyInstance)XYZ_Frog_ModifyInstance,
  (MI_ProviderFT_DeleteInstance)XYZ_Frog_DeleteInstance,
  (MI_ProviderFT_AssociatorInstances)NULL,
  (MI_ProviderFT_ReferenceInstances)NULL,
  (MI_ProviderFT_EnableIndications)NULL,
  (MI_ProviderFT_DisableIndications)NULL,
  (MI_ProviderFT_Subscribe)NULL,
```

```
  (MI_ProviderFT_Unsubscribe)NULL,
  (MI_ProviderFT_Invoke)NULL,
};

/* class XYZ_Frog */
MI_CONST MI_ClassDecl XYZ_Frog_rtti =
{
    MI_FLAG_CLASS, /* flags */
    0x00786708, /* code */
    MI_T("XYZ_Frog"), /* name */
    NULL, /* qualifiers */
    0, /* numQualifiers */
    XYZ_Frog_props, /* properties */
    MI_COUNT(XYZ_Frog_props), /* numProperties */
    sizeof(XYZ_Frog), /* size */
    NULL, /* superClass */
    NULL, /* superClassDecl */
    NULL, /* methods */
    0, /* numMethods */
    &schemaDecl, /* schema */
    &XYZ_Frog_funcs, /* functions */
    NULL, /* owningClass */
};

/*
**==============================================================================
**
** __mi_server
**
**==============================================================================
*/

MI_Server* __mi_server;
/*
**==============================================================================
**
** Schema
**
**==============================================================================
*/

static MI_ClassDecl MI_CONST* MI_CONST classes[] =
{
    &XYZ_Frog_rtti,
};

MI_SchemaDecl schemaDecl =
```

```
{
    NULL, /* qualifierDecls */
    0, /* numQualifierDecls */
    classes, /* classDecls */
    MI_COUNT(classes), /* classDecls */
};

/*
**==============================================================================
**
** MI_Server Methods
**
**==============================================================================
*/

MI_Result MI_CALL MI_Server_GetVersion(
    MI_Uint32* version){
    return __mi_server->serverFT->GetVersion(version);
}

MI_Result MI_CALL MI_Server_GetSystemName(
    const MI_Char** systemName)
{
    return __mi_server->serverFT->GetSystemName(systemName);
}
```

A.6 'GNUmakefile'

```
include /opt/omi/share/omi.mak

DEFINES = MI_API_VERSION=2
PROVIDER = frog
SOURCES = $(wildcard *.c)
CLASSES = XYZ_Frog

$(LIBRARY): $(OBJECTS)
        $(CC) $(CSHLIBFLAGS) $(OBJECTS) -o $(LIBRARY)

%.o: %.c
        $(CC) -c $(CFLAGS) $(INCLUDES) $< -o $@

reg:
        $(BINDIR)/omireg $(CURDIR)/$(LIBRARY)

gen:
        $(BINDIR)/omigen -m frog schema.mof XYZ_Frog
```

```
clean:
        rm -f $(LIBRARY) $(OBJECTS) $(PROVIDER).reg
```

Appendix B Asynchronous Enumerate Instances Client Example

B.1 'AsyncEnum.cpp'

```
#include <cstdio>
#include <omiclient/client.h>

#define T MI_T

using namespace mi;

class MyHandler : public Handler
{
public:

    MyHandler() : done(false)
    {
    }

    virtual void HandleConnect()
    {
        printf("==== MyHandler::HandleConnect()\n");
    }

    virtual void HandleNoOp(Uint64 msgID)
    {
        printf("==== MyHandler::HandleNoOp()\n");
    }

    virtual void HandleConnectFailed()
    {
        printf("==== MyHandler::HandleConnectFailed()\n");

        // Handler error!
        done = true;
    }

    virtual void HandleDisconnect()
    {
        printf("==== MyHandler::HandleDisconnect()\n");
        done = true;
    }

    virtual void HandleInstance(Uint64 msgID, const DInstance& instance)
    {
        printf("==== MyHandler::HandleInstance()\n");
```

```
        instance.Print();
    }

    virtual void HandleResult(Uint64 msgID, MI_Result result)
    {
        printf("==== MyHandler::HandleResult()\n");
        done = true;
    }

    bool done;
};

int main(int argc, const char* argv[])
{
  int r = 0;

  // Create handler:
  MyHandler* handler = new MyHandler;

  // Construct client:
  Client client(handler);

  String locator;
  String username;
  String password;

  if (!client.ConnectAsync(locator, username, password))
  {
    // Handle error!
  }

  const String nameSpace = "root/omi";
  const String className = "OMI_Identify";
  const bool deep = true;
  Uint64 msgID;

  if (!client.EnumerateInstancesAsync( nameSpace, className,
                                       deep, msgID))
  {
    // Handle error!
  }

  // Wait here for 5 seconds for operation to finish.
  while (!handler->done)
  {
    client.Run(1000);
```

```
    }

    return r;
}
```

Appendix C Cross compiling OMI

C.1 Synopsis

This appendix explains how to build OMI with a cross compiler. Two such targets are supported today:

```
MONTAVISTA_IX86_GNU
NETBSD_IX86_GNU
```

Additional platforms can be supported by extending the `buildtool` script (following MONTAVISTA_IX86_GNU target as an example).

C.2 Terminology

The "host" platform is where the compiler is run to build OMI. The "target" platform is where the output binaries files will run. These can be the same, but in the case of cross-compiling they are different. This appendix uses the term "target" to refer to the platform where the binaries will be run.

C.3 Configuring

By default, the `configure` script guesses the platform based on the host environment. But with cross-compiling, the platform is given by the –target option as shown below:

```
./configure --target=MONTAVISTA_IX86_GNU
```

Additional options are required to specify to location of the cross-compiler components. These include:

```
--with-cc=PATH          Use C compiler given by PATH.
--with-cxx=PATH         Use C++ compiler given by PATH.
--with-ar=PATH          Use archive command (ar) given by PATH.
--openssl=PATH          Full path to the "openssl" command.
--opensslcflags=FLAGS   Extra C flags needed for OpenSSL.
                        (e.g. "-I/usr/local/include").
--openssllibs=FLAGS     Extra library options needed for OpenSSL.
                        (e.g. "-L/usr/local/lib -lssl -lcrypto").
--openssllibdir=PATH    The path of the directory containing the desired
                        OpenSSL libraries (ssl and crypto).
```

So to run configure, one might have something like this:

```
./configure
    --target=NETBSD_IX86_GNU
    --with-cc=/opt/toolchain/bin/586-gcc
    --with-cxx=/opt/toolchain/bin/586-g++
    --with-ar=/opt/toolchain/bin/586-ar
    --opensslcflags="-I /opt/toolchain/include"
    --openssllibdir=/opt/toolchain/lib64
    --openssllibs="-L/opt/toolchain/lib64 -lssl -lcrypto"
```

Of course this is only an example. The exact location of these components will vary.

C.4 Installing

When cross-compiling, it is obviously no longer possible to simply type "make install" to install the components (since the components must be copied to another platform). Instead, components may be installed into an interim directory and then copied from there onto the target platform. The DESTDIR variable can be used for this purpose. For example, suppose that we configured like this:

```
./configure
    --target=NETBSD_IX86_GNU
    --prefix=/opt/omi
    --with-cc=/opt/toolchain/bin/586-gcc
    --with-cxx=/opt/toolchain/bin/586-g++
    --with-ar=/opt/toolchain/bin/586-ar
    --opensslcflags="-I /opt/toolchain/include"
    --openssllibdir=/opt/toolchain/lib64
    --openssllibs="-L/opt/toolchain/lib64 -lssl -lcrypto"
```

Next, the installable components can all be copied to an interim directory called "/tmp/install" like this:

```
make DESTDIR=/tmp/install install
```

Finally, the components can be copied from the /tmp/install directory to the target machine.

Appendix D NITS Integrated Test System

D.1 Introduction

This document defines a C unit test framework and instructions for its use. This framework is a replacement for existing unit test infrastructure that makes debugging, maintenance, and code coverage significantly easier. The unit test framework encompasses several related components:

- **libnits.so**: Contains the framework implementation. This binary contains shared memory spaces appropriate for test builds only. Product binaries must not link to `libnits.so` on official builds.
- **libnitsinj.so**: Injects the test framework and any other mocked function tables into relevant processes during unit test runs. No one links to this binary directly.
- **libpal.a**: Contains no-op implementations of the NITS API for product binaries using HOOK_BUILD or DEBUG_BUILD linkage.
- **nits**: Contains a command-line interface for running the framework on a set of tests. Spawns child processes if necessary, runs test cases, and reports results.
- **Unit test shared libraries**: Contains product unit test suites. All test cases are declared within the .cpp files of these binaries. Links to libnits.so using TEST_BUILD linkage.
- **Product shared libraries and executables**: Uses macros and either runtime instrumentation or conditional blocks of code to run under unit test passes. Should use HOOK_BUILD or DEBUG_BUILD linkage.

D.2 Linkage

There are five different possible ways to link to the NITS package, each with its own consequences:

- **<default>**: All test macros are no-ops and NitsCallSite is minimized. It is not possible to hook anything at runtime. This option is used in projects/binaries that do not understand NITS.
- **HOOK_BUILD**: Test macros call NitsFT stubs in libpal.a. NitsCallSite is functional but does not show file/line information. Use the -target option to hook selected binaries at runtime. This option is for production shared libraries and executables.
- **DEBUG_BUILD**: Same as HOOK_BUILD, but includes `__FILE__`, `__LINE__`, and `__FUNCTION__` information. This option is intended for production shared libraries and executables running under a debug/checked build.
- **TEST_BUILD**: Test macros call the NITS implementation by linking with `libnits.so`. This is for test shared libraries.
-

D.3 Project Setup

This section describes the basic source tree configuration steps required to access the NITS API. Once these steps are complete, product and test binaries are ready to start using the

framework. The project will need access to the following files from the framework: `nits.h`, `libnits.so`, `libpal.a`, and `nits`.

The recommended, supported project configuration is as follows:

D.3.1 Product Binaries

- Ideally the product code should take the form of a shared library, and executable modules should only be thin wrappers over the shared library.
- Define HOOK_BUILD in product binaries as part of the GNUmakefile (for more debug information use DEBUG_BUILD, but it also bloats the binary).
- Include `<nits/base/nits.h>` in product and test GNUmakefile.
- Link to `libpal.a` from product binaries (`$(PALLIBS)` expands to libpal.a).
-

D.3.2 Unit Test Binaries

- Define TEST_BUILD in unit test binaries as part of the GNUmakefile.
- Link to `$(UNITTESTLIBS)` from unit test binaries. `$(UNITTESTLIBS)` expands to everything needed to build the test library.
- Include `libnits.so`, `libnitsinj.so`, and `nits` when distributing unit test binaries to test environments.

D.3.3 Pitfalls/Notes

- There are drawbacks to using `HOOK_BUILD` against executable modules (*i.e.* the trap table technique for calling internal product APIs does not work).

D.3.4 Sample Project

For canonical up-to-date working examples, see the sample product and test code in the following directories:

- There are examples of various linkage types under: `nits/linkageSample`.
- There are many test examples under `nits/sample`.
- There is product code under `nits/sampleproduct`.

D.4 Deployment

The nits binaries are built only in the `unittest` environment, *i.e.* when the `--dev` option is passed to the `./configure` script during configuration and build. the `nits` executable is created inside the `<output>/bin/` directory (where `"<output>"` is the directory where output from the OMI build is saved. The `libnits.so`, `libpal.a`, and `libnitsinj.so` are all present in the `<output>/lib` directory and can be used directly from there.

In order to run tests in all directories or in a single test directory, you can run:

```
make tests
```

This in turn invokes `nits` with appropriate command-line arguments and runs the single test or all test binaries.

The file `<output>/tmp/nitsargs.txt` contains the command-line arguments passed to nits during the test run, if you use the `-file:` option passed to nits to include it.

D.5 How to Call Product APIs

NITS provides facilities to allow private APIs to be callable at will without individually exporting all of them. This makes it convenient to link test shared libraries directly against product shared libraries, which in turn allows simple and reliable code coverage measurements. The alternative is linking the test code to product static libraries, which leads to product code existing in many binaries at once. While this is not technically broken, it is impossible to retrieve accurate coverage data in complex cases, and this has some tendency to encourage duplication of product code in multiple product binaries.

D.5.1 Public APIs

Public interfaces must be exported somehow. This may be done individually for each API, or function tables may be used (as in `mi.h`) to export an entire collection of APIs from one binary using a single export macro like `MI_EXPORT`. The latter technique leads to smaller binary size but requires more care to avoid introducing breaking changes to the contents of the table.

D.5.2 Private APIs

Non-public interfaces do not need to be exported in the same way. The recommended approach is to place all of the externally callable private APIs from a given product binary into a single `NitsTrapTable`, and then import that table into all the binaries that call any of the APIs. Macros can be used to hide this behavior from the calling binaries. A simple example is available in `nits/sample/test.cpp`.

D.6 How to Mock Product APIs

NITS supports mocking private APIs that are already exposed through a function table using the `NitsSetTrap` API. The only additional change needed in the product is that all mockable calls in the product must invoke through the trap table, rather than directly calling the function. When NitsSetTrap is called, the function pointer in the table is replaced and all calls through the pointer are redirected to the mock. For a simplified proof of concept, see `AuditTest` in `nits/sample/test.spp`.

D.7 Command Line Interface

The usage pattern for NITS is the following: `nits [option|test]*`. The rest of this section describes the available options and test types, along with the possible results for each test.

D.7.1 Options

- **-bpassert**: Issues a breakpoint any time an assertion fails when it should not. This is useful for debugging when it takes many steps to walk through the code to the point where the assertion fails.
- **-bpfault:number**: Issues a breakpoint on a specific fault injection iteration number. This option may be used multiple times.
- **-file:name**: Appends the contents of name to the command line options.
- **-filter:binary[,binary...]**: Prevents the injector from targeting processes containing none of the binaries in the filter list.

- **-install**: Immediately installs the injector (`libnitsinj.so`) to run in all relevant processes created after this point (including upon future boots).
- **-mode:Skip**: Explores the test tree and skips all body fixtures.
- **-mode:Enable**: (default) Runs test cases without fault injection.
- **-mode:IterativeFault**: Runs test cases with Nth site single faults. This option is exhaustive for deterministic tests, but very slow.
- **-mode:StackFault**: Optimized single fault injection; ignores duplicates based on call stack frames.
- **-pause**: Waits for user input after running. Useful for holding settings like `target` and `trace` when doing manual testing.
- **-reset**: Ignores a test run in progress and forcibly resets the shared memory state. Use this to recover from shared memory corruption.
- **-target:binary[,binary...]**: Enables the injector on specific binaries in any process.
- **-trace:FailedTests**: Shows failed tests only. Least verbose result output.
- **-trace:WarnedTests**: Shows failed and warned tests only.
- **-trace:AllTests**: Shows all test cases.
- **-trace:Asserts**: (default) Shows failed asserts individually.
- **-trace:Warnings**: Shows failed asserts and warnings individually.
- **-trace:Iterations**: Shows fault injection iteration information and asserts/warnings.
- **-trace:Verbose**: Shows all of the above plus NitsTrace calls.
- **-uninstall**: Removes `libnitsinj.so` injection from all processes. Use this before updating NITS binaries.
- **-match:testFixtureNameSubstring**: Runs tests only if they contain fixture names matching the pattern (semicolon or comma-separated).
- **+name:value**: Sets test parameter "name" to "value". Test module can retrieve the value of the parameter by using the API NitsTestGetParam by passing the parameter name.
- **-fault**: This option is used for fault injection.

D.7.2 Tests

Tests may be run using any of the following syntax forms:

- **libmodule.so**: Specifying a file name runs all tests discovered during the loading of that library. NITS now accepts relative and absolute paths.
- **libmodule.so:test**: Specify the module name and a test name, separated by a bang, to run a particular test. This runs all test variations.
- **libmodule.so:+test**: Runs all tests in the module starting with the specified test.
- **libmodule.so:test/variation**: Specify the module name, the test name, and a set of variation choices to run one specific test variation. Each valid variation name is constructed from the set of choices made according to the hierarchy declared in the test code.
- **libmodule.so:*testnameSubstring**: Runs all tests whose names contain the pattern.

The simplest way to discover a list of valid test names and variations for a particular module is to run the entire module with the `-mode:Skip` option.

Note: To run the NITS sample tests, use `nits` as follows:

```
./output/bin/nits -install -reset -trace \
   -target:libnitssample.so,libnitssampleproduct.so \
   ./output/lib/libnitssample.so
```

D.7.3 Results

Each test variation reports a result to the test framework. These results are printed and then summarized. The possible test results are as follows:

- **Faulted**: The test passed, and was then run successfully under automatic fault simulation. There were no unexpected successes or failures during the fault simulation loop.
- **Passed**: The test passed, and automatic fault simulation was not run.
- **Skipped**: The test was disabled by the command line or the test setup code, and the test body was not run. However, no errors were encountered during setup or cleanup.
- **Omitted**: The test declared itself to be irrelevant. These can be safely ignored.
- **Killed**: When using the (old) isolate option, a test case was terminated after running for too long without reporting a result.
- **Failed**: Assertions failed during the test body, or there were unexpected successes or failures during automatic fault simulation.
- **Error**: Assertions failed during test setup or cleanup, or no assertions were attempted during the test body.

D.8 Implementation

This section describes how to create basic unit tests under the NITS framework. This section shows only the most basic examples, using as few features as possible. See the following section for additional examples for advanced features. To run a test binary containing a simple, "hello world" unit test, see the following example:

```
#include <nits/base/nits.h>

NitsTest(Test1)
  NitsTrace(PAL_T("Test Body!"));
  NitsAssert(1 == 1, PAL_T("assert test!"));
  NitsCompare(1, 1, PAL_T("compare test!"));
  NitsCompareString(PAL_T("A"), PAL_T("A"), PAL_T("string test!"));
  NitsCompareSubstring(PAL_T("ABC"),PAL_T("B"),
                       PAL_T("substring test!"));
NitsEndTest
```

The code above contains a simple test body with some basic assertions and traces. Note that the assertion descriptions and traces are using PAL_T(""), which is a single-char or wide-char string depending on the configuration of OMI. Inserting this code into `libhello.so` and compiling it produces a binary that can be run using the following command:

```
./output/bin/nits ./output/lib/libhello.so

        [Passed]  ./output/lib/libhello.so:Test1


Summary:
        Faulted:        0
        Passed:         1
        Skipped:        0
        Killed:         0
        Failed:         0
        Error:          0

        Successes:      1
        Failures:       0
        Total:          1
```

This listing shows that the test named `Test1` in `libhello.so` was run successfully. The test passed because the body ran one or more test assertions, all of which succeeded.

D.9 A Selection of More Advanced Features

This section contains a partial list of available NITS features, roughly in order of increasing complexity. Each feature can be used independently of the others unless otherwise specified.

D.9.1 Tracing

The API `NitsTrace` is available for tracing messages using the current source location as the call site. These messages will appear in `stderr` only if tracing is enabled.

D.9.2 Assertions

The `NitsAssert`, `NitsCompare`, `NitsCompareString`, and `NitsCompareSubstring` APIs validate results obtained during the test. These functions return true if successful, but otherwise they are printed to `stderr` if assertions are enabled.

D.9.3 Call Sites

NITS defines a `NitsCallSite` class which is available on all linkages, though on some linkages it becomes trivialized. This class contains a source file, line, function name, and call site ID to be used for fault injection and source location identification. The APIs above use the current source line (returned by the `NitsHere( )` macro) as the `NitsCallSite`. However, this is not appropriate in common helper functions where it is actually the caller"s location that is relevant. The following example shows how to carry a `CallSite` through to a helper function:

```
void TestHelper( NitsCallSite cs )
{
  //Prints location of NitsHere( ) below.
  NitsTraceEx( L"Helper function!", cs, NitsAutomatic );
}
```

```
NitsTest(Test1)
    TestHelper(NitsHere());
NitsEndTest
```

The macros `NitsTraceEx`, `NitsAssertEx`, `NitsCompareEx`, `NitsCompareStringEx`, and `NitsCompareSubstringEx` are available for this purpose, each taking the same argument list as their counterparts, plus a call site argument. `NitsCallSite` objects may be created easily using either the `NitsHere( )` or `NitsNamedCallSite` macros. For manual fault simulation, use `NitsNamedCallSite(id)`, where the site ID is defined by the application. Otherwise, `NitsHere( )`, which is anonymous, generally suffices.

D.9.4 Fixtures

NITS tests are made from one or more **Fixtures**. Fixtures are reusable units of test code. They support composability; *i.e.* they can be composed out of other fixtures. They also allow you to share data between each other by letting you associate a type(a C `struct`) with them and providing an initialization value for that type. There are various types of fixtures:

- **Setup**
- **Split**
- **Test**
- **Cleanup**
- **ModuleSetup**

These types of fixtures are described in the following sections, along with an illustration of how you can share data between fixtures using an auto-created abstraction called a `NitsContext( )`.

D.9.5 Setup Fixtures

D.9.5.1 Basic Setup Fixtures

The most basic setup fixture involves bracketing some test code between two NITS macros, namely `NitsSetup(<fixtureName>)` and `NitsEndSetup`. Here is an example of such a fixture:

```
NitsSetup(SimpleSetup)
        NitsTrace(PAL_T("SimpleSetup being run"));
NitsEndSetup
```

- `SimpleSetup` is the name of the fixture.
- `NitsSetup` and `NitsEndSetup` form the start and end of the Setup fixture.
- The part in between represents the body of the Setup fixture that is run during the test run.

Note: `NitsEndSetup` and similar end macros are part of every fixture, and they help NITS provide a function continuation style of usage, in which all the fixtures from which your test is composed are on the same stack. This lets you use local variables throughout the entire lifetime of the test.

D.9.5.2 Associating a Data Type with a Setup Fixture

To associate a data type with a Setup fixture, you need to define a C `struct` as the data type, and write test code between the `NitsSetup0( <fixtureName>, <name of the data type> )` and `NitsEndSetup` macros. Below is an example that shows how to associate a type with a Setup fixture:

```
struct MyStruct
{
    int x;
};
NitsSetup0( Fixture0, MyStruct )
    NitsContext( )->_MyStruct->x = 0;
NitsEndSetup
```

- `Fixture0` is the name of the fixture.
- `NitsSetup0` and `NitsEndSetup` start and end the Setup fixture. The numeral `0` indicates that it is composed of zero other fixtures. As explained below, `NitsSetup<N>` is used to define Setup fixtures composed of `N` other fixtures.
- `MyStruct` defines a C `struct` that is then associated with `Fixture0` as its data type.
- Once you associate a data type with a Setup fixture, NITS auto-creates an object of the type specified by `MyStruct` and makes it available within the `NitsContext( )` abstraction.
- The `NitsContext( )` will have an auto-created variable with a name composed of an underscore followed by the data-type name (*e.g.* in this example `_MyStruct`). This variable is a pointer to the auto-created data object, and can be used for data sharing between fixtures.
- As seen in the example below, `(NitsContext()->_MyStruct->x)` lets you access the `int x` from the auto-created object of type `MyStruct` that is pointed to by the variable `_MyStruct` within `NitsContext( )`.

D.9.5.3 Setup Fixture Composition

The syntax for composing a Setup fixture with 1 other child fixture wold begin with the start macro:

```
NitsSetup1( <fixtureName>,
            <name of fixture data Type>,
            <name of child fixture 1>,
            <initialization value for child fixture 1> )
```

The test code would follow, and be ended by the `NitsEndSetup` macro.

Below is an example of how to compose a fixture from one other child fixture while also associating a data type with each fixture so that you can pass data back and forth in either direction:

```
struct FooStruct1
{
  int i1;
};
```

```
struct FooStruct2
{
  int i2;
};

NitsSetup0( FooSetup1, FooStruct1 )
  NitsAssert(NitsContext()->_FooStruct1->i1 == 10,
             PAL_T("wrong value"));
  NitsContext()->_FooStruct1->i1 = 15;
  NitsTrace(PAL_T("FooSetup1 being run"));
NitsEndSetup

struct FooStruct1 sFooStruct1 = {10};

NitsSetup1( FooSetup2, FooStruct2, FooSetup1, sFooStruct1 )
  NitsAssert(NitsContext()->_FooSetup1->_FooStruct1->i1 == 15,
             PAL_T("wrong value"));
  NitsContext()->_FooSetup1->_FooStruct1->i1 = 35;
  NitsContext()->_FooStruct2->i2 = 25;
  NitsTrace(PAL_T("FooSetup2 being run"));
NitsEndSetup
```

- Here `FooSetup1` and `FooSetup2` are the names of the two Setup fixtures.
- `FooSetup1` uses the `NitsSetup0/NitsEndSetup` syntax, since it is not composed of any other fixtures.
- `FooSetup2` is composed of one child fixture, `FooSetup1`, and hence uses the `NitsSetup1/NitsEndSetup` syntax.
- The C `struct FooStruct1` is the data type associated with `FooSetup1`.
- The C `struct FooStruct2` is the data type associated with `FooSetup2`.
- When composing from one child fixture in `NitsSetup1`, the macro parameters are `FooSetup2, FooStruct2, FooSetup1, sFooStruct1`, which represent:
  - The name of the fixture.
  - The data type associated with the fixture.
  - The name of the child fixture.
  - The initialization value for the fixture.
- The initialization value `sFooStruct1` given to fixture `FooSetup1` is automatically populated into the data object pointed to by `NitsContext( )->_FooStruct` when the fixture `FooSetup1` runs. Note that the initialization object is used as copy-by-value and it is copied into the data object pointed to by `NitsContext( )` of the fixture that it is initializing. That is why `FooSetup1` in the above example can assert that the value of `NitsContext()->_FooStruct1->i1` will be 10 when it runs.
- The initialization value for a fixture has to be a compile-time-defined global variable of the data type associated with the fixture. Alternately, to pass an all-zero initialization value, you can use the NITS-auto-created global variable named `<name of fixture>Defaults` (*e.g.* in this example, if the `FooSetup2` fixture wanted to specify

an all-zero initialization value for `FooSetup1` then it could have used the auto-created object named `FooSetup1Defaults`.

- When a fixture is composed of other child fixtures, the auto-created `NitsContext( )` abstraction inside the fixture contains appropriate pointers to access the data from all the child fixtures it is composed of. The name of the variable inside `NitsContext( )` that points to the context of the child fixture takes the form, `_<child fixture name>`.
- In this example, `NitsContext( )` inside `FooSetup2` has the following variables:
  - `_FooStruct2`, which is a pointer to the data object of type `FooStruct2` auto-created for itself, since the data type associated with `FooSetup2` is `FooStruct2`.
  - `_FooSetup1`, which is a pointer to the `NitsContext( )` inside the child fixture `FooSetup1`.
  - `_FooSetup1` then contains the pointer `_FooStruct1`, which is a pointer to the data object associated with the child fixture `FooSetup1`, since the data type associated with `_FooSetup1` is `FooStruct1`.
- Thus, as seen in the above example, there is data sharing in both directions between a fixture and all child fixtures it is composed of. This is critical when you want to parameterize fixtures and reuse the same test code written in the Setup fixtures with different data values:
  - `FooSetup2` can pass an initialization value to `FooSetup1`, which it can see when it is run.
  - `FooSetup1` can modify the values inside the `NitsContext( )` associated with it.
  - `FooSetup2`, which will run after `FooSetup1`, can then read the values modified by `FooSetup1` inside `NitsContext( )`.
- The macros, `NitsSetup2`, `NitsSetup3`, `NitsSetup4`, and `NitsSetup5` let you define Setup fixtures composed of 2, 3, 4, or 5 child fixtures. They use similar syntax:

```
NitsSetup<2/3/4/5>( <fixtureName>,
                    <dataTypeOfFixture>,
                    <child fixture1,
                      <initialization value of child fixture1>,
                    <child fixture2>,
                      <initialization value of child fixture2>,
                    <child fixture3>,
                      <initialization value of child fixture3>,
                    <child fixture4>,
                      <initialization value of child fixture4>,
                    <child fixture5>,
                      <initialization value of child fixture5> )
```

- Note that when a specific Setup fixture runs, all child Setup fixtures that it is composed of will be run in left to right order of their definition **before** the fixture itself runs. All of them will run on the same stack, one above the other, and as a result, you can pass pointers to local variables around within `NitsContext( )`. All local variables in child fixtures are still available when the parent fixture runs. The stack trace below shows how the stack might look like when running the fixtures above:

```
(gdb) bt
```

```
#0  FooSetup2 (_NitsContext=0x60e990)
       at NitsNewInterfaceTests.cpp:489
#3  0x00007ffff76c4e3b in FooSetup1 (_NitsContext=0x60e950)
       at NitsNewInterfaceTests.cpp:485
#13 0x0000000000400dd2 in main (argc=6, argv=0x7fffffffe558)
       at nits.cpp:146
```

- If you do not want to associate a data type with a specific Setup fixture, you can use the NITS-defined empty data type `NitsEmptyStruct` as the data type and the NITS-defined initialization value `NitsEmptyValue` to initialize the empty data type fixture.

D.9.5.4 How to Re-use Setup Fixtures in Multiple Files

The `NitsDeclSetup<N>/NitsDefSetup<N>` macros allow you declare fixtures in a `.h` file and define them in `.cpp` files. This lets you reuse the same Setup fixture in multiple files.

Here is an example:

```
Foo.h =>

struct MyStruct
{
    int x;
};

NitsDeclSetup0(Fixture0, MyStruct);

Foo.cpp =>

NitsDefSetup0(Fixture0, MyStruct)
    NitsContext()->_MyStruct->x = 0;
NitsEndSetup
```

- In the above example, `Fixture0` is declared with data type `MyStruct` in `Foo.h`.
- `Fixture0` is defined in `Foo.cpp`.
- Note that `Fixture0` can be used in other `.cpp` files to construct other Setup/Split/Test fixtures.

D.9.6 Split Fixtures

Using Split fixtures, you can split the execution of tests into two or more tests. This lets you execute the same test code in multiple configurations. A Split fixture that splits a test into two child fixtures starts with a `NitsSplit2` macro having the following syntax:

```
NitsSplit2( <fixtureName>,
            <data type of fixture>,
            <name of child fixture 1>,
            <name of child fixture 2> )
```

Place the test code after this macro, and end it with the `NitsEndSplit` macro.

Below is an example of a Split fixture composed of two child fixtures that define two child configurations in which the test will run:

```
struct MyContext
{
    int a;
};

NitsSetup0(MySetup1, MyContext)
    NitsContext()->_MyContext->a = 4;
NitsEndSetup

NitsSetup0(MySetup2, MyContext)
    NitsContext()->_MyContext->a = 8;
NitsEndSetup

NitsSplit2(MySplitSetup, MyContext, MySetup1, MySetup2)
    NitsAssert((NitsContext()->_MyContext->a == 4) || (NitsContext()-
>_MyContext->a == 8), PAL_T("value is wrong"));
NitsEndSplit
```

- Here `MySetup1` and `MySetup2` are two Setup fixtures that have data type `MyContext`.
- Each of them assigns a value of 4 or 8 to the variable `a` inside the `NitsContext` data object.
- The `NitsSplit2/NitsEndSplit` macros provide syntax for defining a Split fixture. `MySplitSetup` is a Split fixture that lets you split the test into two Setup fixtures `MySetup1` and `MySetup2`.
- NITS automatically runs the test in the `MySplitSetup` fixture in two configurations, once with `MySetup1` and next with `MySetup2`.
- Note that the Split fixture must have the same data type as the two or more child fixtures it is composed of. In this example, that data type is the C `struct MyContext`.
- Note that the split fixture syntax doesn't allow you to pass an initialization value to the child fixtures. This is because the initialization value in the child fixtures will be the same as the initialization value that the split fixture gets from the parent fixture.
- NITS makes sure that the `NitsContext( )->_MyContext` inside the split fixture `MySplitSetup` automatically points to the correct child fixture's context (`MySetup1` or `MySetup2`'s context) when the test is run. This allows you to assert that the value of `NitsContext( )->_MyContext->a` will be either 4 or 8 when running `MySplitSetup` since the child fixtures `MySetup1/MySetup2` set it up that way by updating it to 4 and 8 respectively.
- The `NitsSplit3`, `NitsSplit4`, and `NitsSplit5` macros let you define split fixtures which will split into 3, 4, and 5 other child fixtures and follow a similar syntax:

```
NitsSplit<3 or 4 or 5>( <fixtureName>,
                        <data type of fixture>,
                        <name of child fixture 1>,
                        <name of child fixture 2>,
                        <name of child fixture 3>,
                        <name of child fixture 4>,
                        <name of child fixture 5> )
```

- Similar to Setup fixtures, you can also separate the declaration and definition of Split fixtures by using the `NitsDeclSplit<N>/NitsDefSplit<N>` macros.
- Some common usage scenarios for Split Fixtures are:
  - You have a client server test which you need to run with the server started in different configurations (*e.g.* hosted in a separate process or in the test process itself).
  - You have some test that you need to run over multiple Auth modes or different character sets, or over different protocols like HTTP, HTTPs, and so on.

D.9.7 Test Fixtures

Test is just one other kind of fixture in NITS. The implementation section above already defined `NitsTest/NitsEndTest` syntax for basic tests in NITS. This section describes syntax for more advanced tests that are composed of one or more other `Setup` or `Split` fixtures. A Test fixture composed of one child fixture starts with a `NitsTest1` macro having the following syntax:

```
NitsTest1( <testFixtureName>,
           <child fixture 1>,
           <initializer for child fixture 1> )
```

Test code follows, and the fixture is ends with the `NitsEndTest` macro. Here is an example:

```
struct FooStruct1
{
  int i1;
};

NitsSetup0(FooSetup1, FooStruct1)
  NitsAssert(NitsContext()->_FooStruct1->i1 == 20,
             PAL_T("wrong value"));
  NitsContext()->_FooStruct1->i1 = 35;
NitsEndSetup

struct FooStruct1 sFooStruct1 = {20};

NitsTest1(FooTest, FooSetup1, sFooStruct1)
    NitsAssert( NitsContext()->_FooSetup1->_FooStruct1->i1 == 35,
                PAL_T("wrong value"));
NitsEndTest
```

- In the above example, `FooSetup1` is a setup fixture with data type `FooStruct1`.
- The `NitsTest1/NitsEndTest` macro pair determine the syntax for defining a test composed out of one child fixture.
- `FooTest` is the name of the test, which is composed of `FooSetup1` and passes `sFooStruct1` as the initialization value to `FooSetup1`.
- When the test is run, `FooSetup1` runs first, followed by `FooTest` on the same stack, as mentioned earlier.

- `NitsContext( )` inside the test body also follows a format similar to that of a Setup fixtures, in that it has auto-created variables of the format `_<Child fixture name>` that point to `NitsContext( )` data objects inside the child fixture.
- Note that there is no data type associated with a Test fixture; its `NitsContext( )` only lets you access the `NitsContext( )` data objects inside child fixtures.
- In above example, when `FooSetup1` runs, the value of variable `i1` inside the `NitsContext( )->_FooStruct1` data object is 20, since the test passes initializer `sFooStruct1` which sets it to 20.
- `FooSetup1` itself then updates `i1` to 35.
- Therefore, when the test body runs, the value of `i1` is 35. This illustrates data sharing between a Test fixture and its children in both directions.
- The `NitsTest2`, `NitsTest3`, `NitsTest4`, and `NitsTest5` macros let you define tests composed of 2, 3, 4, and 5 Setup/Split child fixtures. Child fixtures are run in the left-to-right direction in which they are defined. These macros use a syntax like the following:

```
NitsTest<2 or 3 or 4 or 5>( <testfixtureName>,
                            <child fixture 1>,
                            <initializer for child fixture 1>,
                            <child fixture 2>,
                            <initializer for child fixture 2>,
                            <child fixture 3>,
                            <initializer for child fixture 3>,
                            <child fixture 4>,
                            <initializer for child fixture 4>,
                            <child fixture 5>,
                            <initializer for child fixture 5> )
```

- The `NitsTestWithSetup` macro lets you define a simple test fixture composed of a child Setup fixture defined using `NitsSetup`. `NitsTestWithInitializableSetup` lets you define a test composed of one Setup fixture taking initialization data. For all the `Nits*` macro definitions, see the `nits.h` header file in the `<topleveldir>/nits/base/` directory.
- Note that the interface lets you construct arbitrary directed acyclic graphs (DAGs) of Test fixtures. It is possible that the same Setup/Split fixture can appear multiple times in such a graph without causing any problem, because NITS instantiates and runs a fixture only once. As a result, you can be sure of not having multiple executions of the same fixture or having multiple copies of its `NitsContext( )` object.

D.9.8 Cleanup Fixtures

Cleanup fixture can be defined on any of the above type of fixtures (*i.e.* on Setup, Split, Test, or ModuleSetup fixtures). The Cleanup fixture is run at the end of the test body for Setup, Split, and Test fixtures, and at the end of the test module for ModuleSetup fixtures. Because a Cleanup fixture has access to the same `NitsContext( )` as the fixture for which it is cleaning up, you can use it to clean up anything you need to.

A Cleanup fixture begins with a `NitsCleanup` macro that uses the following syntax:

```
NitsCleanup( <fixtureName for which the Cleanup fixture is defined> )
```

The code for the body of the test follows this start macro, and `NitsEndCleanup`. Here is an example:

```
NitsSetup(MySetup1)
  NitsTrace(PAL_T("MySetup1 being run"));
NitsEndSetup

NitsCleanup(MySetup1)
  NitsTrace(PAL_T("Cleanup for MySetup1 being run"));
NitsEndCleanup
```

- Here the Cleanup is defined on a Setup fixture, `MySetup1`.
- The syntax starts with the `NitsCleanup` macro, and ends with the `NitsEndCleanup` macro.
- The name of the Cleanup fixture is same as the fixture for which it is defined.
- In a test hierarchy, each fixture could have a Cleanup fixture defined. In that case the Cleanup fixtures are run in the reverse order in which the fixtures themselves are executed (*e.g.* `Setup1=>Test1=>Cleanup(Test1)=>Cleanup(Setup1)`.

D.9.9 ModuleSetup Fixtures

ModuleSetup fixtures let you define test code that is run at the beginning and end of the test module. This is useful in places where you want to run some piece of test code only once per test module at the beginning and end of it.

Here is an example:

```
NitsModuleSetup(MyModuleSetup1)
    NitsTrace(PAL_T("MyModuleSetup1 being run"));
    NitsAssert(PAL_TRUE, PAL_T(""));
NitsEndModuleSetup

NitsCleanup(MyModuleSetup1)
    NitsTrace(PAL_T("Cleanup for MyModuleSetup1 being run"));
NitsEndCleanup
```

- `MyModuleSetup1` defines a ModuleSetup fixture that will be run at the beginning of the module in which it is located.
- As with all other fixtures, you can define Cleanup fixtures also on ModuleSetup fixtures. They will be run at the end of the module.
- You can have multiple ModuleSetup fixtures in a test module(shared object library) and all of them will be run sequentially at the beginning of the module.
- In the case of test isolation mode, *i.e.* running tests in their own processes, the ModuleSetup is run per test process.
- Currently, this feature has limited capabilities. For example, it doesn't let you pass data around or compose ModuleSetup fixtures out of other fixtures. There are plans to add more functionality in the future.

D.9.10 A Note about C++ Tests

In the case where functions to be tested are inside a class written in C++, you need to write C wrappers for those functions. In `class1`:

```
class class1
{
  int foo1();
};
```

The corresponding NITSs C++ file containing traps should look something like this:

```
PAL_BEGIN_EXTERNC
    class1* construct_class1()
    {
        return new class1();
    }
    int class1_foo1()
    {
        class1* obj = construct_class1();
        return obj->foo1();
    }
PAL_END_EXTERNC

NitsTrapValue(class1Traps)
        class1_foo1
NitsEndTrapValue
```

The corresponding NITS traps header file should look something like this:

```
NitsTrapTable(class1Traps,0)
       int (NITS_CALL* _class1_foo1)();
NitsEndTrapTable

NitsTrapExport(class1Traps)
```

A NITS test for the above product class might look like the following (assuming that the product binary is named `libfoo.so`):

```
struct Ptr
{
  void* ptr;
};

Ptr PtrVal = {NULL};

NitsSetup0(MyModuleSetup1, Ptr)
  NitsTrapHandle h = NitsOpenTrap("libfoo.so", class1Traps);
  NitsAssert(h != NULL, PAL_T("Failed to load cla11Traps"));
  NitsContext()->_Ptr->ptr = h;

  // Optionally, you could call a helper function too:
  CallHelperFunction()
NitsEndSetup

NitsCleanup(MyModuleSetup1)
```

```
        NitsTrapHandle h = NitsContext()->_Ptr-ptr;
        if(h != NULL)
                NitsCloseTrap(h);
NitsEndCleanup

NitsTest(MyModuleTest1, MyModuleSetup1, PtrVal)
   NitsTrapHandle h = NitsContext()->_MyModuleSetup1->_Ptr->ptr;
   int result = NitsGetTrap(h, class1Traps, _class1_foo1)();
   NitsAssert(result != 1, PAL_T("Test failed"));
NitsEndTest
```

D.9.11 Automatic Fault Simulation

NITS provides automatic fault simulation functionality, which works as follows:

- The user annotates the places where the Faults should be simulated by using calls to `NitsShouldFault`.
- When NITS tests are run with fault simulation enabled (either by using the `-fault` switch or by using `NitsEnableFaultSim` inside the test body), the test body is run multiple times.
- In the first run, no faults are simulated (*i.e.* all calls to `NitsShouldFault` return FALSE).
- The test body is then run in a loop multiple times, and in each iteration, one specific `NitsShouldFault` call returns TRUE.
- The user can simulate failure behavior by failing the specific external API or internal function when the corresponding `NitsShouldFault` call returns TRUE.
- Below is an example of a function using this functionality.
- By default, only the test body is only run in a loop. However, if the test body uses the macro `NitsFaultSimMarkForRerun`, the entire test hierarchy is run in a loop. That helps in scenarios where the Setup/Cleanup fixtures need to be executed in every fault simulation iteration.
- Before running the tests in fault simulation mode, run `nits -install`. This sets up a file that indicates to `libpal.a` that it should load `libnitsinj.so`, which patches the stubbed version of the NITS API table in product and test binaries with the actual NITS implementation.
- All binaries that are linked with NITS using HOOK_BUILD should be specified using the `-target` switch if they need to be fault-simulated.
- OMI tests already use this functionality for all test binaries.

```
int Foo()
{
  if(NitsShouldFault(NitsHere(), NitsAutomatic))
  {
    // Simulate failure path behavior
    // failure return
    return -1;
  }
```

```
    // continue with implementation

    // success return
    return 0;
}
```

D.10 Enabling Logging during Unit Testing with NITS

On the NITS command line, use the `+loglevel:<value>` option to set the level of logging detail (`loglevel`) to one of the following values, which are NOT case-sensitive:

```
0
1
2
3
4
5
FATAL
ERR
WARNING
INFO
DEBUG
VERBOSE
```

For example:

```
output/bin/nits +loglevel:5 libtest_base.so
output/bin/nits +loglevel:verbose output/lib/libtest_miapi.so
```

The resulting log files are saved in the `<topleveldir>/output/var/log/` directory as follows:

- Logs from `omiserver` appear in `omiserver.log`. Individual unit tests start the server when required, and the `loglevel` passed in is the same as the test process"s current `loglevel`.
- Logs from `omiagent` appear in `omiagent*.log`.
- Logs from tests involving the client stack (using the MI API) appear in `miclient.log`.
- For tests that do not involve the client stack in the test process (*i.e.* everything other than `test_cli` and `test_miapi`), the test process prints its logs to `omitest.log`.
- If you want to see output of test process in the console itself, you can use the `+logstderr:1` option on the NITS command line. For example:

  ```
  output/bin/nits +loglevel:verbose +logstderr:1 \
                  output/lib/libtest_base.so
  ```

  Remember, however, that this works only for tests that do not use the MI API (*i.e.* everything other than test_cli and test_miapi). MI-API code closes the log every time the test closes the MI application. As a result, even if the test library tried to set the log to `stderr` so as to print to the console, output would be routed to the `miclient.log` every time the MI application closes and re-opens.

Note: Because `+loglevel:<value>` and `+logstderr:1` are not built-in NITS command-line options, they are not included in NITS usage help. They are implemented through the NITS module setup/cleanup constructs that let us run code at the beginning and end of a test module run. In this way, we use a common Setup/Cleanup fixture defined inside the `ut\` directory to set the loglevels and route the log data to `stderr` at the beginning of the module level setup, and to close the log during the module level cleanup.

Below is an example of using `NitsModuleSetup/NitsCleanup` from ut\omitestcommon.cpp. You can have multiple module-level setups and cleanups in a module. All Module setups will be run at the beginning of the test module and all cleanups at the end after running all tests within the module.

```
NitsModuleSetup(OMITestSetup)
  NitsTrace(PAL_T("OMITestSetup being run"));
  if(NitsTestGetParam(PAL_T("logstderr")))
  {
    Log_OpenStdErr();
  }
  else
  {
    PAL_Char finalPath[PAL_MAX_PATH_SIZE];
    /* Create name and Open the log file */
    if((CreateLogFileNameWithPrefix("omitest", finalPath) != 0) ||
      (Log_Open(finalPath) != MI_RESULT_OK))
    {
      NitsTrace(
        PAL_T("failed to open log file; routing output to stderr"));
      Log_OpenStdErr();
    }
  }
  const PAL_Char *loglevelParam = NitsTestGetParam(PAL_T("loglevel"));
  if(loglevelParam && Log_SetLevelFromPalCharString(loglevelParam) != 0)
  {
    NitsTrace(
      PAL_T("loglevel parameter invalid; not setting loglevel"));
    NitsAssert(PAL_FALSE, PAL_T("loglevel parameter invalid"));
  }
  NitsAssert(PAL_TRUE, PAL_T(""));
NitsEndModuleSetup

NitsCleanup(OMITestSetup)
  NitsTrace(PAL_T("Cleanup of OMITestSetup being run"));
  Log_Close();
  NitsAssert(PAL_TRUE, PAL_T(""));
NitsEndCleanup
```