Solstice Enterprise Manager Application Development Guide

2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

SunSoft
A Sun Microsystems, Inc. Business

Please
Recycle

Contents

Figures

Tables

Preface

The *Solstice Enterprise Manager Application Development Guide* provides an overview of the Solstice™ Enterprise Manager™ (Solstice EM) development environment and examples of its use. It is a companion document to the *Solstice Enterprise Manager API Syntax Manual*, which provides a definitive list of the Solstice EM product's Application Programming Interface (API) classes, methods, and functions.

Who Should Use This Book

The document is intended for programmers developing applications who are very familiar with C++ and have experience with using complex programmatic interfaces.

Before You Read This Book

If you have just acquired the Solstice EM product, you should read the *Solstice Enterprise Manager Reference Manual* for an overview of the Solstice EM product functions, features, and components. You should also read the *Solstice Enterprise Manager 2.0 Release Notes* for information on installing and starting, compatibility and minimum machine and software requirements, known problems, an inventory of the product components, and late breaking information about the Solstice EM product.

How This Book Is Organized

This document contains the following chapters:

Chapter 1, “Introduction,” provides an introduction to the Solstice EM Development Environment.

Chapter 2, “Network Management Concepts,” describes basic concepts of network management that you need to understand in order to develop solutions using Solstice Enterprise Manager.

Chapter 3, “EM Programming Concepts,” describes some of the concepts associated with developing Solstice EM solutions.

Chapter 4, “Developing EM Solutions,” describes how to develop Solstice EM solutions and discusses issues to consider when developing those solutions.

Chapter 5, “Developing Object Behaviors,” explains how to develop object behaviors for a specific managed object class.

Chapter 6, “Debugging EM Code,” provides information about how to evaluate problems in your code and how to fix them.

Chapter 7, “Scenarios,” describes some problem scenarios associated with developing Solstice EM solutions.

Chapter 8, “API Examples,” presents a number of examples, showing the capabilities and breadth of the development environment’s classes, methods, and functions.

Chapter 9, “Protocol and Management Adaptors,” reviews some of the important concepts behind the Management Protocol Adaptor (MPA) and Protocol Driver Module (PDM) model.

Chapter 10, “Topology Database Service,” explains the Topology Database Service.

Chapter 11, “Writing RPC Agents for EM,” provides a high-level overview of the agent writing process.

Appendix A, “Terminology References,” provides a road map into the ISO specifications for terminology definitions.

Appendix B, "Access to Data in the MIS," explains how information in the MIS is accessed.

Appendix C, "em_debug Analysis," provides a walkthrough of a debugging session using the `em_debug` utility.

Appendix D, "Topology Database Architecture," explains the topology database, which stores topological information about the managed networked environments.

Conventions Used in This Book

This section describes the conventions used in this book.

What Typographic Changes and Symbols Mean

The following table describes the type changes and symbols used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`system%` **`su`**
		`Password:`

<AaBbCc123>	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *<filename>*`.`
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide.* These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

All command line examples in this guide use the C-shell environment. If you use either the Bourne or Korn shells, refer to `sh`(1) and `ksh`(1) man pages for command equivalents to the C-shell. The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	**Prompt**
C shell prompt	`host_name%`
C shell superuser prompt	`host_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Introduction 1

1.1 Introduction to the Development Environment

The Solstice Enterprise Manager (EM) product consists of a Management Information Server (MIS) that serves as a repository of information about the network, and a set of applications (or services) that use the MIS. Applications running on various workstations communicate with the MIS to get information about the network. The MIS plays the role of server to the applications and services that are its clients. Management functions are distributed between the MIS and its clients.

The Solstice EM Development Environment provides a set of Application Programming Interfaces (APIs), including the high-level Portable Management Interface (PMI) and low-level PMI, and other tools that enable a software developer to build comprehensive SNMP- and CMIP-based network management solutions using Solstice EM.

A Solstice EM *solution* is a collection of one or more user-configured or user-developed subcomponents that work together to solve a problem. If you are an application developer who is new to developing network management solutions using Solstice Enterprise Manager (EM), you are probably asking yourself, “Where do I start?” Solstice EM offers a very rich and powerful environment for developing solutions for your enterprise-wide network management problems. This richness can be intimidating until you come to recognize that you probably only need to focus on a subset of this environment at a given time.

1.2 Using the Solstice Enterprise Manager Application Development Guide

This document is designed to help you identify the Solstice EM components you need to work with in order to solve some of your initial problems and provide you with some sample components that you may want to use as a starting point when you develop your own solutions.

There are three main sections in this guide:

- The first section provides background information. This section includes “Network Management Concepts” and “EM Programming Concepts.” These chapter introduce network management and Enterprise Manager concepts that you need to be familiar with in order to develop Solstice EM solutions.
- The second section explains how to develop applications for the EM platform. It includes these chapters: “Developing EM Solutions,” “Developing Object Behaviors,” and “Debugging EM Code.”
- The third section includes the “Scenarios” and “API Examples” chapters. These chapters provide examples that illustrate how to create applications in the Solstice EM platform that solve specific problems.

This guide does not cover development of EM network management agents, although most of the network management concepts covered in this guide are also applicable to agent development. This guide does cover development of RPC Manager agents in “Writing RPC Agents for EM.”

1.3 Technical Definitions and Resources

For precise technical definitions and resources associated with managers, agents, SNMP, or CMIP, refer to the specifications, RFCs and books listed in Appendix F, “Additional Sources of Information” in the *Solstice Enterprise*

Manager Reference Manual. The Network Management Form OMNIPoint specification is also an excellent reference source that is easier to read and work with than the CCITT and ISO specifications.

Network Management Concepts 2

2.1 Network Management Concepts

If you are not familiar with network management, this chapter presents some basic concepts you need to understand in order to develop applications and solutions using Solstice EM. If you are already familiar with network management, this chapter is a review of the concepts you need to keep in mind as you develop applications and solutions using Solstice EM.

Some of the key terms that will be covered in this section are:

- Manager or Manager Role
- Agent or Agent Role
- Request
- Response
- Notification
- Network Management Protocol
- Management Information Base or MIB

2.1.1 General Concepts

The basic network management model (or architecture) is a starting point for understanding network management and Solstice EM. Four fundamental concepts of this model are:

- Manager or Manager Role
- Agent or Agent Role
- Network Management Protocols
- Management Information Base (MIB)

These concepts are used by network management protocols, including CMIP, SNMP, and SunNet Manager RPC protocols. The basic model and concepts are shown in Figure 2-1 and described in the sections that follow.

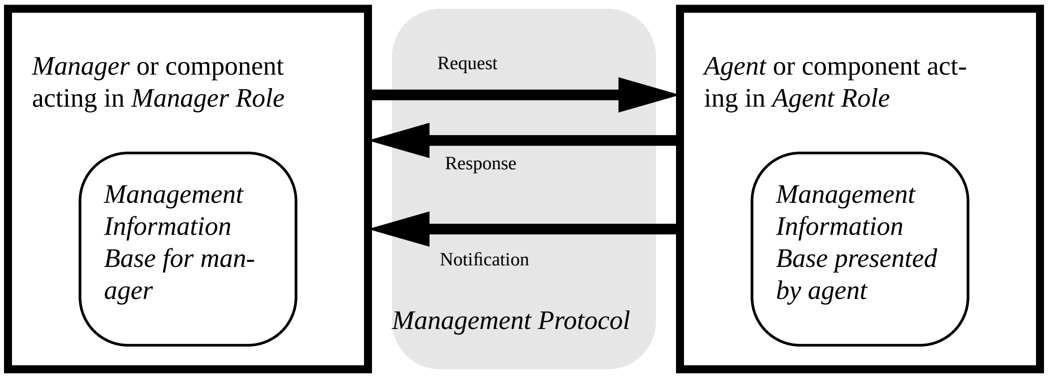

Manager or component acting in *Manager Role*
Management Information Base for manager
Request
Response
Notification
Management Protocol
Agent or component acting in *Agent Role*
Management Information Base presented by agent

Figure 2-1 Network Management Model

2.1.1.1 Manager or Manager Role

In the network management model, a *manager* is a unit that:

- Provides information to users

- Issues *requests*[1] to devices in a network. A request is used to ask a device to take some action. Typically the action requested is for a device to respond with specific information requested by the manager.
- Receives *responses* to the requests
- Receives unsolicited information from devices in the network concerning the status of the devices. These unsolicited reports are referred to as *notifications* and are frequently used to report problems, abnormalities, or changes in the agent environment.

Performing these activities is also referred to as acting in the *manager role*. Generally, Solstice EM operates in the manager role for each of the management protocols it supports.

2.1.1.2 Agent or Agent Role

In the network management model, an *agent* is a unit that:

- Is part of a device in the network that monitors and maintains status about that device
- Can act upon and *respond* to requests from a manager
- Can provide unsolicited information (or *notifications*) to a manager

Performing these activities is sometimes referred to as acting in the *agent role*. Solstice EM can act in the agent role for ISO (CMIP) management. EM can also act in the agent role to a limited extent for Internet (SNMP) management.

2.1.1.3 Network Management Protocols

Managers and agents require some form of communication to issue their requests and responses. SNMP is the protocol used to issue requests and receive responses in a TCP/IP network. CMIP is the protocol used in ISO networks. CMIP and SNMP define:

- Types of requests and responses that can be issued (for example, get, set, get response, and set response)

1. The request defined here is a management protocol request and should not be confused with Solstice EM Request Templates or Requests.

- Who can issue requests and responses
- Wording to use when issuing requests and responses (the syntax and encoding of each request and response)
- How the requests and responses are exchanged (for example., using OSI or TCP/IP network protocols to pass the requests and responses back and forth)

Both SNMP and CMIP specify ASN.1 as the language used to encode and decode request and response messages.

Other management protocols besides CMIP and SNMP exist. These other protocols are used primarily to manage devices that existed before SNMP and CMIP became available and are referred to as *legacy* or *proprietary* protocols. These other protocols also define functions and services similar to those described for CMIP and SNMP. EM supports both CMIP and SNMP management protocols and the SunNet Manager RPC management protocol. EM can also be extended to support other management protocols.

2.1.1.4 *Management Information Base (MIB)*

In addition to being able to pass information back and forth, the manager and the agent need to agree on and understand what information the manager and agent each receive in any exchange. This information includes:

- The *attributes* or types of data that can be supplied by an agent to a manager
- The *operations* or *actions* performed by an agent that can be requested by a manager
- The *behavior* exhibited by the agent
- The *notifications* or the types of unsolicited information an agent can send to a manager

This information varies for each type of agent. For instance, an SNMP agent running on a Synoptics hub will be described by one set of attributes and actions, and an SNMP agent running on a Cisco router will be described by a different set of attributes and actions.

The collection of this information is referred to as the *management information base*. The ISO standards organization defines a management information base in ISO/IEC 7948-4 as follows, “The conceptual repository of management information within an open system.” A manager normally contains management information describing each type of agent the manager is capable of managing. This information would typically include Internet MIB definitions and ISO GDMO definitions for managed objects and agents. An agent typically presents (or contains) management information for one type of device, although this information can include descriptions and data for several types of devices.

2.1.2 ASN.1

ASN.1 is an acronym for “Abstract Syntax Notation One” which is an ISO defined language for defining types of data or *datatypes*. A set of encoding rules is also associated with ASN.1 which describe how the ASN.1 datatypes are encoded for transfer between machines in a network.

ASN.1 is used by both ISO and Internet network management for the following purposes:

- To describe management information (GDMO and Concise MIB)
- To describe request, response, and notification messages (CMIP and SNMP)

CMIP, GDMO, SNMP, Concise MIB, SNMPv2 and SNMPv2 MIBs are all defined using ASN.1.

2.1.2.1 ASN.1 Encoding Rules

ASN.1 is encoded using rules specified in CCITT X.209 ISO / IEC 8825 Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1). The set of rules defined in this standard are frequently abbreviated as BER (Basic Encoding Rules). ASN.1 values encoded using ASN.1 BER generally have three fields:.

- The *tag* identifies what type of data is contained in the value field. The ASN.1 BER specification refers to the tag field as the *identifier octets*[1] although is more commonly referred to as a tag.

1. The standards use the term octets instead of bytes. An octet is eight bits. A byte is the smallest addressable unit in a computer’s memory. Most application developers consider the term byte to refer to eight bits, and most computers on the market today use eight bit bytes. When working with Solstice EM you can think of octets and bytes as meaning the same thing.

- The *length* field indicates the number of bytes (or octets using standards terminology) used for the value field[1]. The length is referred to as the *length octets* in the ASN.1 BER.
- The *value* field contains a data value of the type specified by the tag field and having a length in bytes as specified by the length field. The value field is referred to as the *contents octets* in the ASN.1 BER specification.

Solstice EM uses the terms tag, length, and value, rather than referring to identifier octets, length octets, and the contents octets respectively. Figure 2-2 shows an example of the number six encoded as an ASN.1 BER integer (Note: a tag of 2 is used to specify an integer value).

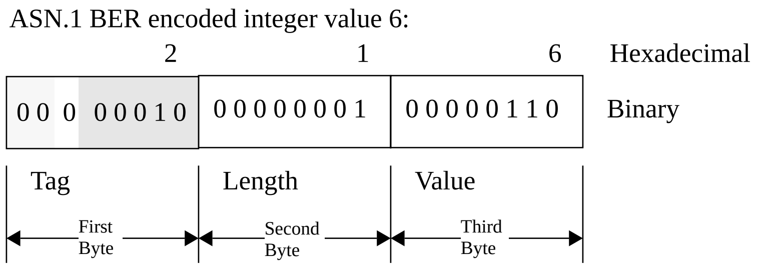


Figure 2-2 ASN.1 BER Encoding Example

The tag field is further subdivided into three subfields, represented by the shaded areas in the first byte shown in Figure 2-2.

- The first subfield is the class of the tag and is contained in the first two bits of the tag.
- The second subfield specifies whether or not the tag is for a constructed type and is one bit in length.
- The third field is the tag number which specifies the type of data contained in the value field and is five bits in length.

The subfields are a slightly advanced topic that you should be aware of but do not need to understand in detail. The remaining sections provide some additional overview of the ASN.1 language and BER encoding. For more

1. ASN.1 also supports the concept of indefinite length. Refer to the CCITT X.209 ISO / IEC 8825 specification for more information on this concept.

detailed information than is presented here refer to CCITT X.209 ISO / IEC 8825 Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).

2.1.2.2 *ASN.1 Datatypes and Tag Numbers*

A datatype is a type of value, such as integer, real, or string. ASN.1 defines a number of datatypes and assigns a *tag number* to identify each of the datatypes. The ASN.1 datatypes and tag numbers are shown in Table 2-1. CMIP supports all ASN.1 datatypes. SNMP Version 1 supports a subset of the datatypes (shown using light shading in Table 2-1). SNMP Version 2 supports the same datatypes as SNMPv1 plus one additional datatype (the additional datatype is shown using dark shading in the table).

Table 2-1 ASN.1 Datatypes and Tag Numbers

Tag Number	**ASN.1 Datatypes**	**Primitive / Constructed**
1	BOOLEAN	Primitive
2	INTEGER	Primitive
3	BIT STRING	Either
4	OCTET STRING	Either
5	NULL	Primitive
6	OBJECT IDENTIFIER	Primitive
7	OBJECT DESCRIPTOR	Primitive
8	EXTERNAL	Primitive
9	REAL	Primitive
10	ENUMERATED	Primitive
11-15	Reserved for future use	
16	SEQUENCE, SEQUENCE-OF	Constructed
17	SET, SET-OF	Constructed
18	NumericString	Either
19	PrintableString	Either

Table 2-1 ASN.1 Datatypes and Tag Numbers

Tag Number	ASN.1 Datatypes	Primitive / Constructed
20	TeletexString	Either
21	VideoTexString	Either
22	IA5String	Either
23	UTCTime	Either
24	GeneralizedTime	Either
25	GraphicsString	Either
26	VisibleString	Either
27	GeneralString	Either
28	CharacterString	Either
29	Reserved for future use	

2.1.2.3 *Primitive and Constructed Datatypes*

An ASN.1 value is a *primitive type* if its value field directly represents the value specified by the tag. Primitive types are sometimes referred to as *simple types*. An ASN.1 value is a *constructed type* if its value field contains a combination of one or more simple types. Constructed types are sometimes referred to as *structured types*. Table 2-1 lists whether each ASN.1 datatype is a primitive type, a constructed types, or whether it can be used for either type.

The encoded integer values (6 and 7) shown in Figure 2-2 are examples of primitive datatypes. The encoded SET-OF shown in Figure 2-3 is an example of a constructed type. The SET-OF is constructed from the two integer values.

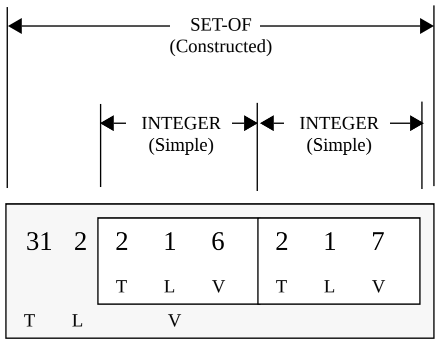


Figure 2-3 ASN.1 BER Constructed Value

2.1.2.4 *Classes of Tags*

ASN.1 defines the following four classes of tags:

- Universal Class—Universal class tags are defined by the ASN.1 international standards (CCITT X.208 ISO / IEC 8824 and CCITT X.209 ISO / IEC 8825). Each universal class tag is assigned to a unique datatype or is assigned to a datatype used to construct new datatypes. Each tag number shown in Table 2-1 is a universal class tag.
- Application Class—Application class tags are assigned to datatypes by other international standards. An application class tag is unique within a standard. An example of an application class tag is the IpAddress datatype used for Internet network management applications, which is defined in an Internet RFC.
- Context-specific Class—Context-specific tags are interpreted within the context in which they are used. They are normally used with constructed datatypes such as SET or SEQUENCE. In the case of SET or SEQUENCE, a context-specific tag only has meaning within the context of the SET or SEQUENCE.

- Private Class—Private class tags are not assigned by international standards. Private class tags can be used by enterprises to define proprietary datatypes.

The key thing to be aware of is that four classes of tags exist and that the ASN.1 specifications define universal class tags.

Note – The debugging and tracing facilities provided with Solstice EM displays tags using a combination of the class and the tag number. For instance, a context-specific class which used the value 7 for some specific meaning would be displayed as C7. Application class tags would be displayed as A1, A2, A3, etc. Private tags would be displayed as P1, P2, P3, etc. *Some of the Universal class tags are displayed using text identifiers. For instance, an OID tag displays as OID rather than as U6. These conventions are also used throughout this document.*

2.1.2.5 *OBJECT IDENTIFIER Datatype (OID)*

The OBJECT IDENTIFIER datatype (commonly abbreviated as OID) is used to provide unique labels. Object identifiers are used extensively by GDMO, CMIP, Concise MIB, and SNMP and are also referred to and defined in a number of CCITT and ISO standards. OIDs are used extensively by Solstice EM and referred to throughout the EM documentation. Object identifiers are generated from a global naming tree, commonly referred to as either the Object Identifier Tree or the Registration Tree. One of the key things to understand is that an *OID is a globally unique label.*

The Object Identifier Tree contains nodes labeled using nonnegative integer values and a text label. The top of this tree is called the root (this is actually an inverted tree, with root being at the top, branches going down the tree, and leaf nodes being at the bottom). There are three labeled nodes under root which are administered by the ISO and CCITT standards organizations. There are in turn sub-nodes under the three ISO and CCITT nodes some of which are administered by other organizations. A unique label can be formed for any node in the tree by concatenating the name of each node in the tree, starting at root, and proceeding down the tree to a node in the tree. The label formed using this method is an OBJECT IDENTIFER. Figure 2-4 shows a portion of the global naming tree.

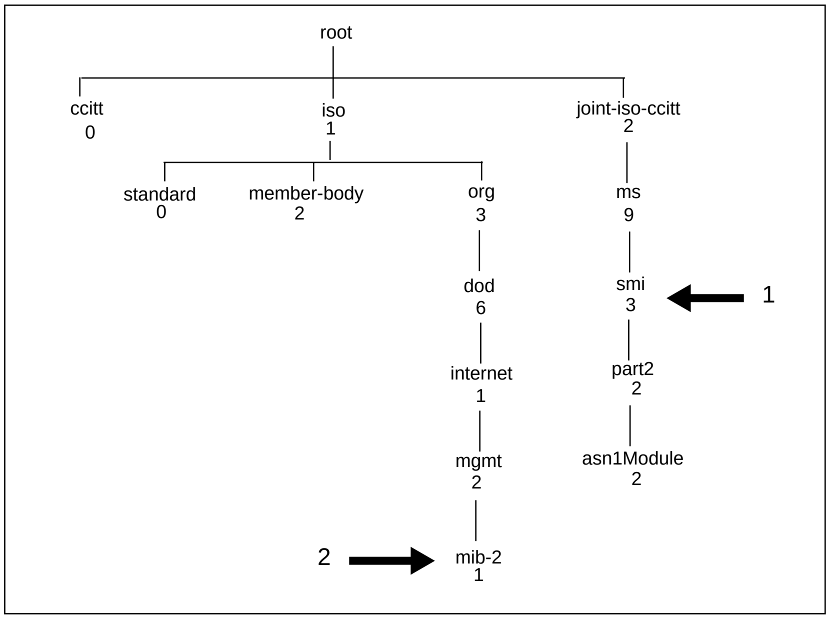


Figure 2-4 Object Identifier (Registration) Tree

Human Readable Notation For OIDs

Object identifiers are typically written out using one of the following notation types:

- The first notation uses the integer label for each node in the registration tree in the path from root to the node of interest, separated by periods.

Example 1: The OID for the node smi(3) in the above figure (pointed to by arrow number one) would be written as follows:

```
2.9.3
```

Example 2: The OID for the node mib-2(1) above (pointed to by arrow number two) would be written as follows:

```
1.3.6.1.2.1
```

- The second notation is a little more formal and uses a combination of the following forms separated by white space and surrounded by curly braces:
  - The node label (text) and the integer label
  - Just the node label
  - Just the integer label

  Example 1: The OID 2.9.3 could be written using any of the following forms:

```
{2 9 3}
{joint-iso-ccitt(2) ms(9) smi(3)}
{joint-iso-ccitt ms smi}
{joint-iso-ccitt ms(9) 3}
```

  Example 2: The OID 1.3.6.1.2.1 could be written using any of the following forms:

```
{iso(1) org(3) dod(6) internet(1) mgmt(2) mib-2(1)}
{iso(1) org 6 1 2 mib-2(1)}
{iso org dod internet mgmt(2) mib-2(1)}
```

- It is also common and legal to assign a label to a node (i.e., an OID) and then to use that label in subsequent assignments. For example, the node 2.9.3 (shown in Figure 2-4) used in the preceding examples could be assigned a label as follows:

```
smiISO OBJECT IDENTIFIER ::= {join-iso-ccit(2) ms(9) smi(3}
```

The smiISO label could then be used subsequently to name the two nodes shown below node 2.9.3 as follows:

```
{smiISO part2(2)}
{smiISO part2(2) asn1Module(2)}
```

BER Encoded OIDs

OIDs are encoded using ASN.1 BER. Solstice EM provides library functions to handle encoding and decoding of OIDs and other ASN.1 BER encoded values for you. You do not need to understand the BER encoding process to develop Solstice EM applications. If you are interested in finding out more about the actual mechanics of how the OID in encoded to produce the hexadecimal value, refer to CCITT X.209 ISO / IEC 8825.

BER Example: The ASN.1 BER encoded value for the OID 2.9.3 would be:

06025903_{16}

Breaking this out into tag, length, and value for purposes of readability would result in:

06_{16}	02_{16}	5903_{16}
Tag	Length	Value

2.2 ISO Network Management Concepts

This section provides an introduction to ISO network management. Key concepts introduced include:

- The ISO management model
- CMIS and CMIP
- Guidelines for the Definition of Managed Objects (GDMO)

The ISO management model uses tree structures for parts of its model. Understanding these tree structures and what they represent is important to understanding the ISO model. The tree structures described in this section include:

- Object class inheritance tree
- Object identifier (registration) tree
- Management Information Tree (MIT)

2.2.1 ISO Management Model

The ISO management model defines or uses the following terms and concepts:

- A *managed resource* is a device or logical unit in a network that can be managed. The device or logical unit likely contains more information and provides more services than are needed for managing the device (in other words, the device has some purpose or provides some service in addition to being manageable). You can think of a managed resource as a real device in your network, although that definition is limited.
- A *managed object* or *managed object instance* presents a view of a managed resource or a portion of a managed resource in your network. The managed object presents information needed to manage the resource. If you think of a managed resource as a real device, a managed object is an abstraction of that device.

  An ISO agent will typically contain or provide views of multiple managed objects. Figure 2-5 shows a managed resource and multiple managed objects that represent (or are an abstraction of) the resource.

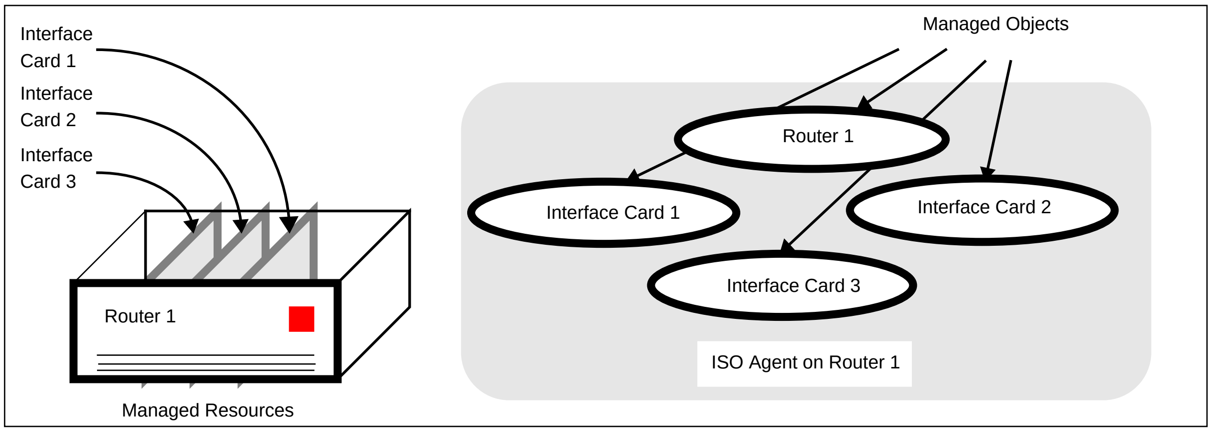


Figure 2-5 Managed Resources and Managed Objects

- A *managed object class* specifies the structure and behavior of a managed object. A managed object class is defined using the GDMO object definition guidelines (GDMO and managed object classes are described further in Section 2.2.3). Using object-oriented terminology, an instance of a managed-object-class is a managed-object-instance.
- *Inheritance* is a relationship between managed object classes. A managed object class can inherit the properties of one or more other managed object classes. A class that the properties are inherited from is a *base class* of the class that inherits the properties. A class that inherits the properties is a *derived class* of the class from which it inherited the properties.[1]
- *Containment* is a relationship between managed object instances. The containment relationship is used to name managed object instances. An example of one possible form of containment is shown in Figure 2-5 - the managed resource Router 1 *contains* three interface cards: Interface Card 1, Interface Card 2, and Interface Card 3. When the managed object instances shown in the figure are named, the naming could reflect the containment relationship of the router and the interface cards.

2.2.2 CMIS and CMIP

CMIS (Common Management Information Services) is a set of ISO-defined management services. CMIP (Common Management Information Protocol) is an ISO-defined management protocol. CMIS specifies types of requests, responses and notifications and defines what each request, response and notification can do. CMIP defines how those requests, responses and notifications are encoded (using ASN.1) into messages and specifies what operations are used to transport those encoded messages between managers and agents. A CMIP request typically specifies one or more managed objects to which the request is to be sent. To summarize, CMIS defines a set of services; CMIP defines a protocol used to encode and transmit information provided by the services.

1. The term superclass is sometimes used instead of base class and the term subclass is sometimes used instead of derived class. The terms superclass and subclass have largely been superceded by the terms base class and derived class. The superclass and subclass terminology still appears in a number of the CCITT and ISO / IEC standards.

2.2.2.1 CMIS Services and CMIP Messages

CMIS defines several management services. All of the CMIS services support confirmed request operations, and several also support unconfirmed operations. *Confirmed* request operations always require a response, regardless of the success or failure of the operation. *Unconfirmed* request operations never receive responses.

The CMIS specifications define the services listed in Table 2-2:

Table 2-2 CMIS Services

Service	Description	Operations
M-Get	Used by a manager to request information from an agent	Confirmed only
M-Cancel-Get	Used by a manager to request cancellation of a previously requested M-Get operation. M-Cancel-Get is typically used to cancel M-Get requests which specify scoping and filtering and expect multiple responses.	Confirmed only
M-Set	Used by a manager to request that an agent set attribute values of a managed object to specific values	Confirmed or unconfirmed
M-Create	Used by a manager to request that an agent create a managed object	Confirmed only
M-Delete	Used by a manager to request that an agent delete one or more managed objects	Confirmed only
M-Action	Used by a manager to request that an agent invoke a specific behavior supported by a managed object	Confirmed or unconfirmed
M-Event-Report	Notification used by an agent to send information to a manager	Confirmed or unconfirmed

Each CMIS service defines several CMIS primitives which map into CMIP messages or *protocol data units* (PDUs). Table 2-3 lists the mappings between the CMIS services, the CMIS primitives, and the CMIP messages.

Table 2-3 CMIS Services, CMIS Primitives, And CMIP Messages

CMIS Service	CMIS Primitive	CMIP Message Type
M-Get	M-Get Request	m-Get request PDU
	M-Get Response	m-Linked-Reply response PDU
		m-Get response PDU
M-Cancel-Get	M-Cancel-Get request	m-Cancel-Get-Confirmed request PDU
	M-Cancel-Get response	m-Cancel-Get-Confirmed response PDU
M-Set	M-Set Request	m-Set request PDU
		m-Set-Confirmed request PDU
	M-Set Response	m-Linked-Reply response PDU
		m-Set response PDU
M-Create	M-Create Request	m-Create request PDU
	M-Create Response	m-Create response PDU
M-Delete	M-Delete Request	m-Delete request PDU
	M-Delete Response	m-Linked-Reply response PDU
		m-Delete response PDU
M-Action	M-Action Request	m-Action request PDU
		m-Action-Confirmed request PDU
	M-Action Response	m-Linked-Reply response PDU
		m-Action response PDU
M-Event-Report	M-Event-Report Request	m-Event-Report request PDU
		m-Event-Report-Confirmed request PDU
	M-Event-Report Response	m-Event-Report response PDU

Table 2-3 does not list the error response messages that could result from each request or response operation. For a complete list of CMIS errors by type of service and for the complete list of CMIP messages and message formats, refer to the CCITT X.710 ISO / IEC 9595 and CCITT X.711 ISO / IEC 9596-1 specifications.

2.2.2.2 CMIS Functional Units

CMIS specifies services that managers and agents can support. However, managers and agents are not required to support all the services defined by CMIS. CMIS also defines a set of *functional units* that specify which CMIS services a manager or agent supports. When an ISO manager and an ISO agent first establish communication with each other, they agree on what types of services (as defined by functional units) can be used over the communication path between them. *You do not need to worry about functional units in order to develop Solstice EM solutions. However, the concepts presented the CMIS functional units are important to understand.* CMIS defines the following functional units, most of which EM supports:

1. The **multipleObjectSelection** functional unit pertains to CMIS requests that specify scoping. Scoping is a mechanism that allows a single request to be fanned out to several different managed objects. The fanned out request will result in a response being generated by each managed object that receives the request (see multipleReply below). Support for multipleObjectSelection is optional for ISO managers and agents. If a manager or agent supports multipleObjectSelection, the manager or agent must also support multipleReply. Solstice EM supports multipleObjectSelection in both manager and agent roles.

2. The **multipleReply** functional unit pertains to responses. A request that uses scoping (see multipleObjectSelection above) can result in multiple responses. The multipleReply functional unit specifies that an identifier field in each of the multiple responses, called the *linked identifier,* will indicate whether or not any response is the last response. Support for multipleReply is optional for ISO managers and agents. If a manager or agent supports multipleReply, the manager or agent must also support multipleObjectSelection. Solstice EM supports multipleReply in both manager and agent roles.

3. The **filter** functional unit pertains to CMIS requests that specify a test to be applied to a managed object before the request is carried out. If the test is successful, the request is performed. If the test fails, the request is not performed for that managed object. A filter is almost always used in a request that also uses scoping. Support for the filter functional unit is optional for ISO managers and agents. Solstice EM supports the filter functional unit in both manager and agent roles.

4. The **kernel** functional unit specifies support for the following CMIS request, response and notification services: M-Event-Report; M-Get, M-Set, M-Action, M-Create; and M-Delete. All ISO managers and agents (including Solstice EM) are required to support the kernel functional unit.

5. The **confirmedCancelGet** functional unit specifies support for the M-Cancel-Get CMIS service. Support for the confirmedCancelGet functional unit is optional for ISO managers and agents. Solstice EM supports confirmedCancelGet in the manager role and provides limited support in the agent role.

6. Solstice EM does not support the **extendedService**[1] functional unit and you do not need to worry about it to develop an application. For more information, refer to the CMIS specification (CCITT X.710 ISO 9595) and the specifications for the ISO communication model (CCITT X.200, X.210, and X.216).

2.2.3 Guidelines for the Definition of Managed Objects (GDMO)

This section is an introduction to GDMO and is not a comprehensive overview. The information included here is intended to explain enough so that you can look at a GDMO managed object class definition and have a basic understanding of the information the object class definition is specifying. For complete information about GDMO, refer to the following specification: CCITT X.722 ISO /IEC 10165-4, *The Guidelines for the Definition of Managed Objects.*

1. The ISO communication model defines seven different protocol layers. Each layer in this model exchanges information with the layer below it. The top layer in the ISO model is the application layer. In this model CMIP is in the application layer. The layer just below the application layer is the presentation layer. CMIP requests, responses, and notifications are packaged into P-Data wrappers provided by the presentation layer. In addition to providing P-Data wrappers, the presentation layer also provides additional services. The extendedService functional unit indicates that these additional services are available for use.

2.2.3.1 GDMO Templates

The GDMO guidelines provide templates for defining managed object classes and their supporting constructs (also referred to as supporting productions). The *managed object classes* defined using GDMO allow agents to describe what information and services they provide in a format that can be understood and used by a manager. GDMO is used to define the following information for managed object classes:

- Attributes or types of data supported by the managed object class
- Operations or actions supported by the managed object class
- Behavior exhibited by the managed object
- Notifications or the types of unsolicited information a managed object can generate and send to a manager

GDMO is based on an object-oriented model and uses object-oriented concepts including classes, instances, inheritance, constructors, destructors, attributes, and actions.

The are two basic GDMO templates:

- The MANAGED OBJECT CLASS template is used to define managed object classes.
- The NAME BINDING template is used to define name bindings, which specify containment for managed objects.

GDMO also defines several other templates that both the MANAGED OBJECT CLASS and NAME BINDING templates directly or indirectly reference.

2.2.3.2 Managed Object Class Template

The format of the MANAGED OBJECT CLASS template is:

```
<class-label> MANAGED OBJECT CLASS
    [DERIVED FROM  <class-label> [,<class-label>]*;
    ]

    [CHARACTERIZED BY<package-label> [,<package-label>]*;
    ]
    [CONDITIONAL PACKAGES<package-label> PRESENT IF
                   condition-definition
                   [,<package-label> PRESENT IF
                                  condition-definition]*;
    ]
REGISTERED AS object-identifier;
```

The capitalized words are GDMO keywords, the words in < > brackets represent information that is filled in when a class is defined, and any item in square braces represents an optional value. A closing square brace followed by an asterisk indicates that the item enclosed in braces may be repeated zero or more times. The meaning of each clause or parameter in the above template is shown in Table 2-4.

Table 2-4 MANAGED OBJECT CLASS Template Definitions

Item	**Description**
<class-label>	This label specifies the name of the GDMO managed object class. The REGISTERED AS clause in the template is used to associate a unique object identifier with this label.
MANAGED OBJECT CLASS	This clause identifies the template as a definition for a GDMO managed object class.
DERIVED FROM <class-label>	This clause identifies other GDMO managed object classes that are base classes of the managed object class being defined. The <class-label> parameters contain the names of the base classes.
CHARACTERIZED BY	This clause identifies the packages that the managed object class supports. A package defines a set of behaviors, attributes, operations, and notifications.
<package-label>	This label identifies the name of a GDMO package.
[CONDITIONAL PACKAGES	This clause identifies any packages that are conditionally supported by the managed object class. Conditional packages will be included in a managed object instance if the conditions specified by the PRESENT-IF clause are true.

Table 2-4 MANAGED OBJECT CLASS Template Definitions

Item	**Description**
<package-label> PRESENT IF	This clause identifies the condition-definitions that must be true in order for a conditional package to be included in a managed object instance.
condition-definition	This specifies a condition. If this condition is true, the conditional packages identified by the CONDITIONAL PACKAGES clause will be included in a managed object instance when it is *instantiated* (i.e., created).
REGISTERED AS	This clause identifies the globally unique name assigned to the GDMO managed object class.
object-identifier	This parameter is replaced with the name of the OBJECT IDENTIFIER that is used to globally and uniquely identify the GDMO name binding.

The other templates referenced directly or indirectly by the MANAGED OBJECT CLASS template are:

- PACKAGE
- PARAMETER
- ATTRIBUTE
- ATTRIBUTE GROUP
- BEHAVIOUR
- ACTION
- NOTIFICATION

2.2.3.3 *Name Binding Template*

The format of the NAME BINDING template is:

```
<name-binding-label> NAME BINDING
    SUBORDINATE OBJECT CLASS<class-label> [AND SUBCLASSES];
    NAMED BYSUPERIOR OBJECT CLASS<class-label> [AND SUBCLASSES];
    WITH ATTRIBUTE     <attribute-label>;
    [BEHAVIOUR          <behaviour-definition-label>
                        [,<behaviour-definition-label>]*;
    ]

    [CREATE             [create-modifier
                        [,create-modifier]]
                        [<parameter-label>]*;
    ]

    [DELETE             [delete-modifier]
                        [<parameter-label>]*;
    ]

REGISTERED AS object-identifier;
```

The capitalized words are GDMO keywords, the words in < > brackets represent information that is filled in when a class is defined, and any item in square braces represents an optional value. A closing square brace followed by an asterisk indicates that the item enclosed in braces may be repeated zero or more times.

The meaning of each clause or parameter from the NAME BINDING template is shown in Table 2-5.

Table 2-5 Name Binding Template Definitions

Item	Description
<name-binding-label>	This label specifies the name of the GDMO name binding. The REGISTERED AS clause in the template is used to associate a unique object identifier with this label.
SUBORDINATE OBJECT CLASS <class-label> [AND SUBCLASSES]	This clause identifies a GDMO managed object class whose instances can be named using an instance of the GDMO managed object class specified in the NAMED BY SUPERIOR OBJECT CLASS clause. The <class-label> subclause identifies the subordinate GDMO managed object class. The AND SUBCLASSES subclause, if included, specifies that instances of a derived class of the subordinate class can also be named using an instance of the class specified in the NAMED BY SUPERIOR OBJECT CLASS clause.
NAMED BY SUPERIOR OBJECT CLASS <class-label> [AND SUBCLASSES]	This clause identifies a GDMO managed object class whose instances can be used to name an instance of the GDMO managed object class specified in the SUBORDINATE OBJECT CLASS clause. The <class-label> subclause identifies the superior GDMO managed object class. The AND SUBCLASSES subclause, if included, specifies that instances of a derived class of the superior class can also be used to name an instance of the class specified in the SUBORDINATE OBJECT CLASS clause.

Table 2-5 Name Binding Template Definitions

Item	**Description**
WITH ATTRIBUTE <attribute-label>	This clause identifies the attribute that will be used to form the relative distinguished name (RDN) of an instance of the managed object class specified in the SUBORDINATE OBJECT CLASS clause. (Refer to Section 2.2.6.3 for a description of relative distinguished names.) This <attribute-label> identifies the attribute used to form the RDN for an instance of the object class specified by the SUBORDINATE OBJECT CLASS clause.
BEHAVIOUR <behaviour-definition-label>	This construct is used to specify any behavior impact that results specifically due to the use of the name binding. The <behaviour-definition-label> identifies the behavior definition.
CREATE	This clause specifies that an instance of managed object specified by the SUBORDINATE OBJECT CLASS clause can be created using a management operation (normally a CMIP m-Create operation).
create-modifier	This subclause is used to specify permitted options for an m-Create operation. The options are: WITH-REFERENCE-OBJECT, which specifies that a reference object may be specified in an m-Create operation; and WITH-AUTOMATIC-INSTANCE-NAMING, which specifies that the object instance name can be omitted from the m-Create operation.
<parameter-label>	This clause is used to identify name binding error parameters associated with the create or delete operations.
DELETE	This clause specifies that an instance of managed object specified by the SUBORDINATE OBJECT CLASS clause can be deleted using a management operation (normally a CMIP m-Delete operation).

Table 2-5 Name Binding Template Definitions

Item	Description
delete-modifier	This subclause is used to specify behaviors for a managed object instance when the instance is deleted. The options are: ONLY-IF-NO-CONTAINED-OBJECTS, which specifies that a delete operation will fail and return an error if there are any contained managed object instances under the instance being deleted; and DELETES-CONTAINED-OBJECTS, which specifies that a delete operation will fail and return an error if any of the managed object instances directly or indirectly under the instance being deleted are subject to the ONLY-IF-NO-CONTAINED-OBJECTS delete modifier.
REGISTERED AS	This clause identifies the globally unique name assigned to the GDMO managed object class.
object-identifier	This parameter is replaced with the name of the OBJECT IDENTIFIER that is used to globally and uniquely identify the GDMO managed object class.

The NAME BINDING template references only one other template: ATTRIBUTE

2.2.3.4 *Additional GDMO Templates*

Section 2.2.3.2 and Section 2.2.3.3 provided an overview of two GDMO templates. This section briefly describes the remaining GDMO templates:

- The PACKAGE template is used to define a combination of behavior definitions, attributes, attributes groups, operations, notifications, and parameters for subsequent inclusion in a MANAGED OBJECT CLASS template. A package can be referenced by more than one managed object class definition.
- The PARAMETER template is used to define parameter syntaxes for subsequent inclusion in PACKAGE, ATTRIBUTE, ACTION, and NOTIFICATION templates. A parameter can be referenced by one or more of each of the four templates.

- The ATTRIBUTE template is used to define attributes used by GDMO classes. A single attribute can be referenced by more than one managed object class definition.
- The ATTRIBUTE GROUP template is used to define a one or more attributes that can be referenced as a group. A managed object class definition can include all attributes of a group by referencing the group, rather than referencing each attribute individually. An attribute group can be referenced by more than one managed object class definition.
- The BEHAVIOUR template is used to describe the expected behavior for one or more operations supported by a managed object class.
- The ACTION template is used to define actions that can be performed by a managed object class. The action template defines actions that are performed using the CMIS m-Action service, and does not include actions or behaviors defined for other CMIS services, such as m-Get, m-Set, etc. An action can be referenced by more than one managed object class.
- The NOTIFICATION template is used to define notifications that can be emitted by a managed object class. A notification can be referenced by more than one managed object class.

For more information about these templates, refer to the GDMO specification (CCITT X.722 ISO/IEC 10165-4).

2.2.3.5 Definition of Management Information (DMI)

The CCITT X.721 ISO/IEC 10165-2 Definition of Management Information (DMI) defines several managed object classes, name bindings, and corresponding support constructs. The managed object classes defined in the DMI are base classes for defining other managed object classes (using the DERIVED FROM clause) or are required in order to support the system management functions defined in the CCITT X.730 ISO/IEC 10164-1 through CCITT X.736 ISO/IEC 10164-7.

One of the managed object classes defined by the DMI is *top*. The managed object class top is base class for every other managed object class defined for ISO systems management and cannot be instantiated directly. The definition of top includes two attributes: objectClass and nameBinding. The definition for top conditionally includes two additional attributes: packages (in the packagesPackage PACKAGE) and allomorphs (in the allomorphicPackage). This means that every managed object class supports the objectClass and nameBinding attributes and that every managed object instance is capable of responding to M-Get requests asking for the objectClass and nameBinding information. The conditional information in top also means that managed object instances might also provide a list of supported packages and allomorphic behaviors.

The managed object classes defined in the DMI are as follows:

- alarmRecord
- attributeValueChangeRecord
- discriminator
- eventForwardingDiscriminator
- eventLogRecord
- log
- logRecord
- objectCreationRecord
- objectDeletionRecord
- relationshipChangeRecord
- securityAlarmReportRecord
- stateChangeRecord
- system
- top

2.2.4 Object Class Inheritance Tree

The *Object Class Inheritance Tree* represents inheritance relationships between managed object classes. A managed object class that inherits the properties of another managed object class gains or includes the attributes, behaviors, operations, and notifications defined for the other managed object class. The class that inherits the properties is the derived class. The class from which the properties are inherited is the base class. This type of relationship is sometimes referred to as an *is-a* relationship because the derived class *is a* base class. For example, a discriminator is-a top, because it supports all the properties supported by top.

The base class is sometimes referred to as a superclass and the derived class is sometimes referred to as a subclass. The superclass and subclass terminology has largely been replaced by the base-class and derived-class terminology but still appears in some standards specifications.

In the GDMO MANAGED OBJECT CLASS template, the DERIVED FROM clause is used to specify direct base classes. All GDMO object classes are a direct or indirect derived class of the managed object class, *top*, defined by the ISO DMI. An indirect base class would be a base-class of a base-class. An example showing the structure of the Object Class Inheritance Tree is shown in Figure 2-6:

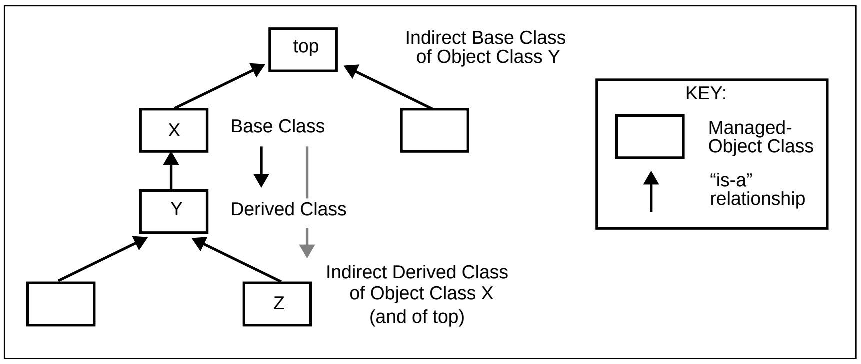

top
Indirect Base Class of Object Class Y
X
Base Class
Derived Class
Y
Z
Indirect Derived Class of Object Class X (and of top)
KEY:
Managed-Object Class
"is-a" relationship

Figure 2-6 Object Class Inheritance Tree

The Managed Object Class Inheritance hierarchy for the ISO DMI is shown in Figure 2-7:

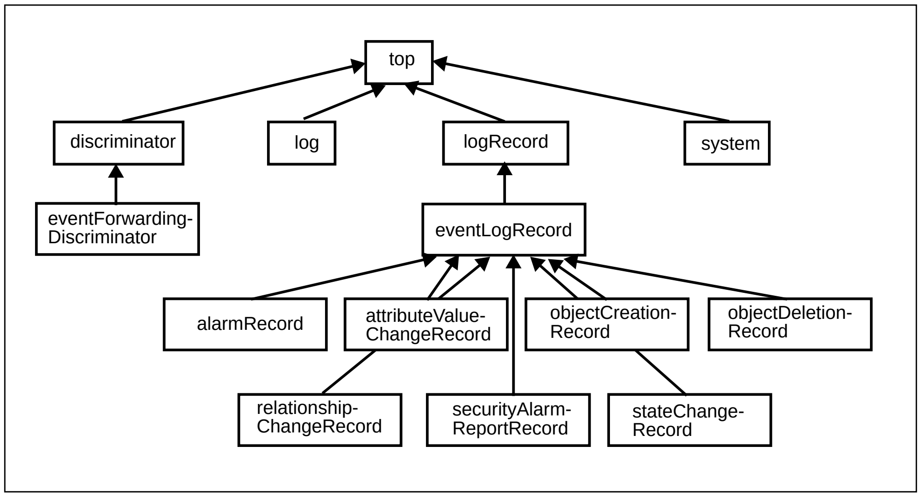


Figure 2-7 ISO DMI Object Class Inheritance Tree

2.2.5 Object Identifier (Registration) Tree

The *Object Identifier Tree* contains nodes labeled using nonnegative integer values and a text label. Refer to Section 2.1.2.5 for more details and an example Object Identifier Tree.

In review, the top of this tree is called *root*. There are three labeled nodes under root which are administered by the ISO and CCITT standards organizations. There are in turn sub-nodes under the three ISO and CCITT nodes some of which are administered by other organizations. A unique label can be formed for any node in the tree by concatenating the name of each node in the tree, starting at root, and proceeding down the tree to a node in the tree. The label formed using this method is an OBJECT IDENTIFIER.

Note – Do not confuse the root of the Object Identifier Tree with the root of the Management Information Tree (MIT) described in Section 2.2.6.2. They are separate entities, although both have the same name.

2.2.6 Management Information Tree and Containment

To understand Solstice EM and develop Solstice EM solutions, you need to understand the concepts of containment and the Management Information Tree. The concepts are simple, but are used in so many ways and underlie so many things that the concepts can seem confusing. In addition to the information presented here, CCITT X.720 ISO/IEC 10165-1 contains a section titled “Principles of containment and naming” which provides a good overview of containment and the MIT. It is strongly recommended that you also review the information in that section of the specification.

In addition to the concepts of containment the Management Information Tree (MIT), this section will also cover the following related concepts:

- Managed object instance naming
  - Attribute Value Assertion (AVA)
  - Relative Distinguished Name (RDN)
  - Distinguished Name (DN) or Fully Distinguished Name (FDN)
  - Local Distinguished Names (LDN)
  - Human readable name forms
  - Encoded name form
- Scoping and multiple replies
- Filtering

2.2.6.1 Containment

The following description of containment is from CCITT X.721 ISO/IEC 10165-1:

> A managed object of one class can contain other managed objects of the same or different class. This relationship is called *containment*. This containment relationship is a relationship between managed object instances, not classes. A managed object is contained within *one and only one* containing managed object. Containing managed objects may themselves be contained in another managed object.

The containment relationship is capable of reflecting a real-world (or physical) hierarchy of relationships between resources. The following are two examples of real-world containment hierarchies:

- A computer system contains a disk drive, the disk drive contains a file system, the file system contains one or more directories, a directory contains one or more files, and so on.
- A network contains a router and the router contains three interface cards.

The containment relationship can also reflect behavioral relationships between components. For instance, if the computer system in the previous example is down for any reason, it would not be possible to access the disk drive contained in the computer system or any of the files contained on the drive.

The preceding examples reflect real-world containment relationships and provide a good means to represent the concept of containment. For ISO management purposes however, *containment is not restricted to reflecting physical containment of one managed resource in another.*

2.2.6.2 *Management Information Tree (MIT)*

The containment relationship is used for naming managed object instances. Object instances that are named in the context of another object are subordinate object instances of that object. The object instance that establishes the naming context is the superior object instance. Superior object instance and subordinate object instance are synonymous with superior object and subordinate object, respectively.

Note – Do not confuse the terms superior object and subordinate object with the terms superior object class (base class or “superclass”) and subordinate object class (derived class or “subclass”). The former refers to object instances, the latter refers to object classes.

A subordinate object instance is named by a combination of the following:

- The name of its superior object
- The name (identifier) and value of one of the subordinate object’s attributes. This attribute is known as the naming attribute. The naming attribute and its value must produce a name that is unambiguous relative to the superior object.

A superior object instance is named in the same way relative to its superior, producing a hierarchy of named managed object instances. This hierarchy is the *Management Information Tree (MIT)*, sometimes referred to as the naming tree. An example MIT is shown in Figure 2-8. The figure also shows an object instance name (a distinguished name). Object instance names and naming is described in the next section.

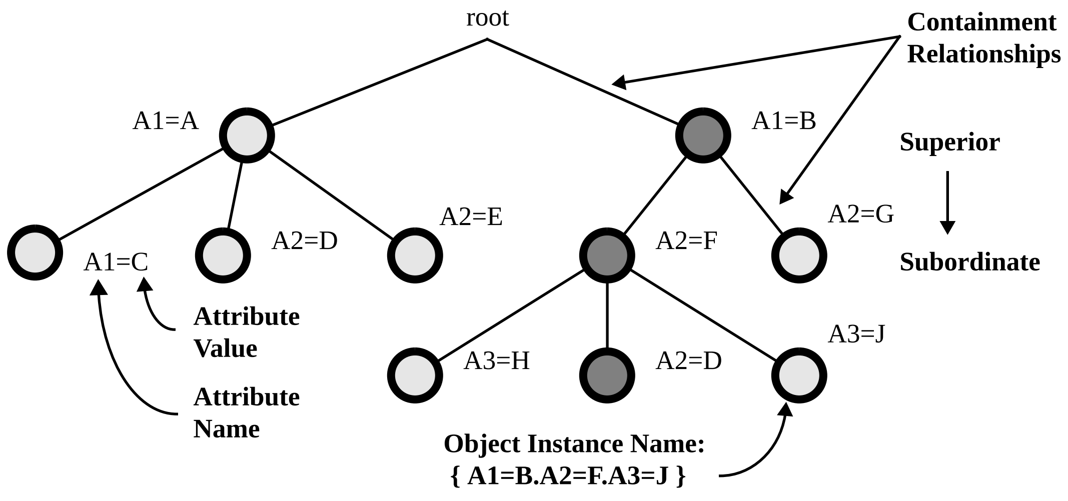


Figure 2-8 Management Information Tree (MIT)

The top of the MIT is *root*. Root is never instantiated as a managed object (in other words, root is a null object representing a conceptual starting point for the MIT). The name of a managed object subordinate to root is the naming attribute and value of the subordinate object.

Note – Do not confuse the MIT root, with the root in the Object Identifier tree. Although neither root has a physical representation, the two roots are separate (conceptual) entities.

The concept of containment and the MIT imply the following for managed objects:

- A managed object instance can only exist if its superior managed object instance exists.
- Every managed object instance has a name, which is derived from where an object instance resides in the MIT.

2.2.6.3 *Managed Object Instance Naming*

As stated previously, a managed object is named using the name of an attribute of the object and the value of that attribute. A name binding is used to specify the attribute that is used for naming (See Section 2.2.3.3, “Name Binding Template”). The attribute identifier specified in the name binding is the attribute used for the naming attribute in the MIT. The name binding also specifies a superior object *class* and a subordinate object *class*. The superior and subordinate object classes in the name binding dictate which object classes have instances that can contain instances of another object class. (The containment relationship is object class based not object instance based. That means an instance of class X can be created as a subordinate to an instance of class Y only if a name binding exists that specifies that class X can be a subordinate class to class Y (superior class). If that name binding exists, then when the instance of class X is created it will be named using the attribute identifier specified in the WITH ATTRIBUTE clause of the name binding and that attribute’s value.)

The following name binding is provided as an example:

```
exampleNameBinding1 NAME BINDING
    SUBORDINATE OBJECT CLASS  exampleClass ;
    NAMED BY SUPERIOR OBJECT CLASSsystem ;
    WITH ATTRIBUTE            exampleID ;
REGISTERED AS { 0 1 2 3 4 1 } ;
```

For the above example, we assume that a managed object class exampleClass has been previously defined and that one of the properties specified for the class is an attribute named objectName. We will further assume that the attribute exampleID is registered as 0.1.2.3.4.5.6 and is defined as being an ASN.1 INTEGER.

The example name binding dictates that instances of the managed object class exampleClass can be created under system (system is defined in the ISO DMI) using exampleID as the naming attribute. Furthermore, if this is the only name binding that exists for exampleClass (i.e., no other naming possibilities exist), it would mean that instances of exampleClass can *only* be created under system.

If additional name bindings exist for a class for a subordinate class, they can specify other superior classes and different naming attributes. An example of a second name binding is as follows:

```
exampleNameBinding2 NAME BINDING
   SUBORDINATE OBJECT CLASS  exampleClass ;
   NAMED BY SUPERIOR OBJECT CLASSroot ;
   WITH ATTRIBUTE            exampleID ;
REGISTERED AS { 0 1 2 3 4 2 } ;
```

This second name binding specifies that instances of exampleClass can be created under root, and those instances will be named using exampleID. The naming attribute is specified in a name binding is left to the name binding designer, subject to the need for an unambiguous identifier relative to the superior object class. A naming attribute other than exampleID could just as easily been used in this second name binding example.

Attribute Value Assertions (AVA)

The combination of an attribute identifier and its value is known as an attribute value assertion (AVA)[1]. An attribute identifier is an OBJECT IDENTIFIER (OID) that has been registered for a particular attribute. Part of the information registered for an attribute also specifies what type of data the attribute contains (such as INTEGER, REAL, or other datatype). Taking this a step further and looking at the encoding rules for an AVA we will see that an AVA is defined as a SEQUENCE containing an attribute identifier (an OID) followed by whatever type of data was specified for that attribute. The example name bindings in the previous section specified a naming attribute called exampleID (which was type INTEGER). If we specify a value for this attribute, we can produce the following human-readable form of the AVA:

```
exampleID=10
```

1. To be precise, an attribute value assertion is a statement, which may be true or false, concerning the value of an attribute. If an object instance is created and it has an INTEGER attribute named exampleID that has the value 10, we are basically asserting the exampleID=10 is indeed a true statement for the object instance created.

that would be encoded as shown in Figure 2-9:

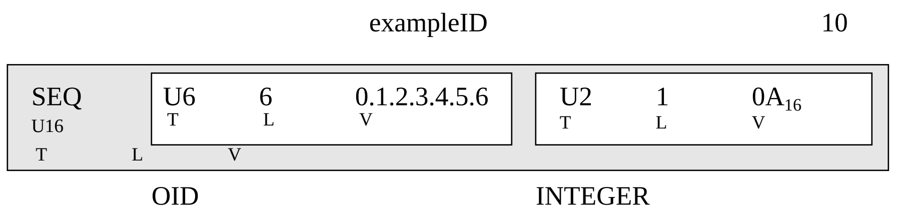


Figure 2-9 Attribute Value Assertion ASN.1 BER Encoding

An AVA is only part of the construct that forms the name of a managed object instance. An AVA exists for every attribute of a managed object instance. The construct that forms the managed object name is the relative distinguished name, which is covered in the following section. The syntax for an attribute value assertion is imported from the ISO directory services and is as follows (the syntax is also be listed in Annex G of CCITT X.721 ISO / IEC 10165-2):

```
AttributeValueAssertion ::=
    SEQUENCE { AttributeType, AttributeValue}
AttributeType ::= OBJECT IDENTIFIER
AttributeValue ::= ANY
```

Relative Distinguished Name (RDN)

A relative distinguished name (RDN) identifies a managed object instance. The RDN when joined with the name of the superior object instance provides a unique name for a managed object within the MIT. The RDN by itself must be unique within the context of its superior object instance. The human readable form of the RDN for the example object used previously is:

```
    exampleID=10
```

which is the same as the AVA in the preceding section. The encoding for the RDN and the syntax definition differ from an AVA which can be seen in Figure 2-10. The encoding specifies a SET OF Attribute Value Assertions.

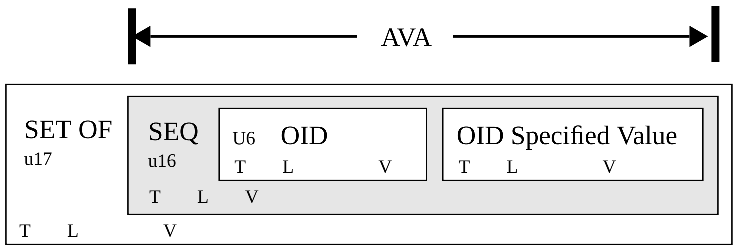


Figure 2-10 Relative Distinguished Name (RDN) ASN.1 BER

However, Relative Distinguished Names only use a single AVA (which is the reason the human readable forms look alike). This encoding is by convention and is not an actual restriction in the standards. The single AVA approach simplifies the processing and formation of RDNs while still allowing the RDN to match the format of an X.500 directory services name. The syntax for a relative distinguished name is imported from the ISO directory services and is as follows (the syntax is also be listed in Annex G of CCITT X.721 ISO/IEC 10165-2):

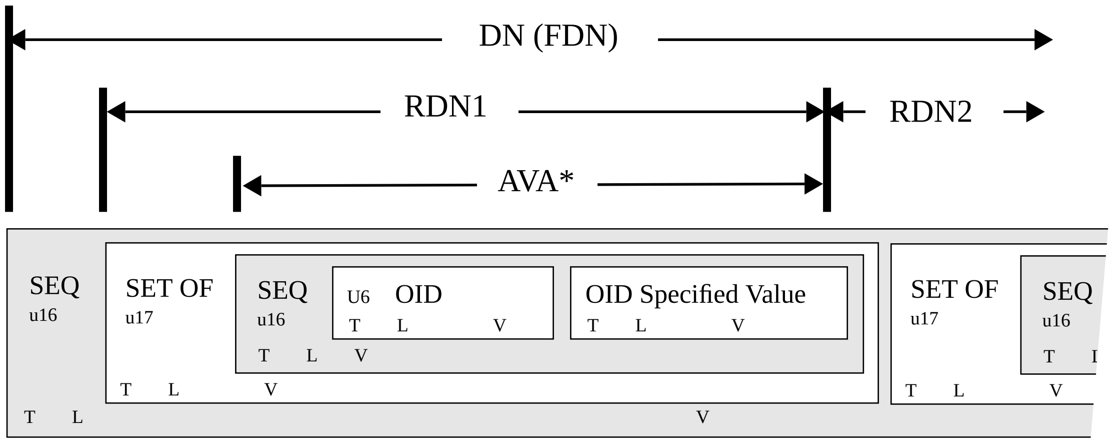
```
RelativeDistinguishedName ::= SET OF AttributeValueAssertion
```

Distinguished Name (DN) or Fully Distinguished Name (FDN)

The *distinguished name(DN)* of a managed object instance is formed from the concatenation of the RDNs in the MIT from root to the managed object instance. The distinguished name is frequently referred to as the *fully distinguished name (FDN)*. An example of a distinguished name would be:

```
systemId=acid.exampleID=10
```

In the above example, two RDNs are concatenated to form the distinguished name shown. Distinguished names are encoded as shown in Figure 2-11:



*Only a single AVA is used for an RDN (this is by convention, not specified in the standards). The above format is identical to X.500 names, although X.500 names may use multiple AVAs for a single RDN.

Figure 2-11 Distinguished Name ASN.1 BER Encoding

The syntax for a distinguished name is imported from the ISO directory services and is as follows (the syntax is also be listed in Annex G of CCITT X.721 ISO / IEC 10165-2):

```
DistinguishedName ::= RDNSequence
RDNSequence ::= SEQUENCE OF RelativeDistinguishedName
```

Distinguished names are used to name managed object instances. The syntax for a managed object instance is as follows:

```
ObjectInstance ::= CHOICE {
    distinguishedName[2]IMPLICIT DistinguishedName,
    nonSpecificForm[3]IMPLICIT OCTET STRING,
    localDistinguishedName[4]IMPLICIT RDNSequence
}
```

Distinguished names are used for the first form (context-specific tag 2). The second form, nonSpecificForm (context-specific tag 3) is not used an can be ignored. The third form, local distinguished name (context-specific tag 4) is described in the following section.

Human Readable Distinguished Name Forms

Solstice EM frequently uses either a “slash” representation or “curly braces” representation for distinguished names (object instance names) and relative distinguished names.

- The curly braces representation is based on the ASN.1 syntax definition for a distinguished name. The curly braces form for `systemId=acid.example=10` is:

```
    { { { systemID, “acid” } }, { { exampleID, 10 } } }
```

- The slash representation is a simpler form defined for use by Solstice EM and is as follows:

```
    /systemId=name:”acid”/exampleID=10
```

EM also provides a number of C++ classes and functions for encoding distinguished names and supporting constructs.

Local Distinguished Name (LDN)

A local distinguished name is an object instance name that is relative to a node in the MIT other than root. Fully distinguished names are always relative to root, a local distinguished name is relative to another node. This other node is sometimes referred to as the local root. The following slash representation is a fully distinguished name for a managed object:

```
/systemId=name:”host1”/systemId=name:”hub1”/exampleID=10
```

The following slash form is a local distinguished name for the same managed object:

```
systemId=name:”hub1”/exampleID=10
```

The local distinguished name is relative to `systemId=name:”hub1”` (i.e, `systemId=name:”hub1”` is the local root). Local names are provided due to name handling issues for some agents and managers. In particular some agents cannot understand or work with names that have the entire path from root to the agent as part of the name. Local distinguished names are provided as a mechanism that allows an agent (or manager) to not have to worry about understanding names that include attributes and values that it knows nothing about or cannot interpret.

A local distinguished name is used to identify a managed object instance. As listed in the section describing FDNs, the syntax for a managed object instance identified by a local distinguished name is the third choice in the following definition:

```
ObjectInstance ::= CHOICE {
    distinguishedName[2]IMPLICIT DistinguishedName,
    nonSpecificForm[3]IMPLICIT OCTET STRING,
    localDistinguishedName[4]IMPLICIT RDNSequence
}
```

When a local distinguished name is encoded, it is given a context-specific tag of 4, which identifies it as a local distinguished name.

Distinguished Names and the MIT

Figure 2-12 shows an example MIT and the naming constructs described in the preceding sections:

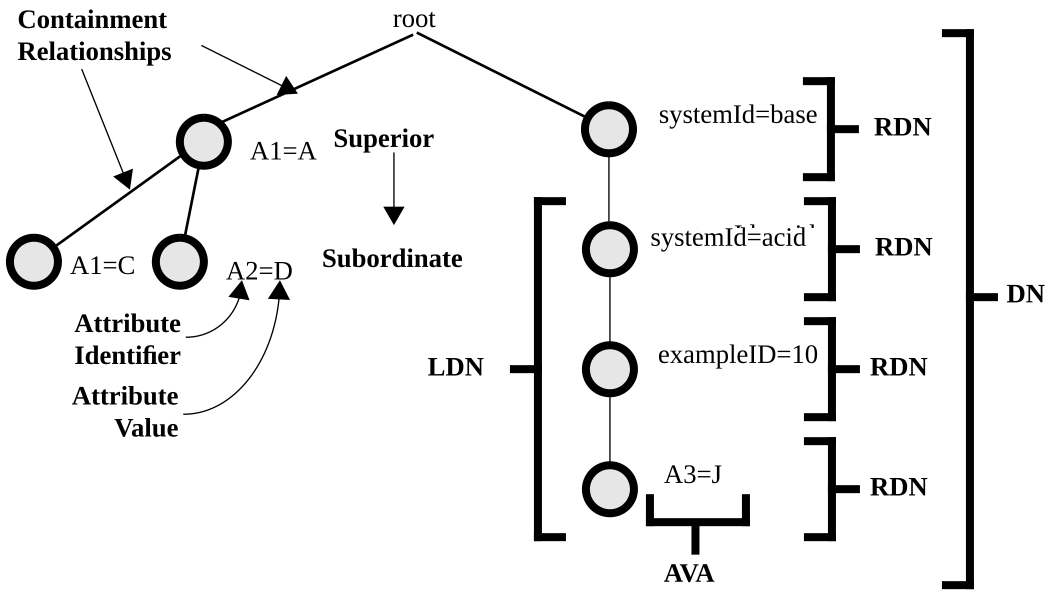


Figure 2-12 MIT Components

2.2.6.4 *Scoping, Filtering, and Multiple Replies*

A manager can specify scoping and filtering in a CMIP request message. Scoping is used to identify a sub-tree of managed object instances in the MIT. Once a sub-tree has been identified the filter (or test) can be applied to the identified instances. The instances that pass the test successfully are requested to perform the management operation specified in the CMIP request message. Finally, each managed object that performed the management operation returns any required response.

Scoping

A scoping operation always starts from a specified object instance in the MIT known as the *base object instance*. The name of a base object instance is specified in CMIP request messages that support scoping (i.e., m-Get, m-Set, m-Delete, m-Action). Four different types of scoping can be specified. Each type of scoping identifies a set of managed object instances (subordinate to and including the base object instance) in the MIT against which a filter will be applied. The different types of scope are as follows:

- The base object only. This is the default scope which selects only the base object instance specified in the CMIP request. The base object is defined to be level 0.
- The n^{th} level subordinates of the base object. This selects object instances subordinate to the base object that are *n* levels below the base object. Object instances directly subordinate to the base object are at level 1. Object instances below those objects are at level 2, etc.
- The base object and all subordinates down to the n^{th} level. This selects a subtree that starts with the base object and includes all objects instances subordinate to the base object down to and including level *n*.
- The base object and all its subordinates. This selects the base object and all object instances subordinate to it.

Figure 2-13 shows examples of the four types of scoping. Each type of scope shown in the example starts from the same base object.

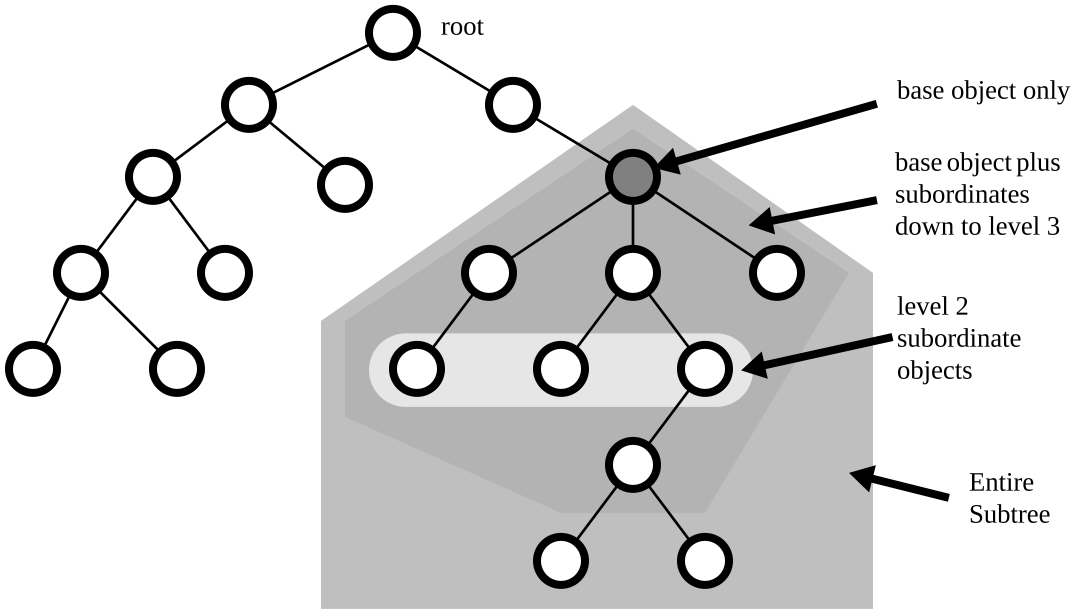

root
base object only
base object plus subordinates down to level 3
level 2 subordinate objects
Entire Subtree

Figure 2-13 Subtrees Selected by Scope

Filtering

A filter is used along with scoping to select a set of managed objects which are requested to perform a management operation. Scoping identifies a subtree of objects in the MIT. A filter specifies a test that is to be applied to the managed objects selected by the scoping operation. If the test is successful, the request is performed by the managed object. If the test fails, the request is not performed.

A filter can test for a variety of things, such as:

- Is the managed object instance a member of a particular object class?
- Does the managed object instance contain a particular attribute?
- Does the managed object class contain a particular attribute and does that attribute have a particular value?

Multiple Replies

Several managed objects are usually selected when scoping and filtering are specified in a CMIP request. Each managed object that carries out the request will return a response to the manager that issued the scoped and filtered request. The response sent back by the managed object will be: an error response; a response containing information specified by the request; or a final response. Each response which contains information specified by the request is called a linked reply. The linked reply message is used regardless if the original request was an m-Get, m-Set, m-Action or m-Delete request. The final response is an m-Get response, m-Set response, m-Action, or m-Delete response that corresponds to the types of request originally issued by the manager.

A number of different error responses are also possible -- refer to CCITT X.711 ISO/IEC 9596-1 for more information about the types of error responses.

2.2.7 ASN.1 and CMIP

ASN.1 is used to encode CMIP requests, responses, and notifications. CMIP does not place any restriction on the ASN.1 datatypes that can be used for management operations (i.e., see the GDMO and CMIP ANY DEFINED BY clause). CMIP also allows the use of constructed datatypes.

2.3 Internet Network Management Concepts

This section provides a brief introduction to Internet network management. Key concepts introduced in this section include:

- SNMPv1 Protocol
- SNMPv2 Protocol
- Concise MIB
- SNMPv2 MIB
- ISO and Internet Management Coexistence

SNMP relies on a model based on a model using managers and agents. Concise MIB defines management information for use with the SNMPv1 protocol. SNMPv2 MIB defines management information for use with the SNMPv2 protocol. SNMPv1 is used to exchange management information between an SNMPv1 manager and an SNMPv1 agent. SNMPv2 is used to exchange management information between an SNMPv2 manager and an SNMPv2 agent and also has a provision for exchanging management information between two SNMPv2 managers.

2.3.1 SNMPv1 Protocol

An SNMPv1 message consists of a version number field, a community name field, and a data field. The data field can contain any one of five types of protocol data units (or PDUs). The first four PDUs in the following list have the same PDU format. The Trap-PDU has a different format form the other PDUs. The five SNMPv1 PDUs are as follows:

- GetRequest-PDU

  The SNMP get-request is used by a manager to request attribute information from an agent. The agent will typically respond with a get-response containing attribute name-value pairs, called *variable bindings*. A variable binding is similar in nature to an ISO attribute value assertion (AVA), which is also an attribute name-value pair.

- GetNextRequest-PDU

  The SNMP get-next request is used by a manager to request sequential attribute information. It is typically used by the manager to request that an agent read through the rows in a table one row at a time.

- SetRequest-PDU

  The SNMP set-request is used by a manager to request that an agent set selected attributes to values specified in the request.

- GetResponse-PDU

  The SNMP get-response is used by an agent to return the result of a get-request, a get-next request, or a set-request to a manager. The response for each of these requests uses the format of the get-response. SNMP version 2 simply refers to this operation as a response.

- Trap-PDU

  An SNMP trap is used by an agent to send unsolicited information to a manager.

Refer to the following RFC for more information about SNMP version 1 protocol:

RFC 1157, A Simple Network Management Protocol (SNMP Version 1).

2.3.2 SNMPv2 Protocol

An SNMPv2 message consists of a destination party field and a data field. The data field can contain either encrypted or clear text. The data contains authentication information, the destination party (repeated), the source party, a context identifier, and a PDU. The PDU can be one of seven types. Five of the types are the same as the SNMPv1 PDUs and the trap PDU has been modified to also match the other SNMPv2 PDUs. The seven SNMPv2 PDUs are as follows:

- GetRequest-PDU

  The SNMPv2 get-request performs the same service as the SNMPv1 get-request.

- GetNextRequest-PDU

  The SNMPv2 get-next-request performs the same service as the SNMPv1 get-next-request.

- SetRequest-PDU

  The SNMPv2 set-request performs the same service as the SNMPv1 set-request.

- Response-PDU

  This PDU performs the same services as the SNMPv1 GetResponse-PDU. In addition it is also used as a response to SNMP inform requests.

- snmpV2Trap-PDU

  An SNMPv2 trap is used by an agent to send unsolicited information to a manager. The SNMPv2 trap PDU has the same format as the preceding SNMPv2 PDUs.

- InformRequest-PDU

  The inform request is used to send information from one SNMPv2 manager to another SNMPv2 manager.

The InformRequest-PDU has the same format as the preceding SNMPv2 PDUs.

- GetBulkRequest-PDU

  The SNMPv2 get-bulk request is used by a manager to request large blocks of information from an agent.

Refer to the following RFC for more information about SNMP version 2 protocol:

RFC 1448, Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2)

2.3.3 Concise MIB

Concise MIB provides a set of macros that are used to define management information for SNMPv1. A MIB defined using the Concise MIB format allows SNMPv1 agents to describe what information and services they provide in a format that can be understood and used by an SNMPv1 manager. Concise MIB is used to define the following MIB information:

- Attributes or types of data supported by the managed object class
- Operations or actions supported by the managed object class
- Behavior exhibited by the managed object

The Concise MIB format does not specify SNMP trap formats. Instead a Concise Trap format has been defined to describe traps. SNMP traps are the unsolicited information an SNMPv1 agent can generate and send to a manager (i.e., a notifications).

Refer to the following RFCs for more information about SNMPv1 MIBs and traps:

- RFC1212, Concise MIB Definitions
- RFC1213, Management Information Base for Network Management of TCP/IP-based Internets: MIBII
- RFC1215, A Convention for Defining Traps for use with SNMP.

2.3.4 SNMPv2 MIB

SNMPv2 MIB provides a set of macros that are used to define management information for SNMPv2. A MIB defined using the SNMPv2 format allows SNMPv2 agents to describe what information and services they provide in a format that can be understood and used by an SNMPv2 manager. SNMPv2 MIB is used to define the following MIB information:

- Attributes or types of data supported by the managed object class
- Operations or actions supported by the managed object class
- Behavior exhibited by the managed object
- Traps or unsolicited information an SNMPv2 agent can generate and send to a manager (i.e., a notifications)

Refer to the following RFC for more information about SNMPv2 MIBs:

RFC1442, SNMPv2 Structure Of Management Information (SMI) for version 2 of the Simple Network Management Protocol.

2.3.5 ASN.1 and SNMP

SNMPv1 and SNMPv2 support a subset of the ASN.1 datatypes. SNMPv1 supports the following ASN.1 datatypes:

- INTEGER
- OCTET STRING
- NULL
- OBJECT IDENTIFIER
- SEQUENCE, SEQUENCE-OF

SNMPv2 supports the same ASN.1 datatypes as SNMPv1 and also adds support for the following datatype:

- BIT STRING

Both SNMPv1 and SNMPv2 allow the use of constructed types.

2.3.6 ISO and Internet Management Coexistence (IIMC)

The Network Management Forum and X/Open have defined a number of specifications that describe how ISO (CMIP and GDMO) network management and Internet (SNMP and Concise MIB/SNMPv2 MIB) network management can coexist. Among other things, these specifications define:

- How to map CMIP requests into SNMP requests
- How to map SNMP responses into CMIP responses
- How to map SNMP into CMIP event reports
- How to map SNMPv1 and SNMPv2 MIBs into GDMO

These mappings allow SNMP version 1 and version 2 agents to be represented as GDMO managed objects.

EM Programming Concepts 3

Solstice EM consists of a Management Information Server (MIS) that serves as a repository of information about the network, and a set of applications (or services) that use the MIS. Applications running on various workstations communicate with a Solstice EM MIS to get information about the network. The MIS plays the role of server to the applications and services that are its clients. Management functions are distributed between the MIS and its clients.

The Solstice EM development environment provides a set of Application Programming Interfaces (APIs), including the high-level and low-level Portable Management Interface (PMI), and other tools that enable a software developer to build comprehensive SNMP- and CMIP-based network management solutions using Solstice EM.

3.1 Terminology and Concepts

Solstice EM uses several industry recognized terms and concepts and also introduces some EM-specific terms. Some of the industry recognized terms have multiple definitions.This section provides specific definitions for the following terms:

- Application
- Client-Server
- Client Subcomponent
- Server Subcomponent
- Object Oriented
- Remote Object
- Local Object

The terms *application* or *PMI application* describe a user-developed component (client application) that is linked to and uses the PMI. A PMI application is typically part of a Solstice EM solution, but could also be a complete Solstice EM solution. The PMI is described in detail in the *Solstice Enterprise Manager API Syntax Manual*.

The term *client-server* refers to a distributed architecture, in which a client application accesses information or services provided by a separate server application. The client application can reside on the same workstation as the server application, or it can be *distributed* and reside on a different workstation, accessing the server using a network. The Solstice EM MIS is a server application. EM applications, such as the Viewer or the Request Designer, are client applications.

A *client subcomponent* is any user-developed or user-configured subcomponent that is contained in a PMI application. A *server subcomponent* is any user-developed or user-configured subcomponent that is contained in the MIS.

The term *object-oriented* refers to a software model that uses objects to define behavior and data. Object behaviors are sometimes referred to as *operations* or *actions*. Data are commonly referred to as attributes. An object class is a definition — it defines what attributes and behaviors are supported by a particular type of object. Some of the behaviors defined for a class are typically used to read or modify the attributes supported by the object class. An object instance is a realization (or instantiation) of the object class definition. An object instance will have a value for each attribute defined by the object class. The behaviors defined for the class can be used to access the attribute values. Solstice EM supports *object-oriented* concepts on three levels:

1. Solstice EM supports the ISO management model and GDMO, both of which are object oriented.
2. Solstice EM uses an internal framework of objects classes and instances to support ISO, Internet, SunNet Manager, and other types of management models.
3. Solstice EM has been implemented using C++, which is an object oriented programming language.

Within the context of Solstice EM, a *remote object* (or *remote object instance*) is any managed object instance that is external to the MIS. The MIS maintains a reference to a remote object instance via the MIT, but the object instance is external to the MIS. An example of a remote object is a managed object instance that is part of an agent that represents a device in your network.

A *local object* (or *local object instance*) is any managed object instance that is part of the MIS. The operations supported by the local object are carried out by the MIS process and the attribute values for the local object are stored or maintained in the MIS or its persistent store. The MIS maintains a reference to the local object instance in the MIT. Local objects will be described further when object class definitions with repository behavior are presented.

3.2 MIS Subcomponents

A server subcomponent is a user-configured or user-developed subcomponent that is contained in the Solstice EM MIS. That means that it is part of the same UNIX process as the Solstice EM MIS. A server subcomponent or information from a server subcomponent is generally accessible to any PMI application provided the PMI application has proper access permissions. The following Solstice EM subcomponents are server subcomponents:

- Requests (Nerve Center Engine)
- Event Forwarding Discriminators (EFDs)[1]
- Log and Log Record Objects
- Object Class Definitions and Name Bindings

- Object Class Definitions With Repository Behavior

Figure 3-1 shows the distribution of the MIS subcomponents within the Solstice EM client-server architecture.

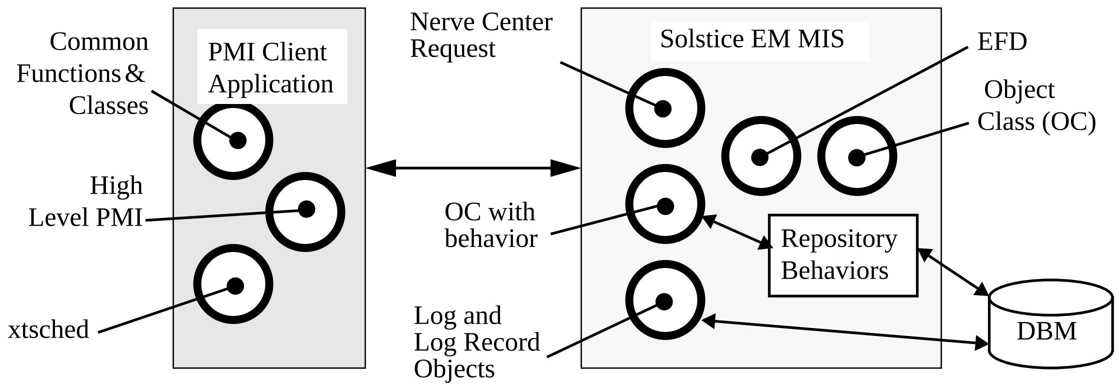


Figure 3-1 Solstice EM Development and Configuration

3.2.1 *Requests*

Requests are a general purpose mechanism for detecting conditions in a network and taking action in response. The conditions in a network are monitored by a Request using combination of polling and notifications. When a particular condition is detected a Request can perform one or more actions in response to the condition. Each Request is defined by a *Request template.* A Request template is a user-configured state diagram that consists of one or more states, network conditions to test while in a state, transition paths between states, and actions that can be taken when a transition between states occurs. Requests are described in greater detail in the *Solstice Enterprise Manager Reference Manual*. Requests are the most powerful user-configurable subcomponent available to a solution developer — you do not need to write software in order to use a Request.

1. EFDs can also reside in an agent or in another Solstice EM MIS -- in which case they would be a remote object.

Note – Do not confuse the Request Templates and Requests with management protocol requests, such as an SNMP get request or a CMIP m-Get or m-Set request. The Requests described here are used to configure the Solstice EM Nerve Center to intelligently manage objects in a network.

What To Use:

- Request Designer application
- View results in the Viewer, the Alarm Manager, and the Log Manager

Note – Requests can also log information to the pre-configured (default) log provided with the MIS.

When To Use:

Requests are a general purpose mechanism for detecting conditions in a network and taking action in response. There is a variety of conditions in which it makes sense to use Requests. The following list provides only a few examples:

- Managing devices in your network

  This is the most common use for Requests and can range in complexity from simply testing a simple up-down state for a device to testing multiple states for a device based on availability, performance, and other factors. A single Request can be used to monitor multiple devices, including different classes of devices. You can also create specialized Requests which monitor servers, backbone devices, or other mission critical resources in your network.

- Managing a variety of other types of physical or logical devices, processes, or services in your network

  Anything that can be represented by a managed object class definition can potentially be managed by a request.

- Managing network performance

  Again this can range in complexity from simple performance to highly specialized and tuned performance monitoring. A request could be configured to raise an alarm when performance or network loads hit specified levels.

- Managing security violations

  An example of security management would be counting SNMP authorization failure traps, logging such types of traps, or raising an alarm when a specified number of these traps occur on a single device.

3.2.2 Event Forwarding Discriminators (EFDs)

Event Forwarding Discriminators (EFDs) provide a powerful ISO standards-based mechanism for disseminating event information. EFDs are a managed object class defined in the ISO DMI (CCITT X.721 ISO / IEC 10165-2). EFDs include a *discriminator construct* attribute (frequently referred simply as a *discriminator*) and a *destination* attribute, as well as other attributes. The discriminator construct specifies a filter which is used to test event information. Event information that passes the test is sent to each destination specified by the EFD's destination attribute.

The destination attribute contains a list of destinations to which the event information (after it has passed the filter) is sent. The syntax for the destination attribute is as follows:

```
Destination ::= CHOICE {
     singleAE-title,
     multipleSET-OF AE-title
}
```

The syntax for an application entity title (AE-title) is as follows:

```
AE-title ::= AP-title
```

The syntax for an application title (AP-title) is as follows:

```
AP-title ::= CHOICE
{
     distinguishedName DistinguishedName,
     objectIdentifier OBJECT IDENTIFIER
}
```

Solstice EM supports both the distinguished name form and the OID form for AP-Titles. However, *only the OID form will work with EFDs.*

An example of a single destination address using the OID AP-title form of addressing is:

```
{ 1 2 3 4 }
```

An example of multiple destination addresses using the OID AP-title form of addressing is:

```
{ { 1 2 3 4}, { 1 2 3 5 } }
```

The syntax for a discriminator construct is as follows:

```
DiscriminatorConstruct ::= CMISFilter
```

This is the same syntax as used for filters in CMIP messages. The default filter value for an EFD discriminator is:

```
and : {}
```

This default value will cause all event information to match the filter and be sent as event reports to all destinations specified in the destination attribute. It should be noted that the EFD will only forward event information if the EFD is also enabled (for example, the value of the administrativeState attribute in the EFD is set to enabled).

What To Use:

- CMIP Configuration Utility: em_oct -cmip
- Object Creation Utility: em_objcreate
- EM_SERVER Environment variable

Note – EFDs can also be created using CMIP m-Create requests or using the Low Level PMI M-Create functions. For more information on using create requests and creating EFDs refer to CCITT X.711 ISO / IEC 9596-1 and CCITT X.735 ISO / IEC 10165-5.

When To Use:

By default, all event information received by the Solstice EM MIS event management subcomponent can be made available to Nerve Center. Nerve Center (using Requests) is capable of performing additional processing and correlation of event information over and above what can be done using an EFD. In most cases a Request should be developed and used instead of an EFD, but there are a number of situations in which an EFD is better suited or necessary:

- An EFD must be used to pass raw ISO-style event information from one Solstice EM MIS to another Solstice EM MIS. Either or both of the MISs can subsequently use their respective Nerve Centers for additional processing of the event information. Currently, Nerve Center does not provide a mechanism to send raw ISO-style[1] event reports to another Solstice EM MIS, or to conditionally pass on raw event information compiled from a number of different event reports, which is the reason an EFD must be used.

  When an EFD is used to pass event information from one Solstice EM MIS to another, the MIS that sends the event is acting in the agent role (or acting in both agent and manager roles). The MIS receiving the trap is acting in the manager role (or in both agent and manager roles).

- An EFD should also be used to pass event information from the Solstice EM to a CMIP platform other than Solstice EM MIS. Alternatively, information could also be passed to another CMIP platform using a SendAction function from a Nerve Center Request (which translates to a CMIP m-Action request), but this will work only if the other platform contains an object instance that supports the specified action.

1. Nerve Center does provide a mechanism to send trap data to another Solstice EM MIS or other SNMP management platform.

- EFDs must also be used in CMIP agents in order to pass event information from the agent to the Solstice EM MIS. A CMIP m-Create request can be issued by the Solstice EM MIS in order to create an EFD in an agent. Some agents may contain pre-configured or hard-wired EFDs making the m-Create operation unnecessary.

  CCITT X.734 ISO / IEC 10164-5.

  CCITT X.735 ISO / IEC 10164-6.

  CCITT X.711 ISO / IEC 9596-1.

3.2.3 Log and Log Record Objects

Event logs (frequently referred to as *logs*) maintain a record of information provided in notifications. Examples include tracking changes to configuration information for one or more devices, recording security notifications received from different devices in the network, and recording alarm notifications received. Each log contains a discriminator construct attribute that selects which event reports to record in the log. The syntax of the discriminator construct is the same as the syntax of the event forwarding discriminators (described in Section 3.2.2). An MIS can contain multiple logs[1], each containing a different discriminator, in order to record different types of information in separate logs. The Log Manager is a client application for creating logs in the Solstice EM MIS. Each log is a local managed object.[2]

A log contains *event log records* (sometimes referred to as *log records* or *log record objects*). Event log records are generated from notifications received by the MIS or generated by Nerve Center. A log record is created and stored in a log if the notification matches the discriminator defined for the log. Each log record is a local managed object.[3]

1. The Solstice EM Alarm Manager can work with information from only one log at a time, although the designated log can be changed while the MIS is running.

2. Logs can also reside in an agent, in another Solstice EM MIS, or in another ISO management platform — in which case, the logs are a remote objects.

3. Log records can also reside in an agent, in another Solstice EM MIS, or in another ISO management platform — in which case the log records are a remote objects.

Information in an event log record typically mirrors information in a notification. If you look in the ISO DMI (X.721 ISO / IEC 10165-2), you will find a one-to-one correspondence between notification types and event log record objects. For example, the DMI defines an attribute ValueChange notification type, which a managed object can emit, and defines a matching attribute ValueChangeRecord managed object class which stores information from an attribute ValueChange notification. If you add a new notification type to the MIS (that is, you parse in a GDMO definition that defines a new notification type), you should also add a new event log record managed object class that corresponds to the new notification type. Solstice EM provides a utility that can automatically define event log record managed object classes based on a notification definition.

The Solstice EM MIS is pre-configured with a single AlarmLog (i.e. log object instance) that matches and logs a number of types of notifications. The syntax for the CMIS filter used by the default log is included in the *Solstice Enterprise Manager Reference Manual*. The types of notifications logged by the default (pre-configured) log object are:

- nerveCenterAlarm (generated by the alarm() function)
- communicationsAlarm
- environmentalAlarm
- equipmentAlarm
- processingErrorAlarm
- qualityofServiceAlarm
- internetAlarm (SNMP traps)

The following types of notifications defined by the ISO DMI are not logged by the default log object:

- attributeValueChange
- integrityViolation
- objectCreation
- objectDeletion
- operationalViolation
- physicalViolation

- relationshipChange
- securityServiceOrMechanismViolation
- stateChange
- timeDomainViolation

What To Use:

- Log Manager application: em_logmgr
- Log Creation window
- CMIS filters
- Request Condition Language alarm() function
- GDMO definition for new event log record
- Object editor to modify the eventZObject Class object that defines the mappings between notifications and event log records.

  For further information see the README file for Scenario One, Example 1 included as part of the Solstice EM release.

Note – Logs can also be created using CMIP m-Create requests or using the Low Level PMI M-Create service. For more information on using create requests and creating logs refer to CCITT X.711 ISO / IEC 9596-1 and CCITT X.735 ISO / IEC 10165-6.

When To Use:

Generally you will not need to create logs, as the pre-configured log object in the Solstice EM MIS will log most types of notifications automatically. You may want to modify the pre-configured log to handle the following cases:

- Logging ISO DMI or other standards-based notifications not handled by the pre-configured log
- Logging notifications that are specific to a device and not previously defined by a standard or supported by the pre-configured log

You may want to create a new logs in order to:

- Separately log specific information also logged by the pre-configured log (what happens to alarm management extensions if this info is logged in two different logs or is removed from the pre-configured log?)

- Create a separate log to selectively track one or more new or existing notification types (i.e., a new discriminator for the log, a new notification type for the log, or both). For example, a new discriminator could be defined to track notifications from a specific managed object instance or from a specific managed object class, or to track only one or two types of notifications.

See Also:

- CCITT X.734 ISO / IEC 10164-5.
- CCITT X.735 ISO / IEC 10164-6.
- CCITT X.711 ISO / IEC 9596-1.

3.2.4 Object Class Definitions and Name Bindings

Managed Object Classes

A *managed object class* specifies the structure and behavior of a managed object instance. A managed object class defines the following properties:

- Attributes or types of data supported by the managed object class
- Operations or actions supported by the managed object class
- Behavior exhibited by the managed object
- Notifications or the types of unsolicited information a managed object can generate and send to a manager

The managed object class does not contain values for any of the attributes defined for a managed object class, it simply defines the properties of the managed object class. The class definition is used as a mechanism for managers and agents to understand and agree on what types of data and operations are supported by instances of a managed object class.

Typically an agent will be developed by a device manufacturer. The device manufacturer will then provide one or more GDMO (or SNMP MIB) definitions that define and represent the properties of the managed object classes supported by the agent. The attributes and operations specified in the definition are intrinsic to the agent. These attributes and operations are not intrinsic to the manager (i.e., the entity acting in the manager role). The manager must be informed of the properties of each managed object class in order for the manager to effectively manage the object instance.[1]

GDMO managed object classes are loaded into the Solstice EM MIS in order to tell the MIS about which managed object classes and properties are supported by an agent. Loading managed object class descriptions will not load any attribute values into the MIS — the loading process simply loads information that describes the structure and behavior of the managed object.

For example, a definition may specify that instances of the managed object class support the operationalState attribute and that the operationalState attribute can take either of the values enabled (0) or disabled (1). However, this definition doesn't specify or provide knowledge of what the contents or actual value of the operationalState attribute in any managed object instance in the network is. It simply says instances of this managed object class contain an operationalState attribute containing one of two values. The definition may also specify that instances of the managed object class support the GET-REPLACE operation for the operationalState attribute which means that the manager is allowed to both read and write the operationalState attribute. If the definition only specified the GET operation for operationalState, it would mean that read operations would be allowed for the operationalState attribute of the instance, but write operations would result in an error responses.

Another way to describe this is as follows:

- The managed object instance stores the data values and carries out the behaviors defined for the managed object class.
- The manager (once the managed object class's GDMO definition has been loaded) can ask the agent to carry out a supported behavior, and also know what type of data the agent can return to the manager as a result of performing the behavior.

Managed object class definitions that have been loaded into the Solstice EM MIS but do not have the capability to exist as instances within the Solstice EM MIS are sometimes referred to as *remote objects* or *remote object instances*. The term "remote" simply implies that the behavior is carried out external to the Solstice EM MIS process and that the data values are also stored or maintained external to the Solstice EM MIS.

1. The properties of a managed object class are sometimes hard coded or "hard-wired" into a manager, which makes it difficult to support new managed object classes without relinking and sometimes recompiling the manager. This is not the case with the Solstice EM MIS which allows managed object classes to be added dynamically at run time.

If you are familiar with SNMP managers and agents, adding a managed object class to the Solstice EM MIS is a similar in purpose and concept to adding a MIB definition to an SNMP management platform.

Name Bindings

In addition to understanding managed object class definitions, the MIS must also understand where an instance of a managed object class can be placed in the MIT. Name binding definitions define potential or allowable containment relationships in the MIT. Name bindings are normally included in the GDMO module that defines a managed object class and are loaded into the Solstice EM MIS (along with the managed object class).

What To Use:

- GDMO Managed Object Class definition and supporting productions. This definition would be developed by you or provided by a device vendor or other vendor
- GDMO compiler: em_gdmo
- ASN.1 compiler: em_asn1

When To Use:

If you are developing a Solstice EM solution that will operate with a particular set of managed object classes supported by an agent, you will need to make sure that those managed object class definitions and corresponding name bindings are loaded into the Solstice EM MIS. This includes CMIP agents, SNMP agents, and SunNet Manager proxy agents. If your solution also depends on logical processes or other constructs represented by a managed object class, you will also need to make sure that those managed object class definitions are loaded into the Solstice EM MIS.

In general you will need to load managed object class definitions and name bindings into the Solstice EM MIS for every managed object class that will be managed by the MIS. The Solstice EM MIS is pre-loaded with a number of managed object class definitions, including:

- The ISO DMI object class definitions
- Managed object class definitions from Forum Object Library 4

- Definitions needed to support the operation of the Solstice EM MIS and the Solstice EM client applications

You may also have loaded managed object class definitions as part of setting up the MIS server and applications to manage device in your network. You do not need to reload any definitions you previously loaded.

3.2.5 Object Class Definitions with Repository Behavior

Managed object class definitions with repository behavior can be instantiated within the Solstice EM MIS. This implies that data values for instances of this object class are stored within the Solstice EM MIS and the operations for the instances are performed within the Solstice EM MIS process. In short, instances of these types of managed object classes operate in the agent role within the Solstice EM MIS.

The repository behaviors are predefined and only support operations to read and write data values that are in memory or in persistent storage. No other types of behavior or operations can be used. A repository behavior is assigned to a managed object class definition using either the em_compose_oc or em_compose_poc command. Em_compose_oc places data values in memory (non-persistent storage), and em_compose_poc places data values in persistent storage.

Because these objects exist within the Solstice EM MIS they are commonly referred to as *local objects* or *local object instances*. Existence within the Solstice EM MIS implies that the behaviors are carried out by the Solstice EM MIS and the data is stored within the MIS or its persistent store.

In addition to providing the managed object class definitions and links to repository behavior, the MIS must also understand where an instance these managed objects can be contained in the MIT. Name binding definitions are again are loaded into the Solstice EM MIS and used to define potential or allowable containment relationships in the MIT.

What To Use:

- GDMO Managed Object Class definition and supporting definitions developed by you
- GDMO compiler: em_gdmo
- ASN.1 compiler: em_asn1

- Object behavior link command: em_compose_oc or em_compose_poc
- Name binding loader command: em_load_name_bindings

When To Use:

You do not need to link repository behaviors to most object class definitions. Most of the managed objects you will deal with will be remote managed object instances that exist in your network and are not part of the Solstice EM MIS. In particular, managed object classes that represent physical devices or managed resources in your network should not have repository behavior linked to them. However, if you are developing an EM solution that requires any of the following, you should use a managed object class definition which has linked repository behaviors:

- You want to summarize and store information from one or more remote managed objects in your network, and you want this information to be available to one or more client applications or one or more subcomponents that form this or other Solstice EM solutions.
- You want to define a logical managed object class which has no physical representation (that is, it is not represented by a managed resource) in your network and share information about instances of that managed object class with multiple client applications or subcomponents that make up a Solstice EM solution.

See Chapter 4, “Developing EM Solutions” for more information about adding a new managed object class to the MIS.

3.3 Client Subcomponents

A client subcomponent is any user-developed or user-configured subcomponent that is contained in a PMI application. That means that it is part of the same UNIX process as the PMI application. The following Solstice EM subcomponents are client subcomponents:

- PMI API
- Common Functions and Classes[1]
- `xtsched`

1. Most of the common functions and classes are also used by the Solstice EM MIS.

3.3.1 High-Level Portable Management Interface (PMI)

The high-level PMI is a set of programming interfaces for developing Solstice EM client applications. The client application are user-developed client components. The high-level PMI provides a number of services to application developers. The services provided by the high-level PMI include:

- Initialization services that simplify connecting with the Solstice EM MIS
- Access to event notification, subscription and propagation services
- Access to object instance and object class information
- Representation and management of different types of relationships

Note – This is a related but different concept from ISO relationship management described in CCITT X.732 ISO/IEC 10164-3.

- Shared access to information that is stored or managed by the MIS
- Simpler syntax for naming and referring to managed objects and the use of nicknames

For a complete description of the PMI services and interfaces refer to *Solstice Enterprise Manager API Syntax Manual*. The following sections provide some overview of some of the basic PMI classes.

3.3.1.1 PMI Object Classes

The PMI contains nine C++ object classes. A PMI client application developer will typically work with one of these basic classes:

- The Image class provides a common consistent mechanism to access information from any managed object class.
- The Album class provides a common consistent mechanism for working with collections of managed object instances.
- The Platform class provides a connection to the Solstice EM MIS using a common consistent mechanism.

3.3.1.2 *Platform: Support for a Specific Solstice EM MIS*

A Platform represents an actual or potential connection to a Solstice EM MIS, along with all the implied semantics of the particular MIS. The class provides a generic attribute-like mechanism that you may use for specifying such platform-specific items as access tickets or default time-outs. The Platform class is shown in Figure 3-2.

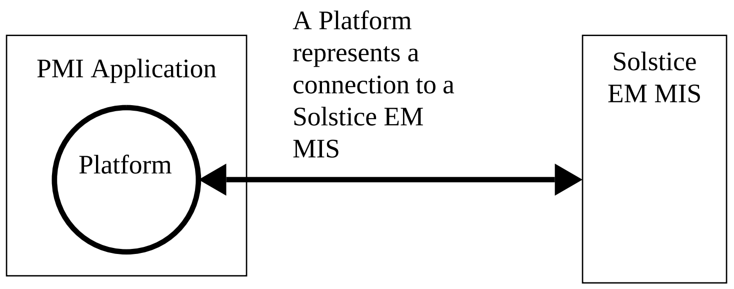


Figure 3-2 PMI Platform Object Class

The Platform object also provides connect, disconnect, and other functions in support of distributed applications and also to support applications that connect to more than one MIS.

3.3.1.3 *Image: Object in Application to Represent Managed Object*

An instance of the *Image* class is the local representation of an actual or potential managed object. Typically, an image is a managed object that represents a physical resource: a host, server, router, subnet (that is, the representation of a physical device) or a conceptual entity (a line, a queue, or some other aspect of network operation that can be represented as a managed object). The Image class is shown in Figure 3-3.

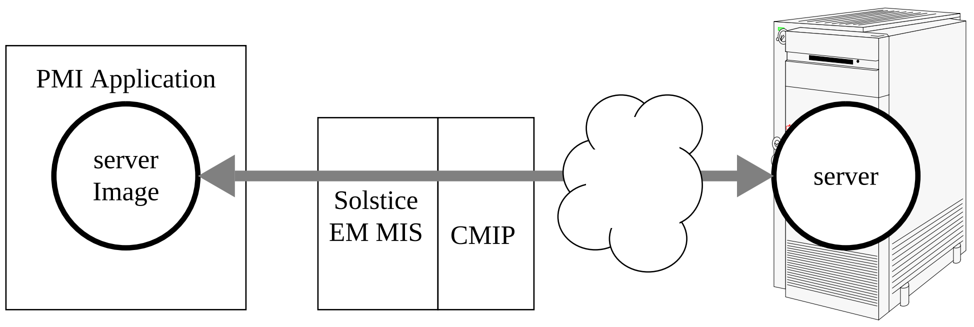


Figure 3-3 PMI Image Object Class

You may think of the Image as the object itself, even though the actual object is across the network, or in the MIS. Images give you access to the object's methods and attributes.

An Image usually represents data as text or as an ASN.1 value, but also allows “raw” data to be passed in the form of Morfs. Images also provide attribute-like access to object and attribute schema information. When representing an actual object, an Image can be synchronized with the object it represents either manually or automatically using the TRACKMODE property of the Image.

The following example shows the use of the boot function to load an image for an alarm log and then set its administrative state to locked:

```
// This code fragment is used to set the adminstrativeState
// of “AlarmLog” to locked so that any further alarms
// generated in the system are not logged
```

```
Image

     test_image(“logId=string:\”AlarmLog\”logId=string:\”AlarmL
og\””);

     test_image.boot();
     test_image.set_str(“administrativeState”, “locked”);
     test_image.store();
```

3.3.1.4 Album: Set of Images Convenient to Treat Together

An *Album* is a set of Images representing a set of objects that are somehow related. An Album thus permits you to treat a collection of images as a whole.

Like a mathematical set, an Album may be constructed either by rule or by enumeration. Certain operations can be performed on an Album, and thereby to each of the Images in the Album. Similar to Images, the membership list of an Album may be maintained either manually or automatically using the TRACKMODE property of the Album.

Figure 3-4 illustrates how a single Album can be used to represent a group of real managed objects (the four servers).

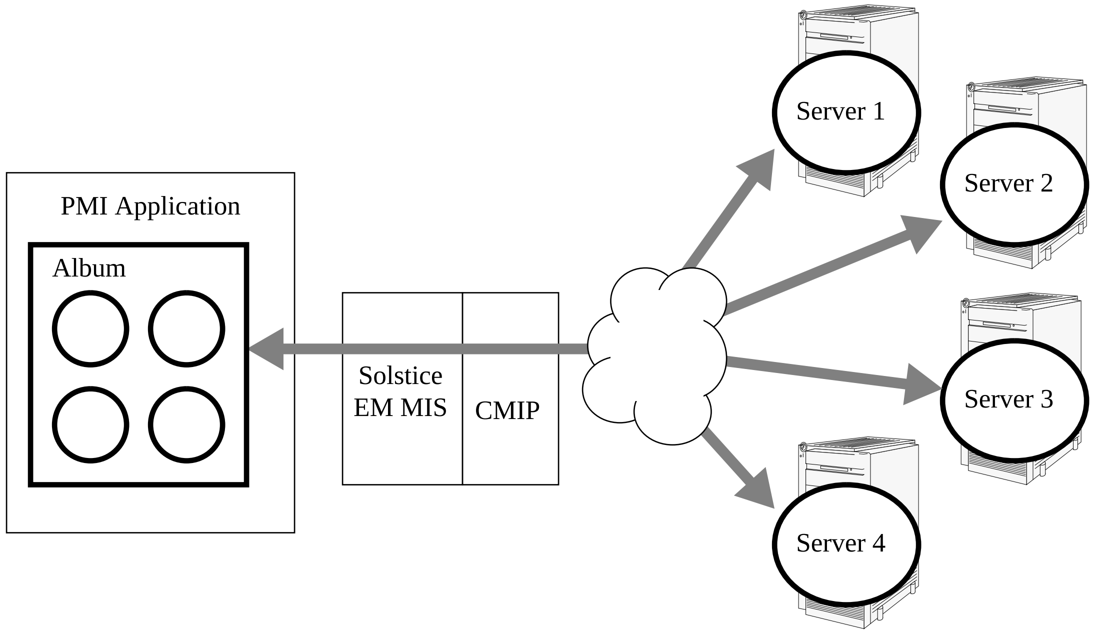


Figure 3-4 PMI Album Object Class

The following code fragment shows an example of creating an album which contains a collection of images for forumTestObjects, and then counts the number of images in the album:

```
    // This code fragment is used to
    // count all the “forumTestObjects” present in the system

    Album test_album = Album(“myalbum”);
    char *fdn = “ALL/CMISFilter(item : equality :
    {objectClass,forumTestObject})”;

    test_album.set_derivation(fdn);

    test_album.derive();
    cout << “Number of forumTestObjects in the system =” <<
    test_album.num_images() << endl;
```

3.3.1.5 *Morf and Syntax*

A unit of data is represented by an instance of the Morf class (Mysterious Object Related to Framework). Each Morf contains an opaque, encoded value, along with the information the PMI needs in order to decode it. For each framework, there’s a derived class that’s able to manipulate that type of data in the context of the framework

An instance of the Syntax class represents a type. All framework-encoded data, whether part of an object or not, has a type. This type specifies, among other things, how to produce and understand human-readable representations of the data. The Syntax class is part of the PMI’s implementation; application developers should not need to deal with Syntaxes directly except when building Morfs.

3.3.1.6 *AlbumImage*

An AlbumImage is the representation of the state of an iterator that is progressing through either all of the Images in an Album or all of the Albums containing an Image. This is analogous to a pointer into a linked list.

3.3.1.7 *CurrentEvent*

A CurrentEvent is the representation of an event. When a callback is requested from the PMI, the PMI returns the CurrentEvent parameter in the callback. The application may then interrogate the CurrentEvent by various methods to find out what kind of event occurred, which Image or Album it relates to and so on.

3.3.1.8 Waiter

A `Waiter` is the representation of an ongoing asynchronous operation. The `Waiter` provides methods for cancelling the operation or awaiting its completion. The `Waiter` may also serve as the base class for asynchronous operations

3.3.1.9 Coder

A Coder is the representation of a pair of methods, for encoding and decoding values. The Coder class is a reference-counting wrapper around an inner class, called CoderData. Application developers may derive from CoderData to provide custom translation routines to the PMI. A Coder may be bound either to a Syntax or to an attribute name

What To Use:

- Platform C++ object class
- Album C++ object class
- Image C++ object class
- Other PMI C++ object classes

When To Use:

The PMI is used to develop client applications. You can use the PMI to develop a Solstice EM solution when you have either of the following requirements:

- You need to present information in a specialized fashion that is not possible to achieve using existing Solstice EM components, such as the Viewer or the Alarm Manager. The presentation of information could include specialized presentation windows, GUI-based device front ends, or terminal output.
- You need to manipulate information in a manner that is not possible using Solstice EM subcomponents such as Nerve Center Requests, EFD's, or log objects. The manipulation of information could include summarization of data, specialized gathering and processing of data, etc.

3.3.2 Common Functions and Classes

The common functions and libraries are used to develop PMI client applications. The functions and classes are used in user-developed client subcomponents.

When To Use:

The common functions and classes are required by a number of other subcomponents -- primarily the high level and low level PMI.

3.3.3 Scheduling PMI Events

Solstice Enterprise Manager includes a scheduler, `sched`, that can be used with non-X-windows PMI client applications. This scheduler is contained in the `sched.hh` library. There is also an X-windows-aware scheduler, `xtsched`, that can be used with X-windows-based PMI applications. This scheduler is located in the `xtsched.hh` library.

A typical X-PMI application must process events on two streams:

- GUI-related X events
- PMI events to and from the platform

`xtsched` is provided for use with PMI client applications that include a graphical user interface (GUI) based on Motif and the X libraries. Both the X libraries and the PMI API present event-driven interfaces. The interfaces for both can be told to initiate an operation and then invoke an event handler (or callback) when the operation completes. Both interfaces use an internal scheduler that detects events (i.e., completion of operations, etc.) and invokes the appropriate callbacks. Not surprisingly, if either the X or PMI schedulers is prevented from running, the interface relying on the halted scheduler will also very quickly come to a halt. Two examples:

- In the case of X, if mouse and keyboard events are no longer detected, all keyboard and mouse input would stop, effectively freezing the display.
- In the case of the PMI, if responses and notifications are no longer detected, no management operations would ever complete, effectively halting the PMI client application.

`xtsched` realizes that another scheduler is running as part of a PMI application and in the `xtsched` `main` scheduler loop provides an opportunity for the xtscheduler to run.

You should use `xtsched` whenever you are developing PMI client applications that provide a graphical user interface based on X and Motif.

To implement xtscheduler, do the following in the file that contains the `main()` loop of your applications:

1. **Include the file** `xtsched.hh`**.**
2. **In** `main()`**, insert** `set_app_context(app_context)` **after** `XtAppInitialize(&app_context)`**.**
   This should occur *before* platform connect.

3.4 Additional Information about Subcomponents

3.4.1 Common Functions and Libraries: Scheduling and Callbacks

The goal of the scheduler is to discover which callback events ought to happen at the present time, and cause those callbacks to happen. There are three types of callback events that the scheduler must manage:

- Immediate callbacks
- I/O ready callbacks
- Timer callbacks

Each Solstice EM client application has its own scheduler, either `xtsched` or `sched`. The Solstice EM MIS also has a similar scheduler.

Callback events are commonly referred to as *callbacks*. Callbacks are used by the Solstice EM MIS and client applications to handle activities that occur and have an effect on Solstice EM internal resources. Most forms of software technology would simply refer to these activities as “events”, but within the context of the Solstice EM design, these activities are referred to as callback events or callbacks so as not to confuse them with network management events and event management.

Callback event routines are called *handlers*. A handler is passed two pieces of data, one specified by the routine defining the callback event, and the other specified by the routine performing the callback event. The C++ class defining a a callback event therefore contains both a pointer to the routine to be called, and the first of the two pieces of data to be passed to the routine. Both pieces of data are defined as pointers to void, which can hold any other pointer (and some kinds of integers).

A routine which wishes to define a callback event does so merely by declaring an initialized variable of the callback class. The routine may then post, or initiate, the callback event in any of several ways, depending on the desired scheduling of the callback event.

File Descriptor (I/O) Callbacks

Callback events destined to occur when I/O becomes ready are stored in arrays indexed by file-descriptor. A maximum of three callback events may be scheduled for each file descriptor: one for read events; one for write events; and one for exceptions.

Since multiple callback events and timers may be posted before the scheduler has time to process them, callback events and timers are kept in queues. The callback-event queue is kept in FIFO order, so callback events occur in the order posted.

The operating system may not provide enough file descriptors to let the client application or server keep everything open at once. Those file descriptors that could be closed if necessary should be registered with the file descriptor manager which will place the file descriptor in a priority queue, along with a callback event to be triggered when that file descriptor is to be reclaimed. file descriptors may be re-prioritized after having been registered with the file descriptor manager.

Timer Callbacks

A *timer* consists of a callback event associated with one (or more) desired times of execution. The initial time of the callback event is specified in milliseconds relative to the posting of the timer. When the timer elapses (and is noticed by the scheduler), the callback event is triggered, and the callback event handler is called. The timer may optionally reschedule itself at some time in the future.

The timer queue is kept in order of the scheduled times of callback events. Within a particular set of timers scheduled for the same time, ordering is FIFO. Times within the queue are kept relative to the previous queue entry. This increases the overhead of posting a timer, but decreases the overhead in the scheduler, which might otherwise have to scan the timer queue many times for each posted timer.

3.4.2 *Common Functions and Libraries: Asn1Value Class*

The Asn1Value class defines storage and operations for ASN.1 values and is part of the Solstice EM Common Functions and Classes. The Asn1Value class relies on the DataUnit class to provide storage for ASN.1 values.

A C++ code fragment example is provided here which shows how to use the Asn1Value class to encode a distinguished name. The distinguished name encoded in the example has the following human-readable form:

```
systemId=acid.exampleID=10
```

The BER encoded version of this distinguished name is shown in Figure 3-5:

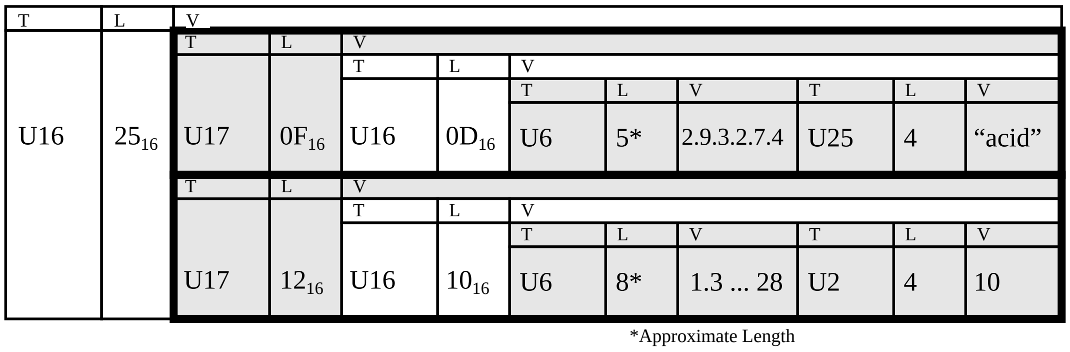


Figure 3-5 Distinguished Name Example

The distinguished name used in this example is the same DN as used in the DN example in Chapter 2, “Network Management Concepts.” The code fragment is as follows:

```
Oid        *exampleID = “1.3.6.1.2.1.42.1.28”;
Oid        *systemID = “2.9.3.2.7.4”;
DU         (char *)*id = “acid”;  //
I32        value = 10;

Asn1Value  dn;
Asn1Value  rdn1;
Asn1Value  rdn2;
Asn1Value  ava1;
Asn1Value  ava2;
Asn1Value  id1;
Asn1Value  id2;
Asn1Value  value1;
Asn1Value  value2;

Result status = OK;

if (
    dn.start_construct(TAG_SEQ) != OK ||// init the DN (FDN)
    rdn1.start_construct(TAG_SET) != OK ||// init 1st RDN
    ava1.start_constuct(TAG_SEQ) != OK ||// init 1st AVA
    id1.encode_oid(TAG_OID, systemID) != OK || // encode OID
value1.encode_octets((Octet *)id, TAG_UNIV(26)) != OK ||
                     // encode value
    ava1.add_component(id1) != OK ||// complete 1st AVA using
    ava1.add_component(value1) != OK ||// ... OID and value
    rdn1.add_component(ava1) != OK ||// complete 1st RDN

    rdn2.start_construct(TAG_SET) != OK ||// init 2nd RDN
    ava2.start_constuct(TAG_SEQ) != OK ||// init 2nd AVA
    id2.encode_oid(TAG_OID, exampleID) != OK || // encode OID
    value2.encode_int(value, TAG_INT) != OK || // encode value
    ava2.add_component(id2) != OK ||// complete 2nd AVA using
    ava2.add_component(value2) != OK ||// ... OID and value
    rdn2.add_component(ava2) != OK ||// complete 2nd RDN
    dn.add_component(rdn1) != OK ||// complete DN using
    dn.add_component(rdn2) != OK  // ... the two RDNs created

)
    Status = NOT_OK;              // If any or’s return false
```

The syntax for an object instance is as follows:

```
    ObjectInstance ::= CHOICE {
    distinguishedName  [2]IMPLICIT DistinguishedName,
    nonSpecificForm    [3]IMPLICIT OCTET STRING,
    localDistinguishedName[4]IMPLICIT RDNSequence
    }
```

In order to create an object instance name from the distinguished name (`dn`) created in the preceding code fragment, the following code fragment would also be required:

```
    Asn1Valueobj_inst;

    obj_inst.start_construct(TAG_CONT(2));
    obj_inst.add_component(dn);
```

This code fragment assumes that the distinguished name represented by dn is relative to the global root in the MIT. If we instead assume that the name represented by dn is relative to a node other than root, then the following code fragment can be used to create an object instance name that is a local distinguished name:

```
Asn1Value  local_obj_inst;
local_obj_inst.start_construct(TAG_CONT(4));
local_obj_inst.add_component(dn);
```

The three preceding code fragment examples for a distinguished name, an object instance names, and a local distinguished name also make it clear that the curly braces or slash name formats supported by the PMI are much simpler to work with. The same three names using curly braces format would be:

- Distinguished name:

```
{ { { systemId, “acid”} }, { { exampleID, 10 } } }
```

- Object instance (same as distinguished name form):

```
{ { { systemId, “acid”} }, { { exampleID, 10 } } }
```

- Local distinguished name:

```
{ { { systemId, “acid”} }, { { exampleID, 10 } } }
```

The same three names using the slash format would be:

- Distinguished name:

```
/systemId=acid/exampleID=10
```

- Object instance (same as distinguished name form):

```
/systemId=acid/exampleID=10
```

- Local distinguished name:

```
systemId=acid/exampleID=10
```

3.4.3 EFD Destination Addresses Syntax

The syntax for a destination address used by an EFD is as follows:

```
Destination ::= CHOICE
{
    single      AE-title,
    multiple    SET OF AE-title
}
```

The syntax for an application entity title (AE-title) is as follows:

```
AE-title ::= AP-title
```

The syntax for an application title (AP-title) is as follows:

```
AP-title ::= CHOICE
{
    distinguishedName DistinguishedName,
    objectIdentifier OBJECT IDENTIFIER
}
```

Solstice EM supports both the distinguished name form and the OID form for AP-Titles. However, *only the OID form will work with EFDs.*

AP-title Examples

An example of an AP-title using the OID form is as follows:

```
{ 1 2 3 4 }
```

An example of an AP-title using the distinguished name form is as follows:

```
{ { { systemId, “MasterMIS”} } }
```

Destination Examples

An example of a single destination address using the OID AP-title form of addressing is:

```
{ 1 2 3 4 }
```

An example of multiple destination addresses using the OID AP-title form of addressing is:

```
{ { 1 2 3 4}, { 1 2 3 5 } }
```

Alternative examples showing use of attribute names and values for destinations is as follows:

```
{ { { systemId, “MasterMIS”} } }
```

An example of an AP-title using the distinguished name form for multiple destinations is as follows:

```
{ { { systemId, “MIS_London”} }, { { systemID, “MIS_Dublin”} } }
```

3.4.4 CMIS Filters

The syntax for a CMIS filter is as follows:

```
CMISFilter ::= CHOICE {
    item    [8] FilterItem,
    and     [9] IMPLICIT SET OF CMISFilter,
    or      [10] IMPLICIT SET OF CMISFilter,
    not     [11] CMISFilter
}
```

The syntax for a filter item is as follows:

```
FilterItem ::= CHOICE {
    equality[0] IMPLICIT Attribute,
    substrings[1] IMPLICIT SEQUENCE OF CHOICE {
               initialString[0] IMPLICIT SEQUENCE {
               attributeIdAttributeId,
               string     ANY DEFINED BY attributeId },
               anyString  [1] IMPLICIT SEQUENCE {
               attributeIdAttributeId,
               string     ANY DEFINED BY attributeId },
               finalString[2] IMPLICIT SEQUENCE {
               attributeIdAttributeId,
               string     ANY DEFINED BY attributeId } },

    greaterOrEqual[2] IMPLICIT Attribute, -- asserted value >=
                      -- attribute value
    lessOrEqual[3] IMPLICIT Attribute, -- asserted value <=
                      -- attribute value
    present [4] AttributeId,
    subsetOf[5] IMPLICIT Attribute, -- asserted value is a
                      -- subset of attribute value
    supersetOf[6] IMPLICIT Attribute, -- asserted value is a
                      -- superset of attribute value
    nonNullSetIntersection
            [7] IMPLICIT Attribute
}
```

3.4.5 Solstice EM Managed Objects

Solstice EM contains managed object class definitions for the ISO DMI managed object classes. These object classes are automatically loaded into the product when it is installed. In addition to the ISO DMI classes, Solstice EM also contains definitions for a number of OMNIPoint 1.1 defined managed object classes.

Developing EM Solutions 4

What is a Solstice EM solution?

A Solstice EM *solution* is a collection of one or more user-configured or user-developed, client and server subcomponents that work together to solve a problem. *User-configured subcomponents* are Solstice EM subcomponents that can be created without writing any software. *User-developed subcomponents* contain application software written by an application developer using the Solstice EM APIs. *Server subcomponents* are contained in the same Unix process as the Solstice EM MIS. *Client subcomponents* are contained in a PMI client application and are in a separate Unix process from the MIS.

This chapter describes the following general types (or classifications) of Solstice EM solutions and briefly describes the some things you need to do to develop a Solstice EM solution for each class of problem:

- Element management (devices, systems, and network elements)
- Multiple MIS management problems
- Information presentation problems (PMI client application issues)
- Management information sharing
- Access control issues
- Managed object name issues

Developing a Solstice EM solution follows the basic process associated with solving any other programming problem:

1. Define your requirements
2. Design and develop your solution
3. Test your solution

This chapter provides the following information related to the above three steps:

- Descriptions of some of the concepts and issues you need to consider as you define your requirements. The concepts and issues to consider are provided for each of the four general classes of problems.
- A list of subcomponents that you are likely to use as part of the design and development of a solutions for each of the four general classes of problems.
- A brief list of some issues you may want to consider as part of testing your solution.

The classes of problems and solutions described are common types of problems that you are likely to encounter when managing your network. The four classes of problems presented here are not the only type of problems that you are likely to encounter as you manage your network. The four types of solutions presented here are also not the only types of solutions you can create using the Solstice EM product. You will likely find that as you gain more familiarity with the Solstice EM product you will “mix and match” components and subcomponents in any number of different ways in order to solve problems you encounter. The classes of problems and solutions described here are only meant as a starting point.

4.1 *Defining Your Requirements: Component and System Analysis*

There are several questions you may want or need to consider when developing a Solstice EM solution. This section lists some of the questions, but it is not meant to be a comprehensive or exhaustive list. The list is just a starting point for issues and concepts you will need to consider.

You may not be able to answer all the questions provided here, but the more of the questions you can answer, and the more questions you can pose and answer on your own, the better will be your understanding of the requirements for the Solstice EM solution that you will develop and the better that solution will be at meeting your needs. In many cases you will be able to use either default values provided with the Solstice EM product in response to some of the questions, or you will be able to use other defaults that you have created as you gain more experience with the platform.

The areas of analysis covered by this section are as follows:

- Element management (devices, systems, and network elements)
- Multiple MIS management problems
- Information presentation problems (PMI client application issues)
- Management information sharing
- Access control issues
- Managed object name issues

This section assumes that you are familiar with the principles and concepts of network management and that you have had some experience managing or developing applications to help manage a network.

4.1.1 Element Management Analysis (Devices, Systems, and Networks)

Device Type Analysis

The questions listed in this section are geared towards understanding what you will need to do to manage a new class of device that you add to your network.

- Are there any notifications, traps, or other types of unsolicited information associated with the device?

  If so, are the notifications or traps a new type, or does the device generate one of the notification or trap[1] types that the MIS already knows about? If the notification is an existing type, can it be handled as that notification type is handled currently, or does it need special handling? If the notification is new, does it need to be handled, or can it be ignored? If it needs to be handled, can it be handled like one of the existing notifications or does it require special handling?

- Does the device need to be polled?

  If so, can it use existing poll rates or should other poll rates be used? What action should be taken if the polling operation detects an error? Are the default actions sufficient? Does the polling rate need to change when an error is detected? Does an alarm need to be raised for each error detected?

- What are normal and error states that the device can be in? What severities should the normal and error states be assigned (This is a policy decision)? Who should be notified of error conditions, or is notification on the Viewer (i.e., icon color change) sufficient? Is there any standardized process or command that can be used to correct error conditions associated with the device? What device (i.e., object class) attributes or behaviors can be used to identify normal and error states?

- What is normal performance of the device. What performance states exist for the device and which states should cause alarms? What attributes or behaviors can be used to identify normal performance and abnormal performance? Can existing performance defaults be used?

1. All SNMP traps are mapped into the CMIP Internet Alarm notification, a notification which the MIS knows about. You only need to consider whether or not you need special handling for the trap information contained in the Internet Alarm.

Network and Systems Analysis

- Does this device or class of device play a key role in the network? E.g., is this device part of my backbone, or is this device a dedicated file or application server? If its a server, how frequently is it accessed? What is the impact when the server is down? If this device is polled, should the polling rate be more frequent than for other less critical devices?
- What is normal performance for the backbone of the network? What is marginal performance for the backbone? What performance levels should generate alarms?

4.1.2 Multiple MIS Management Analysis

- Will more than one Solstice EM MIS be used to manage your network? If not, you can ignore these questions and issues listed in this section.
- Which managed objects (representing devices, MISs, or other resources) will be located under root in the MIT? What are the FDNs for these objects (the RDN and the FDN will be the same in this case, because the objects are under root) and will need to be manually updated into the FDN table.
- Which managed objects will each MIS contain and manage?
- What types of information do you want to automatically pass from one MIS to another? Will one MIS act as a manager or managers (i.e., will all other MISs pass information to this manager or within the MIT will one MIS contain another MIS), will all managers act as peers (i.e., sharing certain types of information with most or all other MISs), or will some combination of a manager or managers and peers be used?

4.1.3 Information Presentation Analysis (Client Application Issues)

- How does the information need to be presented to the user? Does the information need to be presented in a specialized fashion that is not possible to achieve using existing Solstice EM components, such as the Viewer or the Alarm Manager? The specialized presentation of information could include special presentation windows, GUI-based device front ends, or terminal output. If specialized presentation of information is required, does it also make sense to still display some information using the Viewer, the Log Manager, the Alarm Manager, or other Solstice EM application? Can any other Solstice EM client application you have developed by used to display the information or part of the information?

- What information needs to be presented to a user? Where does this information come from? Is it provided from a device? Is it provided by another application (such as NerveCenter or from an EFD)? Do you need to manipulate information in a manner that is not possible using Solstice EM subcomponents such as NerveCenter Requests, EFDs, or log objects? The manipulation of information could include summarization of data, specialized gathering and processing of data, etc. Does the information come from one or more devices or applications? Can a NerveCenter Request or other Solstice EM subcomponent be designed to provide a portion of the information?
- What is the appropriate launch point for the client application? Should it be started from the application launcher? Should it be started from a NerveCenter Request? What information is needed upon start-up (e.g., context or initialization information)? Where does this start-up information come from? Does this start-up information differ from the normal information used by or displayed by this client application?

4.1.4 Management Information Sharing Analysis

- Will more than one person want to use the client application? Is concurrent access to MIS information from multiple PMI client applications required? Does the information you plan to share already exist in the MIS? (If the answer to this last question is yes, you would likely only need develop a PMI client application to access the data)
- Does the information gathered or summarized by the client application need to be stored permanently or temporarily and also made accessible to other client applications? Does some of the information only need to stored only as long as the client is running? Or as long as the server is running? In addition to data stored or accessed by the MIS, this data could also include event information or information from NerveCenter.

Note – PMI applications have shared access to information stored by or managed by (i.e., accessed) the Solstice EM MIS. The PMI does not provide a direct interface from one PMI client application to another. Any information passed from one PMI client to another to would need to flow through the MIS (as an M-Action, or M-Set that triggers an EFD, etc.) or would need to use some “out of band” service provided by the application developer (e.g., a socket connection established by one client to communicate with either another PMI client or an other non-Solstice-based application without involving the MIS).

- Do multiple copies of the client application need to receive notifications, traps, or other types of unsolicited information?
- Does information used by one copy of the client application need to be secure from information used by another copy of the same client? Or other client?

Note – Solstice EM does not currently support this type of data partitioning for security purposes.

4.1.5 Access Control Issues

Solstice EM lets you control levels of access for any application that you develop. When you create the EM applications, you need to consider the following:

- Do all users have open access to all features in your application or do you need to control access to individual features?
- How will your application respond if someone does not have access to the application?
- How will your application respond if someone does not have access to a specific feature or set of features?

When answering these questions, consider the following:

- The applications must give proper feedback to the user if the user doesn’t have access rights to perform certain operations.
- Where possible, the application should prevent the user from performing any operation that he/she is not authorized to perform.
- The application developer should also make the list of application features available to the system administrator so that the system administrator can grant users access rights to perform various operations.

For more information about designing access control for your applications, see Section 4.3, “Designing Access Control for Applications.”

4.1.6 Managed Object Name Issues

Solstice EM provides a nickname service that translates managed object names from FDNs (Fully Distinguished Names) into nicknames that are more easily comprehensible for a user. The nickname service is provided as a standard part of the EM platform. As a developer, you need to decide:

- Whether to use nicknames in place of FDNs for your applications
- How to implement getting and setting nicknames for your applications

For more information on setting up nicknames, see Section 4.4, "Using Nicknames Instead of FDNs."

4.2 Designing and Developing Your Solution

Once you understand your requirements you need to determine which Solstice EM components and subcomponents you need to use to develop your solution.

This section assumes that you are familiar with the principles and concepts of network management and that you have had some experience managing or developing applications to help manage a network.

4.2.1 Element Management (Devices, Systems, and Networks) Solutions

User Configured Subcomponents

You will typically use the following user-configured subcomponents as part of a network element solution:

- NerveCenter Requests (server subcomponent)
- Solstice EM MIS Logging Services (server subcomponent)
- You will only need to modify the default log if the notification generated by the device or system is not handled by the default log discriminator or if you wish to store the notification in a separate log.

User Developed Subcomponents

You will typically not need to use user-developed subcomponents as part of network element solution, unless you want to display the information obtained from a network element in some special way. If you have special information display requirements, you may want to consider developing a PMI client application.

Solstice EM Application and Utilities

The following Solstice EM applications and utilities are likely to be used *as part of* your solution in order to display information about the class of network element that you are adding to your network:

- Solstice EM Viewer
- Solstice EM Alarm Management Application
- Solstice EM Data Viewer

The following Solstice EM applications and utilities are likely to be used *to help create* your network element solution:

- GDMO Compiler: em_gdmo
- ASN.1 Compiler: em_asn1
- Name binding loader: em_load_name_bindings

If you are working with an SNMP device you will also need to use the following utility *to help create* your solution.

- Concise MIB Compiler: em_cmib2gdmo

If you are working with a SunNet Manager object you will also need to use the following utility *to help create* your solution:

- SunNet Manager Schema Compiler: em_snm2gdmo

Other Components, Information, or Steps You May Need

You will need one of the following types of definitions for the new class of network element that you are developing a solution for:

- GDMO Managed Object Class definition
- Concise MIB definition

- SunNet Manager Schema files

In addition, you may want to add an icon to the Viewer Object Palette to represent the new class of network element.

Where to start?

- Refer to Chapter 7, “Scenarios,” of this Guide. In particular, see Scenario 1. This scenario can potentially be used as a starting point for developing your own solution.
- Refer to the “See Also” subheading in the preceding chapter for each of the subcomponents you are using as part of your solution. The “See Also” sections lists sources of further information for each of the subcomponents.

4.2.2 Multiple MIS Management Solutions

User Configured Subcomponents

You will typically use the following user-configured subcomponents as part of a multi-MIS solution:

- Event Forwarding Discriminators (server subcomponent)
- Solstice EM MIS Logging Services (server subcomponent)

  You will only need to modify the default log if the notification generated by the other MIS or other CMIP manager is not handled by the default log discriminator or if you wish to store the notification in a separate log.

User Developed Subcomponents

You will typically not need to use user-developed subcomponents as part of multi-MIS solution, unless you want to display the information obtained other MISs or other CMIP manager in some special way. If you have special information display requirements, you may want to consider developing a PMI client application.

Solstice EM Application and Utilities

The following Solstice EM utilities are likely to be used *as to help create* your solution:

- CMIP Configuration Utility: em_oct -cmip

- Object Creation Utility: em_objcreate

The following Solstice EM applications and utilities are likely to be used *as part of* your solution in order to display information about the other MISs or CMIP platforms that you are using in your network:

- NerveCenter Requests
- Solstice EM Viewer
- Solstice EM Alarm Management Application
- Solstice EM Data Viewer
- Solstice EM Application Launcher

Other Components, Information, or Steps You May Need

You will need one of the following types of definitions that describes the other MIS or other CMIP manager that you are using as part of your multi-MIS solution:

- GDMO Managed Object Class definition or definition describing notifications emitted by the other MIS or CMIP manager.

In addition, you may want to add an icon to the Viewer Object Palette to represent the other MIS or other CMIP manager.

Where to start?

- Refer to Chapter 7, “Scenarios,” of this Guide. In particular, refer to Scenario 2. This scenario can be used as a starting point for developing your own solution.
- Refer to the “See Also” subheading in the preceding chapter for each of the subcomponents you are using as part of your solution. The “See Also” sections lists sources of further information for each of the subcomponents.

4.2.3 Information Presentation (PMI Client Application Issues) Solutions

User Configured Subcomponents

If you are only developing a PMI client application that accesses and presents data already present in the MIS, you may not need to use any user-configured subcomponents. However, most user-configured server subcomponents could be used to provide information to a PMI client application.

User Developed Subcomponents

You will typically need to use at least the following PMI C++ classes (which are all client subcomponents) to develop a PMI client application:

- Platform C++ Class
- Image C++ Class
- Album C++ Class

You may also find it useful to use other PMI C++ classes as well as the following common functions and classes:

- Asn1Value C++ Class

In addition, if you are building a GUI-based PMI client application you should also use the following client subcomponent:

- xtsched

Other Components

- GDMO Managed Object Class definitions
- Concise MIB definitions
- SunNet Manager Schema files

Where to start?

- Refer to Chapter 7, “Scenarios,” of this Guide, in particular, Scenario 3. This scenario can be used as a starting point for developing your own solution.
- Refer to the *Solstice Enterprise Manager API Syntax Manual* for further information about PMI C++ object classes.

- Refer to the “See Also” subheading in the preceding chapter for each of the subcomponents you are using as part of your solution. The “See Also” sections lists sources of further information for each of the subcomponents.

4.2.4 Management Information Sharing Solutions

User Configured Subcomponents

If you are developing an information sharing solution you may not need to use any user-configured subcomponents. However, most user-configured server subcomponents could be used to provide shared management information to one or more PMI client applications.

User Developed Subcomponents

You will need to develop a GDMO Managed Object Class definition describing the information you which to share and then create a local managed object (server subcomponent). The local managed object class is created using em_compose_oc or em_compose_poc.

You will typically need to use at least the following PMI C++ classes to develop a PMI client application that works with your local managed object:

- Platform C++ Class
- Image C++ Class
- Album C++ Class

You may also find it useful to use other PMI C++ classes as well as the following common functions and classes:

- Asn1Value C++ Class

In addition, if you are building a GUI-based PMI client application you should also use:

- xtsched

Solstice EM Application and Utilities

The following Solstice EM applications and utilities are likely to be used *to help create* your device, system or network solution:

- GDMO Compiler: em_gdmo

- ASN.1 Compiler: em_asn1
- Name binding loader: em_load_name_bindings
- Default Object Behaviors: em_compose_oc or em_compose_poc

Other Components, Information, or Steps You May Need

The following Solstice EM applications and utilities could be used *as part of* your information sharing solution:

- NerveCenter Requests
- Solstice EM Viewer
- Solstice EM Alarm Management Application
- Solstice EM Data Viewer
- Solstice EM Application Launcher

Where to start?

- Refer to Chapter 7, “Scenarios,” of this Guide, in particular, Scenario 4. This scenario can be used as a starting point for designing and developing your own solution.
- Refer to the “See Also” subheading in the preceding chapter for each of the subcomponents you are using as part of your solution. The “See Also” sections lists sources of further information for each of the subcomponents.

4.3 Designing Access Control for Applications

User Configured Subcomponents

If you are developing an information sharing solution you may not need to use any user-configured subcomponents. However, most user-configured server subcomponents could be used to provide shared management information to one or more PMI client applications.

User Developed Subcomponents

The following Solstice EM PMI functions could be used *as part of* your access control solution:

- `Platform::get_authorized_features`

- `Platform::get_authorized_applications`

Solstice EM Application and Utilities

The following Solstice EM applications and utilities are likely to be used to assign access privileges to users:

- Access Manager: em_accessmgr
- Access Control Command-Line Utility: em_accesscmd

Where to start?

- Access control examples are shipped with Solstice EM. For more information about these examples, see Section 8.5, “Access Control Examples.”

4.3.1 Platform Connection and User Profile

A user needs a connection access right to establish connection with MIS. The connection access rights are stored in the access control profile objects. The application name used in the call to `Platform::connect()` identifies the application being run. User’s privileges to run this application are checked by the MIS to determine whether to grant or deny the connection. All EM applications shipped with the product use the same name as their executable name for connection (for example, the Discover application connects as “em_discover” since the executable name for that application is `em_discover`).

Anybody can connect with any application if connection level access control is off or if the user is a super user running the application on one of the *trusted* hosts. See Section 4.3.5, “Access Control Configuration Variables” for details on the EM-config variables defining connection level access control.

4.3.2 Password Authentication

The user’s password is verified when starting EM applications, including the EM application launcher. No password is queried if the application is connecting to the local host and the user is “root” or the application being run is owned by “root” with setuid bit set. If the password is required, then it is queried automatically from inside the `Platform::connect()` function.

All GUI EM applications popup a dialog box to query the password. All other applications use the TTY interface for password query. The application can define the password query mechanism by extending the `PasswordTty` class.

The password is not queried if password authentication is off or for super users running applications on one of the *trusted* hosts. See Section 4.3.5, “Access Control Configuration Variables” for details on the EM-config variables defining password authentication.

4.3.3 Command Line Utility

When developing applications that use EM’s access control function, use the access control command line interface to create access control objects and assign privileges.

4.3.4 Feature-Level Access Control

You, as an application developer, defines features for your application. The system administrator assigns access rights for the users to use these features. The access rights are stored in MIS in the access control profile objects. Typically, at application start-up, the application finds out which features users can access by calling a PMI function. It then prevents the user from invoking any unauthorized features. It is the responsibility of the application to enforce the application feature access.

1. **After connection to the MIS succeeds, the application calls the `Platform::get_authorized_features` API function.**
2. **MIS returns the list of features the user is authorized to access.**
3. **The application enforces the application feature access control. A GUI-based application might do this by making buttons/menus and other controls inactive, corresponding to those features the user is not authorized to access.**

4.3.5 Access Control Configuration Variables

The `$EM_RUNTIME/conf/EM-config` (typically, `/var/opt/SUNWconn/em/conf/EM-config`) file contains some variables that control the MIS’s behavior with regards to password authentication and enforcement of access control. This file is read at MIS startup only. If you change this file, the changes won't be effective until the next em_services. Table 4-1provides a list of these variables, their default values, and a description for each.

Table 4-1 EM-config Variables

Variable	Default Value	Description
`EM_ACCESS_PASSWORD_CONTROL`	`TRUE`	If EM_ACCESS_PASSWORD_CONTROL is not TRUE, password authentication is turned off, password will not be queried for anybody.
`EM_ACCESS_CONNECTION_CONTROL`	`TRUE`	If EM_ACCESS_CONNECTION_CONTROL is not TRUE, then everybody is assumed to have full privilege, i.e. no access control for application connection, features, or non-access control objects modification. Password query may still be on.
`EM_ACCESS_SUPER_USERS`	<null>	EM_ACCESS_SUPER_USERS is a space separated list of user names who are considered as super users. These users get full access control privilege. They still have to go through password authentication if connecting from a remote machine. On the local machine, (or from one of the EM_ACCESS_TRUSTED_HOSTS), they don't have to provide password. The "root" user doesn't have to be explicitly included in EM_ACCESS_SUPER_USERS.
`EM_ACCESS_TRUSTED_HOSTS`	<null>	EM_ACCESS_TRUSTED_HOSTS is a space separated list of host names. If super users defined in EM_ACCESS_SUPER_USERS or the "root" user is connecting from these trusted hosts then they are not subject to password authentication and get full access control privilege. Local host is implicitly in the EM_ACCESS_TRUSTED_HOSTS. By adding a host name to this variable, you can run daemons remotely without being subject to access control or password authentication.
`EM_ACCESS_BACKWARD_COMPATIBILITY`	`FALSE`	If EM_ACCESS_BACKWARD_COMPATIBILITY is TRUE, you can connect to EM2.0 MIS with applications linked with EM1.2 libraries when run as a super user from a trusted host.

4.4 Using Nicknames Instead of FDNs

The Solstice EM nickname service translates FDNs (Fully Distinguished Names) into user-defined nicknames. Solstice EM provides the following nickname utilities that you use to configure and run the nickname service:

- `em_nnadd`—Use this utility to add a nickname server.
- `em_nnconfig`—Use this utility to define mappings between FDNs and user-defined names (nicknames).
- `em_nnmpa`—This starts and stops the nickname server daemon.

For more information about these utilities, see the Command Line Utilities chapter in the *Solstice Enterprise Manager Reference Manual*.

The High-level PMI provides four functions that a developer can use to work with nicknames:

- `album.find_by_nickname`—Use this function to find an album by its nickname rather than by its FDN.
- `image.find_by_nickname`—Use this function to find an image by its nickname rather than by its FDN.
- `image.get_nickname`—Use this function to get the nickname associated with a given FDN.
- `image.set_nickname`—Use this function to specify the nickname to be associated with a specified FDN.

For more information about these functions, see the *Solstice Enterprise Manager API Syntax Manual*. For information about nickname example shipped with Solstice EM, see Section 8.3, “High-Level PMI Examples.”

4.5 Testing Considerations

The Solstice EM product provides a number of tools and components that can help you to test your solution. This section lists development components and run time components that can help you as part of your testing process.

4.5.1 Development Components

The following debugging and testing components should be built into any user-developed subcomponents that are used in your solution:

- PMI Error Classes
- tracing and debugging classes

Refer to the *Solstice Enterprise Manager API Syntax Manual* for more information about these components.

4.5.2 Run Time Components

This section briefly describes some run time components and strategies can help you to test your Solstice EM solution.

- When you develop a solution for a network element, you may have a GDMO description for the new device or system, but the device or system may not be available or installed in your network when you are ready to begin testing your solution. Or if you are a vendor and developer of network equipment, the device or system may not be available for testing when you are ready to begin testing your management solution for the device. If you run into this situation you can use the Solstice EM MIS to simulate your device. You can do this by running two copies of the Solstice EM MIS; one copy will act in the manager role and operate with or contain your management solution and the other copy will act in the agent role and contain a local managed object that simulates your agent. You can create the agent simulator just like you create any other local managed object:

1. Parse in the GDMO managed object class description, name bindings, and ASN.1 productions into the MIS acting in the agent role

2. Run em_compose_oc or em_compose_poc for the new object class on the MIS acting in the agent role

   The agent simulator will not be able to support complex actions, notifications, or behaviors, but will be able to perform creates and deletes on managed objects, test out name binding relationships, and perform get and set operations on attributes and potentially emit DMI defined notifications.

- If you are having difficulty accessing your agent using your solution, you should also try to access the agent using the Solstice EM Data Viewer of the Solstice EM Object Editor Browser (OBED). If these two applications cannot access the agent either, then you may be having agent problems, configuration and addressing problems on your agent or on the manager, or the managed objects you need may not have been created or exist on the agent.

- When testing out NerveCenter requests, you may find it is easier to generate simulated internetAlarms than it is to generate simulated notifications.

  A sample program is included with Scenario Three, Example 6, which generates a notification. This sample program could be modified to be a generalized notification emitter. There are also a number of public domain or shareware versions of trap generators available. If you include an internetAlarm in a NerveCenter state, you can partially test out the template using an SNMP trap generator. You will eventually need to perform notification testing with a device that emits the CMIP notifications you process as part of your solution. In addition, you may also need to go back and remove the internetAlarms from your NerveCenter Request if you do not need them for the solution you are deploying.

4.6 Additional Solution Development Considerations

Developing an Solstice EM application is like design in any other area: you must make a clear definition of the requirements. For an Solstice EM application, you must decide how to make best use of the facilities the MIS provides. The following is a brief list of topics to explore as the design process progresses.

4.6.1 Consistent Object Behavior Among Applications

If an application shares some objects with other applications, it must do so in a way that makes sense in its particular context. The MIS does not enforce inter-object consistency. It depends on the object’s behavior to enforce any constraints. For example, if a circuit is composed of sub-circuits, the behavior of a circuit would be a failure if any of its sub-circuits disables the composed circuit. The Solstice EM MIS does not enforce this behavior. The implementor of the circuit object class is responsible for implementing and enforcing this constraint. The MIS simply provides the environment in which you can give an object its desired behavior.

4.6.2 Persistent Store and Data Sharing

MIS data consists of a number of objects; they can be in persistent or volatile storage. Data that needs to survive MIS shutdown is said to be *persistent*. Data that is short lived is said to be *volatile*. There are two varieties of volatile data: association-linked data and MIS-life data. Association data lasts only for the life of the association that created it. MIS-life data remains as long as the Solstice EM MIS keeps running, but is gone the next time the MIS is started.

You have to decide what makes sense for your application. If you need persistence, you must define your application object classes so that their instances are persistent. The object framework includes persistence as part of its services.

Since the object framework provides persistence as one of its services, there is little you must do to make use of it. If the data's volume or content require special attention, you have to design your application in ways that make sense. You may want to encrypt the data, or provide special buffering or compression schemes. Either of those would require you to go beyond the built-in services that the object framework provides.

4.6.3 Application Real-time Responsiveness

The Solstice EM MIS is not designed to deliver real-time responses. There is some normal latency in every request. If the application requires real-time transfer of data, an out-of-band mechanism might be appropriate. This mechanism could be whatever delivers sufficient response. The MIS could still manage and monitor this special activity, provided you can define an object with the needed behaviors and make its description available to the MIS.

You might want an application to monitor a data source continuously. This data feed might not be regular, and so might not fit easily into a `GET REQUEST/RESPONSE` model. But you could design and implement objects that:

- Poll the data source and then feed special Event Reports to the user
- Subscribe to the data source (if an interface is available) and issue event reports whenever data becomes available.

4.7 Additional Considerations for Using High-Level PMI

The high-level functions of the PMI can be used to manage all interactions with the MIS for an application that does not require extensions to the MIS. These functions provide encoding and decoding of ASN.1 values, and CMIS-like messages used for communication with the MIS and managed objects. The high-level functions also provide for initialization and for event subscription and event propagation.

4.7.1 Error Handling

Error handling is provided by the base class `Error`. Each of the object classes are derived from the `Error` class, except for the class `AlbumImage`. Refer to the “Error Handling Provided by the PMI” section in the *Solstice Enterprise Manager API Syntax Manual* for a detailed description of the Error Handling capabilities provided with the PMI. Also refer to the example of error handling in the sample program `get.cc` in Chapter 8, “API Examples.”

4.7.2 Overview of the Development Steps

To develop an application using the high-level functions of the PMI, follow these basic steps.

1. Write your application. For a full list of the object classes and their methods, refer to the *Solstice Enterprise Manager API Syntax Manual*.

   In your code, `#include <pmi/hi.hh>` should be sufficient to provide access to the PMI components.

2. Use the DevPro C++ compiler to compile and link your application.

4.7.3 Event Handling

One of the features of the Solstice EM MIS, available through the high-level functions of the PMI, is its ability to inform an application of events that occur within the MIS, or that the MIS finds out about. An application lets the MIS know the type of events it wants to hear about, and the MIS then notifies the application any time such an event occurs.

The two components of this capability are the Solstice EM MIS event loop, and application specific callbacks. An application specifies the types of events it wants to find out about, and provides the name of a callback routine that the MIS can call when such an event occurs. The callback routine is code within the application that takes appropriate action when called by the Solstice EM MIS.

The PMI provides a `Callback` class you use to specify the callback routine. The callback routine is called with a known (predefined) set of arguments, specified as part of the `Callback` class definition. There are a number of object types, such as `CurrentEvent`, and methods within the PMI that know how to decompose and manipulate the data passed to a callback routine. Thus it isn't always necessary for the application developer to have a detailed understanding of the format of the data passed with the callback.

Once you have specified your callbacks, you then initiate an event loop. In this case, you call a method from which there is no return. Control returns to your application only when one of your callbacks is invoked.

4.8 Adding a New Managed Object Class to the MIS

This section describes how to create and add a new managed object class to the MIS. This assumes that you have already determined what kind/type of object you are building.

For an example that illustrates this procedure, see Section 7.1, "Scenario 1: Adding and Managing a New Class of Device."

4.8.1 Overview of the Development Steps

1. **Determine what it is that you want to manage, and identify its characteristics.**
   What are the attributes? The attributes define the actual data which is stored about the managed object.

2. **Determine if there are any managed object classes that already exist that you can reuse.**
   To determine this, inspect the GDMO documents available in the `$EM_HOME/etc/gdmo` directory. `$EM_HOME` is an environment variable set to the directory the Solstice EM product was installed in. `/opt/SUNWconn/em` is the default installation directory.

3. **Implement the Managed Object Class**
   Code the different parts that comprise a managed object class. These are:
   a. GDMO document
   b. ASN.1 document (if necessary)
   c. Create a link in the MIS between the MOC definitions and the MIT (name bindings) such that the MIS knows where in the MIT to place instances of the new MOC.

   Run a set of utilities that creates links to the MIT and to default behaviors that exist within the MIS.

4.8.2 Details of a Managed Object Class

Note – This section details, from the application development perspective, how a developer can add a new managed object to the Solstice EM MIS. Any discussion regarding the internal representation of the managed object is only for clarification.

A managed object contains a set of services and attributes that describes a type of managed resource. It defines what type of data members exist, and the various methods which will perform various operations.

In the Solstice EM MIS, an instance of a managed object will represent a particular managed resource - data describing the resource is contained in the attributes, and the associated behaviors operate on this data. The goal therefore is to first create an appropriate managed object class that accurately defines the type of managed resource you wish to model. Next, actual instances of this managed object can be created in the MIS.

To represent the managed object within the Solstice EM MIS, a managed object class, or MOC, is created. When an *instance* of a managed object needs to be created in the MIS to represent a managed resource, the MOC is consulted to determine how to create the instance (e.g. what kinds of attributes will it have, and what are its behaviors). The representation of an instance of a managed resource within the Solstice EM MIS is known as a managed object instance, or MOI.

Both the MOC and the MOI described above are implemented in the Solstice EM MIS as C++ classes.

An overview of these pieces is shown in Figure 4-1.

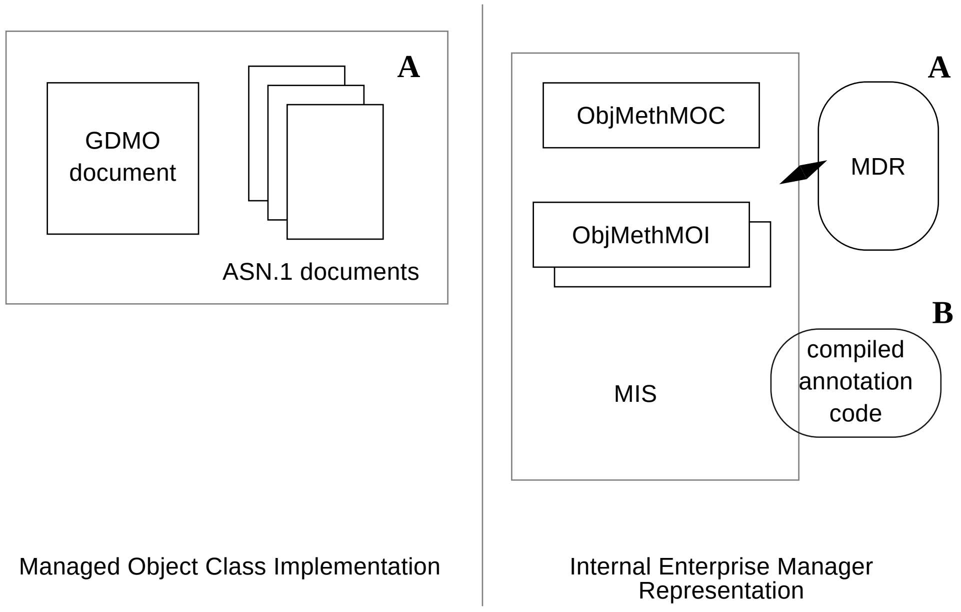


Figure 4-1 Managed Object Implementation Components

Figure 4-1 shows the relationship of the components produced by the developer to their position within the Solstice EM MIS.

The GDMO and ASN.1 documents that describe the managed object (A) are represented in the MIS within the MetaData Repository (MDR). The annotation code that implements the object behaviors (B) are linked to the MIS in the form of compiled code modules. The annotation code is used by objects and is part of the MIS.

The `ObjMethMOC` and `ObjMethMOI` objects are created dynamically by the MIS, as needed. The `ObjMethMOC` is created when needed to represent a managed object class; an `ObjMethMOI` is used when an instance of a managed object is created to represent a managed resource.

4.8.3 Defining the Properties of a Managed Object Class

The components of a MOC are specified in a GDMO document and possibly one or more ASN.1 documents. The GDMO document must contain the following:

- Managed object class definition. This includes the name of the object class, and the attributes which are in this managed object class. A unique OID is also specified such that the MOC can be identified from the MIT.
- Attribute definitions. Each attribute within the MOC has its syntax defined, along with a unique OID.
- Name Binding definition. The naming attribute is identified in the name binding, along with the various other pieces of information.
- Notification definitions. These are optional, and define any notifications the MOC should generate. The notifications can be ones from the set of predefined notifications available within the MIS, or you can define your own. Predefined notifications include such things as attribute value changes, object creation or deletion, and various error, alarm, or violation conditions.

If the GDMO document refers to any ASN.1 syntax notations, these ASN.1 documents must also exist. In most cases, the GDMO document will refer to syntax defined in existing ASN.1 documents; however if new syntax is needed, the ASN.1 document must be created. The managed object class, defined in the GDMO document, and the naming attribute(s) will be referenced according to their unique OID values.

From an Solstice EM internal implementation perspective, the OID of the managed object class is specified and the appropriate definition is then loaded into the MIS to help in creating the `ObjMethMOC` which will then represent the managed object class definition. Remember that the `ObjMethMOC` is a C++ class that represents a managed object class within the MIS. The MIS gains access to the manage object class definition (from the GDMO document) by reading the definition from the MetaData Repository. This assumes that the GDMO document was compiled into the MDR so the MIS can find the definition.

4.8.4 *Integrating a MOC into the MIS*

If you plan to link your new Managed Object Class to default behaviors, you can use the utilities `em_compose_oc` (or `em_compose_poc`) and `em_load_name_bindings` to create the links for a MOC currently installed in the MDR in an existing Solstice EM runtime environment.

Take the following steps:

1. **Edit the GDMO and ASN.1 files to define the MOC.**

   The convention is to name GDMO files as <*className*>`.gdmo` and ASN1 files as <*className*>`.asn1.`

2. **Compile the ASN.1 file.**

```
host# em_asn1 -o $EM_RUNTIME/usr/data/ASN1 -v <file>.asn1
```

   (Refer to the "Command Line Options" chapter in the *Solstice Enterprise Manager Reference Manual* for instructions).

3. **Compile the new GDMO file.**

```
host# em_gdmo -o $EM_RUNTIME/usr/data/MDR -f -v <file>.gdmo
```

   (Refer to the "Command Line Options" chapter in the *Solstice Enterprise Manager Reference Manual* for instructions).

4. **Run the command** em_compose_oc **(or** em_compose_poc**) to create links between the new MOC and a set of default behaviors.**
   Use em_compose_oc if the instances of the MOC are volatile; use em_compose_poc is the instances should be persistent. The command is located in the $EM_HOME/bin directory, and is used as follows:

```
host# em_compose_oc <className>
```

   <className> is the name of the new managed object class.

   This takes the managed object class definition, which was defined in a hierarchical fashion in the GDMO document, and 'flattens it' such that the Object Access Module within the MIS can access the definition (of the managed object class). Once this step is done, the MIS will be able to create managed object instances of this type and understand how to access the attributes.

5. **Run the command** em_load_name_bindings **to define where in the MIT a managed object instance of this type can be created.**
   The command is located in the $EM_HOME/bin directory, and is used as follows:

```
host# em_load_name_bindings <name_binding1> <name_binding2> ...
```

   <name_bindingn> are the names of the name-binding (as defined in the GDMO document) to be loaded into the MIS.

The em_compose_oc, em_compose_poc, and em_load_name_bindings have persistent effect. That is, for a specific MOC or name binding, you only need to run the command once before the next time em_services -r is run. To undo the effect, you need to remove all files of the format "MAxxxxxx" from the $EM_RUNTIME/data/MDR and $EM_RUNTIME/usr/data/MDR.

Once these steps are done, you should be able to create instances of the new MOC.

4.9 Application Development Tools

The Solstice EM Development Environment provides a set of features and utilities to aid the developer in building an Solstice EM application. These include classes and macros useful for debugging, as well as several utilities and tools useful for extending or inspecting the contents of the MIT.

4.9.1 Debugging and Tracing

The Solstice EM development environment provides a debugging facility that you can use when you run your application under debug. With this facility you can print debugging information within your application

4.9.2 Object Compilers and Viewers

Solstice EM provides several programs that translate object descriptions written in one format to another format. These programs provide ways to extend the Solstice EM configuration, and to display and adjust some of the MIS data on which applications and services rely. These translator and compiler tools are:

- Object Configuration tool
- Data Viewer tool
- GDMO compiler
- ASN.1 compiler
- Concise MIB compiler
- Schema compiler

4.9.2.1 The Object Configuration and Data Viewer Tools

The Object Configuration and Data Viewer are tools for inspecting and changing objects in the Solstice EM MIS's Management Information Tree (MIT). You can use these tools as aids in determining the state of objects your application is concerned with. Using these tools you can:

- Browse the Management Information Tree (MIT)
- Display existing collections of objects

- Display and edit CMIP and SNMP object attribute information
- Add objects to and delete objects from the MIT
- Create pointer objects that refer to remote managed resources

The Object Configuration and Data Viewer tools are described in the *Solstice Enterprise Manager Reference Manual*

4.9.2.2 The GDMO, ASN.1, Concise MIB, and Schema-to-GDMO Compilers

If your application will interact with classes of managed objects not previously known to Solstice EM, you need to add the descriptions of these objects to the Solstice EM MIS's MIT.

The Solstice EM product provides the following compilers for adding these descriptions to the MIT:

- The GDMO compiler (`em_gdmo`) processes a description written in GDMO format.
- The ASN.1 compiler (`em_asn1`) processes a description written in ASN.1 type format. Such a description may be produced as output from the Concise MIB compiler, or be supplied directly.
- The Concise MIB compiler (`em_cmib2gdmo`) translates a description written in the Internet MIB format into both GDMO format and ASN.1 format, and thus serves as a preprocessor for both the GDMO and ASN.1 compilers.
- The Schema-to-GDMO compiler (`em_snm2gdmo`) is used to translate SunNet Manager 2.x schema files to GMMO format. The resulting GDMO file must then be compiled using the GDMO compiler (`em_gdmo`).

These compilers are described in detail in the “Compilers” chapter in the *Solstice Enterprise Manager Reference Manual*.

4.9.3 Compiling

To perform a “full blown” compilation, linking every possible library, enter:

```
CC -g -I. -I/usr/openwin/include \
-I/opt/SUNWmotif/include -I/opt/SUNWconn/em/include \
-DSYSV -DSVR4 -DDEBUG \
-o <output> <file>.cc \
-L/usr/openwin/lib -L/opt/SUNWmotif/lib \
-L/opt/SUNWconn/em/lib \
-lpmi -lsched [or -lxtsched] -lnci -lnsl -lsocket \
-lemapp_toolkit -lxpm -lxmp -lpswidget -lXm -lXt -lX11 -lgen \
-R/usr/openwin/lib:/opt/SUNWmotif/lib:/opt/SUNWconn/em/lib:
```

The preceding is far more than is needed in most cases. The following is a command that will suffice for most situations:

```
CC <file>.cc -o <output> \
-L../../lib -L/usr/openwin/lib -L/opt/SUNWmotif/lib
-L<path>/pkgs/sunpro.v3.0.1/5.x-1/SC3.0 \
-lpmi -lsched [or -lxtsched] -lnci -lnsl -lsocket \
-lpswidget -lXm \
-DSYSV -DDEBUG
```

Leaner even than the previous command is the following:

```
CC <file>.cc -I/opt/SUNWconn/em/include \
-L/opt/SUNWconn/em/lib -lpmi -lsched [or -lxtsched] -lnci \
-lnsl -lsocket
```

You can remove `-lnci` from the preceding commands if you do not need the Nerve Center Interface Library. This library is described in the *Solstice Enterprise Manager API Syntax Manual*.

Also, if you want to use the 1.0 scheduler with an application, you must link with `-lsched` for non-GUI applications and `-lxtsched` for GUI applications; this is in addiction to using `-lpmi`.

4.9.3.1 *Compiling and Linking GUI Applications*

For compiling with xtsched, complete the following:

- Include the xtsched.hh file.
- Create a valid X application context (for example, XtAppInitialize(&app_context)).
- Insert set_app_context(app_context) before doing Platform::connect().

To link with the xtschedular library instead of -lsched, use `-lxtsched`. Refer to the “Scheduling for GUI Applications” section of the “Common Functions and Classes” chapter of the *Solstice Enterprise Manager API Syntax Manual*, and also the example code for Scenario two, Example 4 described in Chapter 7, “Scenarios,” of this Guide.

4.9.4 *Object Development Tools*

EM’s Object Development Tools (ODT) provide a simple and automated framework you can use to add and write behaviors for managed objects that reside in the Solstice EM MIS. You define the objects and their behaviors using GDMO and ASN.1. For more information, see Chapter 5, “Developing Object Behaviors.”

Developing Object Behaviors 5

5.1 Overview

Solstice EM provides a set of Object Development Tools (ODT) that provide a simple and automated framework you can use to add and write behaviors for managed objects that reside in the Solstice EM MIS. You define the objects and their behaviors using GDMO and ASN.1.

ODT includes the following components:

- The Object Code Generator utility (OCG) provides a set of C++ classes and methods that you use to implement the behavior for managed objects defined in GDMO. The OCG is external to the MIS server. The code generated and the user-defined implementation reside in a dynamic shared library linked to the MIS. For more information about this utility, see Section 5.5, “Object Code Generator Utility.”
- The Object Behavior Interface (OBI) lets you write and extend default object implementation for managed objects defined in GDMO. For more information, see Section 5.3.1, “Object Behavior Interface.”
- The Object Services API (OSAPI) lets you access services provided by EM MIS to implement inter-object behaviors or specialized behaviors. For more information see Section 5.3.2, “Object Services API” or the Object Services chapter in the *Solstice Enterprise Manager API Syntax Manual*.
- Framework utilities provide capabilities for building, loading, unloading, and instantiating objects.

5.2 Object Development Environment

5.2.1 Object Development Operations

The ODT allows you to perform the following operations:

- Generate code and interfaces required to support object behavior for a GDMO / ASN.1-defined object.
- Add and remove *attributes* from the GDMO definition and re-generate the code and interfaces.
- Add and remove *actions* from the GDMO definition and re-generate the code and interfaces.
- Add and remove *notifications* from the GDMO definition and re-generate the code and interfaces.
- Switch from using default behaviors to API-user-extended behaviors without re-generating the code and interfaces.
- Add or remove *discriminators* from the GDMO definition and re-generate the code and interfaces.
- Specify persistence or volatility for the attributes of an object class.

- Create and initialize an instance of a new object after the GDMO for it has been loaded.

Early releases of Solstice EM provided only limited default behaviors for an object class definition. ODT lets you define behavior for actions or define behavior for generating any events, not just the standard ones. ODT also lets you define behavior when attributes are accessed or when object instances are created and deleted.

5.2.2 *Object Development Supporting Functions*

To provide a useful object implementation, Solstice EM provides the following supporting functions:

- Ability to invoke operations on an object
- Response/error generation
- Standard event generation
- Transaction management
- Lock management
- Persistence
- Concurrent access to objects
- MIT management
- Scoping and filtering support
- Validation of user requests (For example, DELETE, GET, SET, and CREATE operations for an object follow rules defined in the GDMO.)
- Automatic instance naming
- Package inclusion

Many of these supporting functions are hidden within the EM platform. Thus, to use the framework, user-defined object implementation resides within the MIS process.

5.2.3 Object Development Components

Figure 5-1 shows the major components and interfaces the ODT provides or uses:

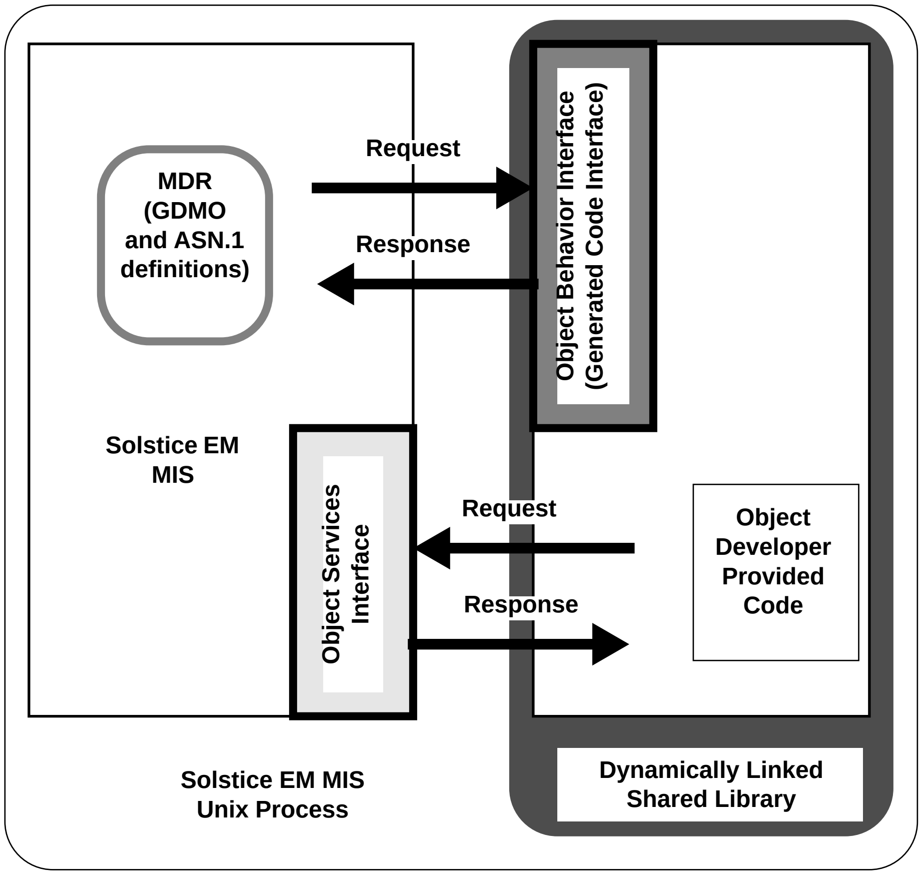

MDR (GDMO and ASN.1 definitions)
Request
Response
Object Behavior Interface (Generated Code Interface)
Solstice EM MIS
Object Services Interface
Request
Response
Object Developer Provided Code
Solstice EM MIS Unix Process
Dynamically Linked Shared Library

Figure 5-1 ODT components

5.3 Object Interfaces

The ODT provides object interfaces for object developers to use when implementing agent/manager-role behaviors using the ODT. The object interfaces are:

- An Object Behavior Interface (OBI)
- An Object Services Application Programming Interface (OSAPI)

5.3.1 Object Behavior Interface

The Object Behavior Interface (OBI) provides functions that allow the MIS to invoke (or request) object behavior functions developed by an API user and receive responses from the user developed functions. It provides the following functions:

- Attribute access for implementing behavior for GET and SET CMIP operations
- Action access for implementing CMIP ACTION operation
- Instantiation access for implementing behavior for CREATE, DELETE CMIP operation
- Notification behavior for implementing event generation and behavior to be executed on receipt of events

A large part of the software in this interface is *generated* code. This interface also contains generated *stub function interfaces* (also referred to as *stubs*) where you can add your own object behavior code. To write unique object behavior code, you can use the Object Services API or write your own C++ code.

Generated stub function interfaces are provided for the following:

- Each attribute defined for a GDMO object class
- Each action defined for a GDMO object class
- Handling the receipt of notifications by an object
- Handling special instantiation and deletion behavior defined for a GDMO object class

Except for the action stub function interfaces, you are not required to provide functionality for every stub function interface. If you do not provide functionality, default behavior functionality is used.

5.3.2 Object Services API

The Object Services Application Programming Interface (OSAPI) provides an interface that user-developed object behavior functions can use to access information and services provided by the Solstice EM MIS. The decision to use these services depends on the behavior defined for an object. For example, if an action defined for a GDMO object requires that object to check the administrativeState of the log object as part of the action, the user-developed behavior code needs to use the object services interface to issue a get request to obtain the value of the administrativeState attribute of the log object.

The OSAPI provides the following set of services that can be used within an object implementation:

- Issue a get request and (asynchronously) receive any responses
- Issue a set request and (asynchronously) receive any responses
- Issue a create request and (asynchronously) receive any responses
- Issue a delete request and (asynchronously) receive any responses
- Issue an action request and (asynchronously) receive any responses
- Issue an unconfirmed event report request

For detailed information on the OSAPI, see the Object Services chapter in the *Solstice Enterprise Manager API Syntax Manual*.

5.4 Object Development Process

5.4.1 Process Description

The process for defining object behavior is as follows:

1. **Define object classes.**
   Define and develop Managed Object Class (MOC) using GDMO and ASN.1 definitions to include the behaviors for the MOC. If you have existing GDMO and ASN.1 documents that define the appropriate behaviors, you can use those existing files.

2. **Compile and load MOC into MDR.**
   Use the GDMO / ASN.1 compiler to compile and load the GDMO and ASN.1 files into the Meta-Data Repository (MDR).

3. **Generate object code and develop behavior.**
   Use the Object Code Generator utility (OCG) to generate the default object implementation for the MOC. The OCG generates function stub interfaces, a Makefile, object loading and unloading utilities, an object instantiation program, and a README file that contains instructions about the generated files and how to extend the default implementation.

   To develop additional behavior, add C++ code at insert areas clearly identified in the generated code. The code you add implements behaviors that are defined in GDMO for the MOC.

4. **Build objects.**
   Use the OCG-generated `Makefile.`<*className*> to build default or user-extended object implementation. The Makefile builds the object implementation as a dynamic library and a PMI client program for instantiating the object.

5. **Load object implementation and restart MIS.**
   Use the object loading utility, <*className*>`.load`, to load the new object implementation into the platform. Then, restart the MIS to read the new object implementation. When the MIS restarts, the new object implementation is loaded dynamically into the MIS. From now on any CMIP operation on an object instance for this object class results in executing the behavior implemented by the user.

6. **Debug object implementation [optional].**
   User-implemented behaviors might contain errors which result in operation failures and, in some instances, MIS crashes. You can use a debugger to attach to the running MIS or directly debug `em_mis` to debug new object implementations. Using `em_debug`, you can enable or disable developer-provided object-operation traces at runtime.

For a complete scenario that illustrates this process, see Section 5.10, “Object Development Scenario Using Chai Object.”

5.4.2 *Possible Errors*

Errors can occur in the following phases of object behavior definition:

- GDMO object class definition

The GDMO and ASN.1 compiler identify syntax errors in your GDMO and ASN.1 documents. For the Object Code Generator to generate appropriate code, you must provide a complete and syntactically correct GDMO object definition.

- GDMO object class composition

  When the MIS restarts, the object class definition and behavior definition are composed and registered. Any errors in this phase display on your screen.

- Object class instantiation

  When an instance of a class is created, any problems in creating an instance arising out of an improper object definition are returned as an error for the create request.

- User-developed code

  If you add any user-defined code to the generated code, you might introduce errors.

5.4.3 Sanity Check Procedure

It might not be possible to detect all GDMO or ASN.1 errors using the GDMO / ASN.1 compiler or the object implementation process, for example, OID registration clashes, name binding and attribute mismatches for initial values, default values, and so on. Sometimes, late in your object development process, you find errors or failures that result from mundane errors in the GDMO or ASN.1 definition for the MOC. You can use the following sanity-check procedure to minimize potential problems:

1. **Comment out ACTION definitions in GDMO.**
   Comment out the ACTION definitions in the GDMO definition for the object class. You must do this because you cannot compose an object class that contains actions without loading the appropriate action implementation in a dynamic library.

2. **Compile and load object class in MDR.**
   The object class definition in GDMO and ASN.1 must be compiled and loaded in the MDR using the following commands:

```
host_name% em_gdmo -v -f -o /var/opt/SUNWconn/em/usr/data/MDR/
<className>.gdmo
host_name% em_asn1 -v -o /var/opt/SUNWconn/em/usr/data/ASN1/
<className>.asn1
```

3. **Compose object class.**
   Use the compose program to compose the new object class. This verifies the OIDs, attributes, and syntax and catches such errors as clashes with existing classes, attribute mismatches, and invalid syntax (referring to a different document/syntax label that is valid but not actually desired by the object implementor). Use the following command:

```
em_compose_oc <className>
```

Note – If you find errors in this step, you can often get additional error details by using the `oammsg*` and `mdr*` tracing flags with the `em_debug` utility.

4. **Load name bindings.**
   Use the load name binding utilities to load the defined name bindings in the platform. This detects possible errors in the name binding or naming attribute. Use the following command:

```
em_load_name_bindings <Namebinding>
```

   Repeat this for all name bindings specified in the GDMO definition for the object class.

Note – If you find errors in this step, you can often get additional error details by using the `oammsg*` and `mdr*` tracing flags with the `em_debug` utility.

5. **Create an instance.**
   Use OBED or a simple PMI program to create an instance of the MOC. This ensures that the GDMO / ASN.1 definitions are correct and that all CMIP operations can be performed. After you verify this, use OBED or the PMI program to delete the instance.

6. **Restore ACTION definitions in GDMO.**
   Remove the comments to the ACTIONS in the GDMO definition for the MOC. You should now be ready to use ODT.

7. **Remove old definitions and prepare to load new object.**
   Run `em_services -r` to reinitialize the MDR and MIS. Follow the object development process (see Section 5.4, "Object Development Process") to load your new implementation.

5.5 Object Code Generator Utility

5.5.1 Introduction

The Object Code Generator utility (OCG) generates the C++ stubs for attribute access, instance access, and action access for the class. You fill in the behavior in the stubs.

The utility hides the process by which user-defined behavior is connected to the framework. In other words, you only change code stubs for:

- Attribute access (CMIP GET and SET)
- Action access (CMIP ACTION)
- Instance access (CMIP CREATE and DELETE)
- Notification emission (CMIP NOTIFICATION)
- Discriminator-match stub to implement behavior when a discriminator matches, if the user includes the discriminator package in the class definition.

The generated code also contains debugging information to help you trace what happens at run time.

5.5.2 Software Requirements

To use the ODT and OCG, you must have the following software:

- SPARC Solaris 2.4 or later
- SparcWorks C++ Compiler 3.0.2 or later

In addition, the EM MIS must be running locally to generate implementation.

5.5.3 Generated Code Interfaces

The generated code interface provides a set of generated code stubs that can be used to invoke user developed object behavior functions. The interfaces and underlying code are produced by the OCG, which operates on information in the meta data repository (MDR) and on information in a configuration file. The GDMO and ASN.1 definitions for GDMO object need to be loaded into the MDR prior to generating the code and interfaces.

The Object Behavior Interface is shown in Figure 5-2, with the generated code interface highlighted:

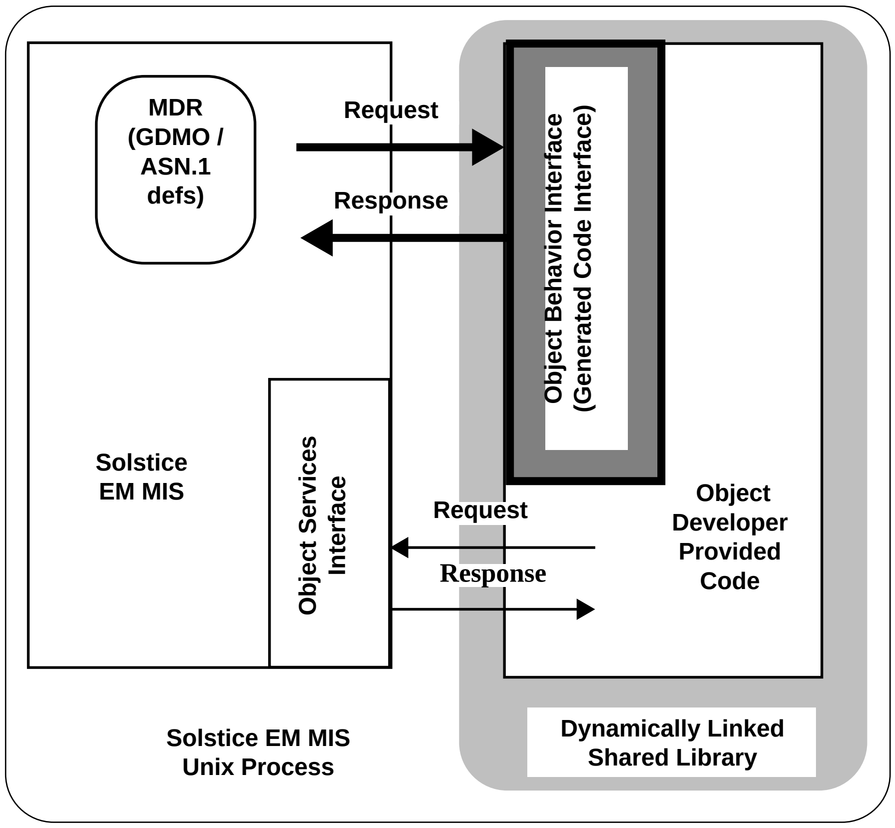

MDR (GDMO / ASN.1 defs)
Request
Response
Object Behavior Interface (Generated Code Interface)
Solstice EM MIS
Object Services Interface
Request
Response
Object Developer Provided Code
Solstice EM MIS Unix Process
Dynamically Linked Shared Library

Figure 5-2 ODT Framework, with Generated Code Interface Highlighted

5.5.4 Code Generation Components

Figure 5-3 shows the components involved in the agent role behavior code generation portion of the Object Behavior Interface. The OCG generates the appropriate agent role behavior code and code stubs for the GDMO-defined managed object class based on the GDMO definition loaded into the MDR and on parameters you specify in a configuration file. The OCG also generates a PMI client create program that you can use to instantiate an instance of the new managed object class.

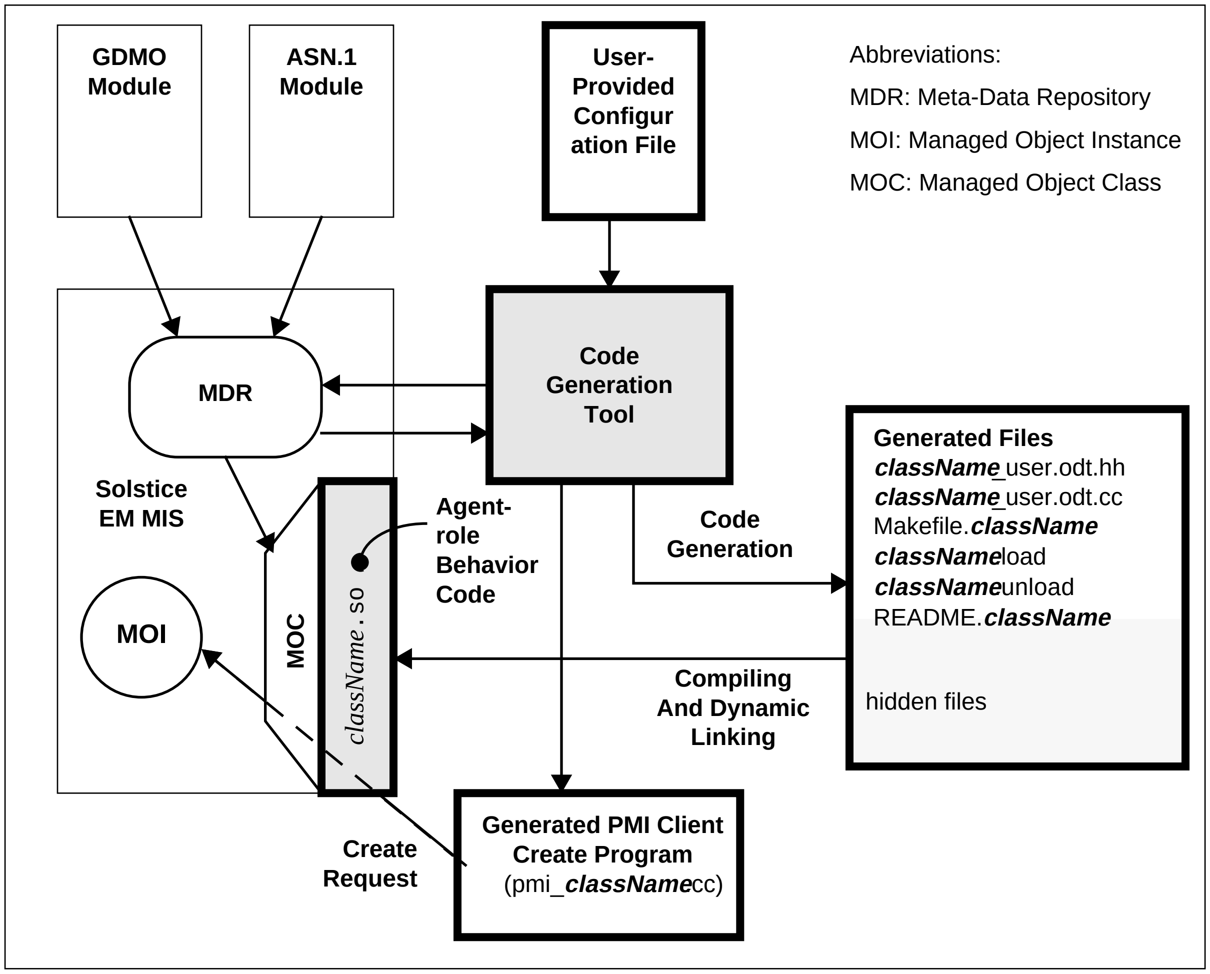


Figure 5-3 Code Generation Components

5.5.4.1 *Inputs*

The Object Code Generator utility takes input from the following sources:

- GDMO and ASN.1 files containing the class descriptions
- Configuration file

5.5.4.2 *Outputs*

The Object Code Generator utility provides the following output:

- C++ code stubs for attribute access, actions access, and instance access
  - The stub for attribute access allows you to add behavior for each attribute when a CMIP GET/SET is done.
  - The stubs for action access allow you to add behavior for each action supported by the GDMO class definition.
  - The stubs for instance access allow you to add behavior to be executed when CMIP CREATE/DELETE operations are done.
- Code that links the object implementation to the framework

  This is called the annotation code. Object implementors should not change any of this code. The annotation OID is unique and is generated automatically.

Note – All the attributes and the object instance are either persistent or volatile. You control volatility on a per-object class basis.

- Makefile that generates an object implementation (shared library)
- Utilities to load and unload object implementation dynamically
- If a GDMO object class has notification definitions, you need to add specialized behavior code to specify when the event should be generated. To do this, you use the SendEventReportRequest function provided by the Object Services API (OSAPI).
- GDMO Inheritance is supported. This means behavior defined for a superior class is re-used transparently when generating code for a derived class. You cannot override any behavior inherited from the superior class.

5.5.5 Using the Object Code Generator Utility

Before you use the OCG, you must compile the GDMO and ASN.1 definitions using the GDMO and ASN.1 compiler and restart MIS to load the GDMO and ASN.1 definitions. You then have the following options for the object behavior:

- Default behavior with persistence
- Default behavior with volatile attributes
- Non-default (user-specified) behavior with persistence
- Non-default (user-specified) behavior with volatile attributes

The OCG is a command line function. To run it, use the following command:

```
    % $EM_HOME/bin/em_obcodegen -help <filename>
```

Note – `$EM_HOME` is an environment variable used to designate the directory in which EM is installed, typically `/opt/SUNWconn/em`.

Table 5-1 identifies the options available for `em_obcodegen`.

Table 5-1 OCG Command Line Options

Option	Description
`-help`	Displays a list of command options.
<*filename*>	Identifies the class name to generate code for. The GDMO and ASN.1 files must match this file name.

Example:

To generate code for the chai example, you would use the following format:

```
% $EM_HOME/bin/em_obcodegen chai
```

5.5.6 Configuring the Object Code Generator Utility

The specific code the OCG generates depends on a number of configuration parameters. You can define these parameters in any of the following locations:

- In your login shell as user-specific environment variables
- In a local configuration file called `EM_obcodegen.cfg`
- In the global configuration file `/etc/opt/SUNWconn/em/conf/odt/EM_obcodegen.cfg`

If several developers need to use a standard configuration, use the global configuration file. Table 5-2 lists the parameters you can define for ODT configuration.

Table 5-2 Object Development Tool Configuration File Parameters

Parameter	Default Value	Description
CODEGENDIR	. (Current directory)	Directory for writing the generated code files.
DATASTORAGE	PERSISTENT	Data storage for the object class. Valid values are VOLATILE or PERSISTENT.
OBAPITRACE	YES	Enables runtime functional tracing.
OBAPIDEBUG	YES	Enables runtime debugging output.
HIDDENDIR	`.hidden`	Indicates where all the hidden annotation and implementation code is generated. Users should not modify files located in this directory.
FILTER_ATTR	DiscriminatorConstruct	If the object class needs to support event discrimination, set this flag to DiscriminatorConstruct. This causes the discrimination secretary to be generated.

5.5.7 How Filter Attributes Affect Code Generation

The GDMO definition for your object class can include three attributes that affect how code is generated for receiving events:

- DiscriminatorConstruct
- OperationalState

- AdministrativeState

If these attributes exist in your GDMO definition and FILTER_ATTR is set to DiscriminatorConstruct, then OCG generates the following code:

```
receive_event(EventType, EventInfo);
```

If FILTER_ATTR is set to DiscriminatorConstruct and any of these attributes are not defined in your GDMO, then you see a warning message and this line of code is not generated.

5.6 Debugging Objects

5.6.1 Process

1. Find out the process identifier of the running MIS.

```
host_name% ps -eaf | grep mis
```

You should see output similar to the following:

```
    root  9324     1 80 08:26:00 pts/12   0:59 em_mis -k
```

2. Run the debugger against the process identifier of the MIS.

```
host_name% debugger - <MIS_pid_from_previous_step>
```

Make sure you put a blank space between the hyphen and the process identifier.

For the output shown in the previous step, you would use the following:

```
host_name% debugger - 9324
```

3. **When the debugger comes up, go to the debugger line and open the file <*className*>`_user.cc`. This should look similar to the following:**

```
(debugger) file chai_user.cc
```

4. **Set a breakpoint in the <*className*>`_user.cc` file.**

```
(debugger) stop in <wherever>
```

5. **Continue.**

```
(debugger) cont
```

5.6.2 *Dynamic Loading in Solstice EM*

OCG generates a default object implementation for a MOC defined using GDMO and ASN.1 and loaded in the MDR. Application developers can modify the default object implementation. The object implementation is built as a shared library that is loaded dynamically into the MIS at startup (`em_services`). Similarly, object implementations can be unloaded dynamically at MIS startup.

ODT provides two utilities that are generated as part of OCG:

- <*className*>`.load` installs the shared library and adds it to the system configuration file that MIS reads to load the object implementation.
- <*className*>`.unload` removes the shared library and removes it from the system configuration file that MIS reads to unload the object implementation.

Because the object implementations are loaded at different address spaces in the MIS when the MIS is started, an application developer cannot set a breakpoint at a well-known location in the dynamically loaded shared library. To enable users to debug object implementation, `em_mis` provides a well-known breakpoint that you can use before providing other breakpoints in the dynamically-loaded object implementation.

5.6.3 ASN.1 and GDMO Debugging

Solstice EM does not provide specific tools for debugging ASN.1 and GDMO files. For complete information on these syntax definitions, see the following:

- ITU X.208 ISO / IEC 8824, Specification of Abstract Syntax Notation One (ASN.1)
- ITU X.209 ISO / IEC 8825 Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)
- ITU X.722 ISO / IEC 10165-4, Information Technology—Open systems Interconnection - Structure of Management Information—Part 4: Guidelines for the Definition of Managed Objects (GDMO)

5.6.4 Printing ASN.1 Values in Human-Readable Form

To print ASN.1 values (information in Asn1Value form) in human-readable form, first define the following in your `.cc` file:

```
Debug_on (<className>_info);
Asn1Value av;
```

Then, use the print method of Asn1Value defined in the PMI (`asn1_val.hh`) as follows:

```
av.print(<className>_info);
```

Where:

- `Debug_on(`<*className*>`_info)` is already generated by OCG in <*className*>`_user.odt.cc`.
- `Debug_on(`<*className*>`_info)` defines a Debug agent (<*className*>`_info`) that can be enabled or disabled at runtime or at compile time by using `Debug_off(),` which turns off the agent.

5.6.5 Debugging Flags

OCG generates debug agents (<*className*>`_error` and <*className*>`_info`) for every object class. If you specify OBAPIDEBUG as YES in the configuration file, OCG enables these agents at compile time. To enable or disable these agents at runtime, use `em_debug` and the following commands:

- `em_debug -c “on` <*className*>`_info”` to enable agent at runtime
- `em_debug -c “off` <*className*>`_info”` to disable agent at runtime
- `em_debug -c “on` <*className*>`_*”` to enable all debug agents for <*className*> at runtime

When debugging behaviors that require you to use Object Services API (for example, if implementing inter-object behaviors), you can enable or disable debug agents specifically for Object Services API calls. To enable these agents, use the “`objsvc_*`” options for the `em_debug` utility as follows:

```
$EM_HOME/bin/em_debug -c "on objsvc_test
$EM_HOME/bin/em_debug -c "on objsvc_error
```

Or use the following command:

```
$EM_HOME/bin/em_debug -c "on objsvc_*"
```

To disable these agents, use `em_debug -c “off objsvc_*.”`

5.7 Generated Files

If you have created valid GDMO and ASN.1 definition files and loaded them into the MDR, when you run OCG it creates the files in the target directory specified in your configuration file:

- `Makefile.`<*className*>
- `README.`<*className*>
- <*className*>`_user.odt.cc`
- <*className*>`_user.odt.hh`
- `pmi_`<*className*>`.cc`

- <*className*>`.load`
- <*className*>`.unload`

For complete examples of each of these files, see Section 5.10, "Object Development Scenario Using Chai Object."

5.7.1 Makefile (`Makefile.`*<className>)*

You use the Makefile to create ("make") a dynamic linked library for every object class for either default or user-extended implementation.

5.7.2 Readme File (`README.`*<className>)*

The README file explains how to use the files generated by OCG.

5.7.3 User Header File (<className>`_user.odt.hh`*)*

This header file contains the class definitions for the object class <*className*>. OCG generates definitions for Attribute class, Action class, and Instance class. The Attribute and Action classes contain several helper methods that users can use to access other attributes or actions defined in this file or to perform read / write actions on other attributes or actions defined in this file, while implementing specific behavior for a given attribute or implementing a specific action.

In addition to the class definitions, the user header file defines Action indices, Attribute indices, and OIDs for Attributes, Actions, and Name Bindings.

Note – You are not allowed to modify this file directly. The default object implementation build uses this file, so changes made to it will cause unpredictable results.

To add function prototypes or members in the header file, copy this file to <*className*>`_user.hh` and add your code there.

Table 5-3 identifies the Attribute classes OCG defines in this file:

Table 5-3 Attribute Class Helper Methods

Attribute Class Name	**Description**
index2AttributeSectyInfo	Converts from AttributeIndex to AttributeInfo
AttributeSectyInfo2index	Converts from AttributeInfo to AttributeIndex
index2ActionSectyInfo	Converts from ActionIndex to ActionSectyInfo
action	Performs action
read / write / fetch / store	Read / Write / Fetch / Store Attribute

Table 5-4 identifies the Action classes OCG defines in this file:

Table 5-4 Action Class Helper Methods

Action Class Name	**Description**
index2AttrSectyInfo	Converts from AttributeIndex to AttributeInfo
index2ActionSectyInfo	Converts from ActionIndex to ActionInfo
read	Reads attribute
write	Writes attribute
fetch	Fetches attribute
store	Stores attribute

5.7.4 PMI Client Create Program for Object Instantiation (`pmi_`<className>`.cc`)

The PMI client create program file contains PMI client application code that is used to instantiate an instance of the new object class after the dynamic library for the new object class has been linked into the MIS.

5.7.5 User Code File (<className>`_user.odt.cc`)

The C++ source file contains the user-level methods defined for Attribute, Action, and Instance classes. The user function stubs that OCG generates include: read, write, fetch, store, action, create_vote, and destroy_vote. In addition, OCG generates a stub for receive_event if discrimination service is used.

Note – You are not allowed to modify this file directly. The default object implementation build uses this file, so changes made to it will cause unpredictable results.

To add user-defined behaviors, copy this file to <*className*>`_user.cc` and add your code in the insertion areas clearly identified.

When implementing intra-object behaviors, you should use only the helper methods defined in Attribute, Action, and Instances classes. When implementing inter-object behavior, you should use the Object Services API calls as needed for behavior implementation.

5.7.6 Dynamic Loading File (<className>`.load`)

When you run this utility, the object implementation build is loaded into the platform as a shared library. The new object implementation is read at MIS startup.

5.7.7 Dynamic Unloading File (<className>`.unload`)

When you run this utility, the object implementation is removed from the platform. The object implementation is not read at MIS startup.

5.8 TRY Exception Macros

Solstice EM's development environment includes some exception-handling macros that are used in cases where the C++ compiler does not handle the exception. These exception-handling macros are known as *TRY macros*. The basic elements of the TRY macros are the *TRY block* and the *Handler block*.

5.8.1 Overview

The TRY block brackets the code from which you want to receive exceptions. It must be followed immediately by a Handler block in which you specify how to handle the exception.

Exceptions are scoped dynamically. What this means is that a TRY block establishes a new exception context. When you exit the TRY block, you return to the previous exception context.

5.8.2 Code Structure

The basic structure of a TRY exception is as follows:

```
TRY {
     some block of code that may generate exceptions
}

BEGHANDLERS
     CATCH macros that handle various exceptions
ENDHANDLERS
```

5.8.3 Code Examples

The following example from the chai scenario shows how the TRY macros are used in the generated code:

```
     TRY
     {
         // Fetch attribute specified by (ai)
         return subfetch(ai,cb);

     }
     BEGHANDLERS
     CATCHALL {

#ifdef ODT_DEFAULT
         return (NOT_OK);
#endif
```

```
    }
    ENDHANDLERS
}
```

5.9 Object Development Examples

Solstice EM comes with several examples that illustrate how to develop object behaviors. All of these examples are located in the `$EM_HOME/src/odt` directory. This directory includes a README file that explains how to build the examples.

Table 5-5 identifies the examples and provides a brief description of what each one includes. A complete scenario that illustrates how to develop an object is included in Section 5.10, "Object Development Scenario Using Chai Object."

Table 5-5 Object Development Examples

Class Name	**Description**
cellSample	Defines a set of intra-object complex behaviors. Basically, it looks at an object and, if its behavior changes then the behavior of its neighboring objects changes.
chai	Looks at an attribute called "chaiReady" to decide whether there is any chai (tea) ready to drink. If not, it sends an action "brewChai" to make more.
demoPing	Defines behavior of a "native agent."
demoregistry	Provides an MIS client function to operate as a "remote agent." This demonstrates how to register an application, similar to a licensing facility.
demoServer	Provides an MIS server function to operate as a "remote agent." This provides required support for demoregistry and diskInfo examples.
diskInfo	Demonstrates behavior to get information from an external (outside the MIS) process.

▼ To Compile all Examples

You can compile and run the object behavior samples shipped with EM individually or as a group. Instructions for running each of the individual samples are provided in Section 5.9.1, “cellSample” through Section 5.9.5, “diskInfo.” In addition, Section 5.10, “Object Development Scenario Using Chai Object” leads you through the entire process for the chai object in detail.

ODT provides a global Makefile that compiles all the object behavior examples. To build these examples, perform the following commands:

```
host_name# cd $EM_HOME/odt/src
host_name# Make all
```

Note – This mechanism does not currently compile the cellSample example. You must compile and run cellSample by itself.

5.9.1 cellSample

5.9.1.1 Important Code Functions

The cellSample example illustrates how to use the Object Services API. Specific sections of the code are not specifically identified as being more important than any others. You might want to look at all the code to see how the Object Services API can be used effectively.

▼ To Build the Example

1. Go to the ODT examples directory.

```
    host_name% cd $EM_HOME/src/odt/cellSample
```

2. **Copy the cellSample GDMO and ASN.1 files to the appropriate directories.**

```
    host_name% cp cellSample.gdmo $EM_HOME/etc/gdmo
    host_name% cp cellSample.asn1 $EM_HOME/etc/asn1
```

3. **Load the GDMO into the MDR.**

```
    host_name% em_services -r
```

4. **Generate the code for cellSample.**

```
    host_name% em_obcodegen cellSample
```

5. **Create the dynamic linked library for the cellSample object class for default implementation.**

```
    host_name% make -f Makefile.cellSample extended
```

6. **Load the cellSample source into an addressable location in the MIS.**

```
    host_name% ./cellSample.load
```

7. **Restart the MIS.**

```
host_name% $EM_HOME/bin/em_services
```

▼ To Execute the Example

1. **Create an instance of the cellSample object (instantiate the class).**

```
    ./cellSample
```

2. **Start OBED and run actions againt the cellSample.**

5.9.2 demoPing

5.9.2.1 Important Code Functions

The demoPing example shows how you can develop a simple native agent using the ODT.

Action Implementation:

```
//-----------------------------------------------------------------//
//                   ACTION IMPLEMENTATION                           //
//-----------------------------------------------------------------//
//  Switch for all actions specified in the GDMO definition of Managed //
//  Object Class. Add the Action implementation in individual case   //
//  statements.//

//  IMPORTANT NOTE:                                                   //
//  ---------------                                                   //
//  When implementating a Action, Please do not forget to return Action//
//  result in cd.result for individual actions in switch statement.    //
//                                                                    //

//  ODT_DEFAULT IMPLEMENTATION:                                       //
//  -----------------------                                           //
//  Default implementation returns a NULL Asn1Value and indicates     //
// success by returning CHECK_DONE in CheckData.                      //
//-----------------------------------------------------------------//


             switch(ai.local_value() )
             {
                 case IDX_pingHost:

#ifdef ODT_EXTENDED
//********* $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] **********//
            {
                DataUnit hostname;
                input.decode_octets(hostname);
                demoPing_error.print("IS %s\n",hostname.chp());

                if(!demoping(hostname, cb))
                {
                cd.result = CheckData::CHECK_ERROR;
                cb.exec(&cd);
                }
                return;
            }

//*********** $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] ********//
#endif
                     break;

             };
    #ifdef ODT_DEFAULT
    // Default implementation (returns NULL Action Response & Success)
        cd.result = CheckData::CHECK_DONE;
        cb.exec(&cd);
#endif
       }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_EXTENDED
//******* $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//******* $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

demoPing Callback Function:

```
void
demoping_cb(Ptr userdata, Ptr calldata)

{
    CheckData cd;
    demoping_userdata *d1 = (demoping_userdata *)userdata;
    DataUnit hostname = d1->hostname;

    struct sockaddr_in from;
    int len;
    char buf[1024];
    int fromlen=sizeof(from);

    if ( (len = recvfrom(d1->sockfd, (char *)buf, 1024 , 0,
            (sockaddr*)&from, &fromlen )) < 0)
    {
    demoPing_error.print("Failed in recvfrom() for ICMP Packet \n");
    cd.result = CheckData::CHECK_ERROR;
    d1->cb.exec(&cd);
    purge_fd_read_callback(d1->sockfd);
    close(d1->sockfd);
    delete d1;
    return;
    }

    demoPing_debug.print("received %d = %s\n",len,buf);
    demoPing_debug.print("from  %ld\n",from.sin_addr);

    pingreply_struct prpl;

    if(!pr_pack( (char *)buf, len, &prpl))
    {
    post_fd_read_callback(d1->sockfd,
        Callback((CallbackHandler)demoping_cb, d1));
    return;
    }

    close(d1->sockfd);
    Asn1Value direply;
    if(!make_pingrpl(&prpl,direply))
    {
    demoPing_error.print("Failed in encoding action reply\n");
    cd.result = CheckData::CHECK_ERROR;
    d1->cb.exec(&cd);
    purge_fd_read_callback(d1->sockfd);
    close(d1->sockfd);
    delete d1;
    return;
    }
    direply.print(demoPing_error);

    cd.rv = direply;
    cd.result = CheckData::CHECK_DONE;
    purge_fd_read_callback(d1->sockfd);
    close(d1->sockfd);
    d1->cb.exec(&cd);
    delete d1;
}
```

▼ To Build the Example

1. Go to the ODT examples directory.

```
host_name% cd $EM_HOME/src/odt/demoPing
```

2. Copy the demoPing GDMO and ASN.1 files to the appropriate directories.

```
host_name% cp demoping.gdmo $EM_HOME/etc/gdmo
host_name% cp demoping.asn1 $EM_HOME/etc/asn1
```

3. Load the GDMO into the MDR.

```
host_name% em_services -r
```

4. **Generate the code for demoPing.**

```
    host_name% em_obcodegen demoping
```

5. **Create the dynamic linked library for the demoPing object class for default implementation.**

```
    host_name% make -f Makefile.demoping extended
```

6. **Load the demoPing source into an addressable location in the MIS.**

```
    host_name% ./demoping.load
```

7. **Restart the MIS.**

```
host_name% $EM_HOME/bin/em_services
```

▼ To Execute the Example

1. **Create an instance of the demoPing object (instantiate the class).**

```
    ./pmi_demoPing
```

2. **Start OBED and run action on instance specifying the hostname you want to ping.**

3. **Alternatively, you can run the ODT Sample Program driver (**`odtsamples`**) and select the Ping option.**

5.9.3 *demoregistry*

5.9.3.1 *Important Code Functions*

Action Implementation:

```
//-------------------------------------------------------------------//
//                       ACTION IMPLEMENTATION                        //
//-------------------------------------------------------------------//
//  Switch for all actions specified in the GDMO definition of Managed //
//   Object Class. Add the Action implementation in individual case   //
//   statements.      //

//  IMPORTANT NOTE:                                                    //
//  ---------------                                                    //
//  When implementating a Action, Please do not forget to return Action//
//  result in cd.result for individual actions in switch statement.   //
//                                                                     //

//  ODT_DEFAULT IMPLEMENTATION:                                        //
//  -----------------------                                            //
//   Default implementation returns a NULL Asn1Value and indicates    //
//  success by returning CHECK_DONE in CheckData.                      //
//-------------------------------------------------------------------//

             switch(ai.local_value() )
             {
                 case IDX_emDemoRegistryReg:
#ifdef ODT_EXTENDED

//********** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] **********//
            {
                Asn1Value hostasn1;
                Asn1Value appasn1;
                Asn1Value appidasn1;
                Asn1Value temp;
                DataUnit appname;
                DataUnit hostname;
                I32 appid;
```

```
                 input.first_component(temp);
                 temp.first_component(hostasn1);
                 input.next_component(temp,temp);
                 temp.first_component(appasn1);
                 input.next_component(temp,temp);
                 temp.first_component(appidasn1);

                 hostasn1.decode_octets(hostname);
                 appasn1.decode_octets(appname);
                 appidasn1.decode_int(appid);

                 if(!register_me(appname, hostname, appid, cb))
                 {
                 cd.result = CheckData::CHECK_ERROR;
                 cb.exec(&cd);
                 }
                 return;
             }

//********** $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] **********//
#endif
                      break;
                  case IDX_emDemoRegistryValidateCookie:

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *********//
            // Use it to implement some good cookie eating in your app
//*********** $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] **********//
#endif


                      break;
                  case IDX_emDemoRegistryUnreg:

#ifdef ODT_EXTENDED
//********* $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] **********//
             {
                 Asn1Value hostasn1;
                 Asn1Value appasn1;
                 Asn1Value appidasn1;
                 Asn1Value temp;
```

```
                  DataUnit appname;
                  DataUnit hostname;
                  I32 appid;

                  input.first_component(temp);
                  temp.first_component(hostasn1);
                  input.next_component(temp,temp);
                  temp.first_component(appasn1);
                  input.next_component(temp,temp);
                  temp.first_component(appidasn1);

                  hostasn1.decode_octets(hostname);
                  appasn1.decode_octets(appname);
                  appidasn1.decode_int(appid);

                  if(!unregister_me(appname, hostname, appid, cb))
                  {
                  cd.result = CheckData::CHECK_ERROR;
                  cb.exec(&cd);
                  }
                  return;
              }

//********** $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] **********//
#endif
                       break;
             };

#ifdef ODT_DEFAULT
    // Default implementation (returns NULL Action Response & Success)
        cd.result = CheckData::CHECK_DONE;
        cb.exec(&cd);
#endif
```

```
        }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_EXTENDED
//******** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//******** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

Function to Register Application:

```
Result
register_me(DataUnit &appname, DataUnit &hostname, int appid,
       Callback &cb)
{

    demoregistry_userdata *d1=new demoregistry_userdata;
    emDemoRegister info;

    strcpy(info.appname,appname.chp());
    strcpy(info.hostname,hostname.chp());

    if(! demoClientSend( &info, DemoRegister, d1->sockfd,
hostname.chp()) )
    return(NOT_OK);

    d1->cb = (Callback *)&cb;
    d1->appname = appname;
    d1->hostname = hostname;
    d1->appid = appid;

    post_fd_read_callback(d1->sockfd,
      Callback((CallbackHandler)demoregistry_cb, d1));

    return OK;
}
```

Function to Unregister Application:

```
Result
unregister_me(DataUnit &appname, DataUnit &hostname, int appid,
       Callback &cb)
{

    demoregistry_userdata *d1=new demoregistry_userdata;
    emDemoRegister info;

    strcpy(info.appname,appname.chp());
    strcpy(info.hostname,hostname.chp());

    if(! demoClientSend( &info, DemoUnregister, d1->sockfd,
hostname.chp()) )
    return(NOT_OK);

    d1->cb = (Callback *)&cb;
    d1->appname = appname;
    d1->hostname = hostname;
    d1->appid = appid;

    post_fd_read_callback(d1->sockfd,
      Callback((CallbackHandler)demounregistry_cb, d1));

    return OK;
}
```

▼ To Build the Example

1. Go to the ODT examples directory.

```
    host_name% cd $EM_HOME/src/odt/demoregistry
```

2. Copy the demoregistry GDMO and ASN.1 files to the appropriate directories.

```
    host_name% cp demoregistry.gdmo $EM_HOME/etc/gdmo
    host_name% cp demoregistry.asn1 $EM_HOME/etc/asn1
```

3. **Load the GDMO into the MDR.**

```
    host_name% em_services -r
```

4. **Generate the code for demoregistry.**

```
    host_name% em_obcodegen demoregistry
```

5. **Create the dynamic linked library for the demoregistry object class for default implementation.**

```
    host_name% make -f Makefile.demoregistry extended
```

6. **Load the demoregistry source into an addressable location in the MIS.**

```
    host_name% ./demoregistry.load
```

7. **Restart the MIS.**

```
host_name% $EM_HOME/bin/em_services
```

▼ To Execute the Example

1. **Create an instance of demoregistry class (instantiate the class).**

```
    ./pmi_demoregistry
```

2. **Run demo_server on your local host or, if running on a remote host make sure EM is installed on the remote host.**
3. **Start OBED and find the object instance under EM-MIS.**
4. **Click on the OI and issue a DemoReg action where the ActionInfo parameter is** `{“hostname”,  “ApplicationName”, 345}`**.**

5. **Alternatively, you can run the ODT Sample Program driver (`odtsamples`) and run the Register option or the UnRegister option.**

5.9.4 *demoServer*

▼ To Build the Example

1. **Go to the ODT examples directory.**

```
host_name% cd $EM_HOME/src/odt/demoServer
```

2. **Create the dynamic linked library for the demoServer object class for default implementation.**

```
host_name% make
```

▼ To Execute the Example

To start the demo_server:

```
> demo_server
```

5.9.5 *diskInfo*

▼ To Build the Example

1. **Go to the ODT examples directory.**

```
host_name% cd $EM_HOME/src/odt/diskInfo
```

2. **Copy the diskInfo GDMO and ASN.1 files to the appropriate directories.**

```
host_name% cp diskInfo.gdmo $EM_HOME/etc/gdmo
host_name% cp diskInfo.asn1 $EM_HOME/etc/asn1
```

3. **Load the GDMO into the MDR.**

```
host_name% em_services -r
```

4. **Generate the code for diskInfo.**

```
host_name% em_obcodegen diskInfo
```

5. **Create the dynamic linked library for the diskInfo object class for default implementation.**

```
host_name% make -f Makefile.diskInfo extended
```

6. **Load the diskInfo source into an addressable location in the MIS.**

```
host_name% ./diskInfo.load
```

7. **Restart the MIS.**

```
host_name% $EM_HOME/bin/em_services
```

▼ To Execute the Example

1. **Create an instance of the diskInfo object (instantiate the class).**

```
./pmi_diskInfo
```

2. **Start OBED and run an action on the object instance, specifying the hostname about which you want to get disk information.**

3. **Alternatively, you can run the ODT Sample Program driver (`odtsamples`) and select the DiskInfo of a Host option.**

5.10 *Object Development Scenario Using Chai Object*

The sample files shipped with Solstice Enterprise Manager for object development scenarios provide important information in README files. Although the information in the README files is fairly complete, this section of the documentation provides an expanded view of the object development scenario for the Chai managed object.

▼ To Create Your own Object Class

1. **Load the chai managed object into the Meta Data Repository (MDR).**

```
# em_gdmo <hostname> chai.gdmo
# em_asn1 -o ‘pwd’ chai.asn1
# cp 1.3.6.1.4.1.42.2.2.2.1.96.3.1
/var/opt/SUNWconn/em/usr/data/ASN1
# cp *-ASN1 /var/opt/SUNWconn/em/usr/data/ASN1
```

2. **Define environment variables, if needed.**
   Location of hidden/intermediate files:
   `HIDDENDIR=/tmp`

   Data storage for OC whether PERSISTENT or VOLATILE:
   `DATASTORAGE=PERSISTENT`

3. **Go to the directory where you want your code to be generated.**

```
% cd user_directory
```

4. **Generate the C++ code for your objects.**
   You will see output similar to the following:

```
host_name% em_obcodegen chai
objdefn_info:   Attribute Name is chaiKettleNumber
objdefn_info:   Attribute Name is chaiBlend
objdefn_info:   Attribute Name is chaiReady
objdefn_info:   Attribute Name is discriminatorConstruct
objdefn_info:   Attribute Name is administrativeState
objdefn_info:   Attribute Name is operationalState
objdefn_info:   Total Number Of Attributes is 6
objdefn_info:   ********************************
objdefn_info:   Action Name is brewChai
objdefn_info:   Total Number Of Actions is 1
objdefn_info:   *****************************
objdefn_info:   Name Binding Name is chai-system
objdefn_info:   Name Binding Name is chai-chai
objdefn_info:   Total Number Of Name Bindings is 2
objdefn_info:   ***********************************
```

   Note that OCG identifies the attributes, actions, and name bindings for which code will be generated.

   The following files are generated:
   - `Makefile.chai`
   - `README.chai`
   - `chai.load`
   - `chai.unload`
   - `chai_user.odt.cc`
   - `chai_user.odt.hh`
   - `pmi_chai.cc`

5. **Compile and make the dynamic library for the default implementation.**

```
% make -f Makefile.chai default
```

   This command compiles and makes a dynamic library called `chai.so`.

6. **Create a customized implementation.**

   To create a customized implementation, you need to first modify the source code. Then, to compile and make a customized library, use the following format:

```
% make -f Makefile.chai extended
```

7. **Load the new dynamic library.**

```
% chai.load
```

8. **Terminate and restart the MIS.**

```
# em_services
```

9. **Instantiate the new chai object.**

```
% cd chai_Create_program
% make chai
% chai// This creates the chai object
% chai -g// This gets attributes of new chai object
```

10. **Run the debugger to verify the object behaves as expected.**

```
% debugger $EM_HOME/bin/em_mis &
    stop in DynLoader::DynLoader
    run
```

5.10.1 Debugging Flags

The following code lines that contain the debug agents for the chai object class are included in the `chai_user.odt.cc` file.

```
    Debug_on(cahi_info)
    Debug_on(chai_error)
```

If debug agents are spread across multiple files, the above definitions must only be included in the `chai_user.hh` file (copied and modified from `chai_user.odt.hh`). The other files need to contain the following code lines:

```
    extern Debug chai_info;
    extern Debug chai_error;
```

5.10.2 Sample Behavior Implementation

The following program implements specialized behavior for the chaiReady attribute. If chaiReady is 0, then set the chaiBlend to "Earl Grey" and send brewAction.

To add this behavior, insert the following piece of code in `chai_user.cc` at the location of `chai_AttrSecty::read()` after the subread is performed.

```
// User Behavior Extension: Start
//  Def: If the chaiReady is equal to 0 then
//      Set the blend to "Earl Grey" and then send a action
//      to brewchai.

// SectyInfo definition
    AttrSectyInfo     AttrInfo;
    ActionSectyInfo   ActionInfo;

    Asn1Value  ready, blend, Orig_Blend;
    I32   Is_chai_Ready;

// Get original index of our Attribute
    int org_index = AttrSectyInfo2index(ai);
```

```
// Check to see we're reading chaiReady, if yes, then
specialize
// the behavior
    if (org_index == IDX_chaiReady) {
        if (av)     {
    av.decode_int(Is_chai_Ready);
    if (!Is_chai_Ready)
    {
    // We're out of chai.. brew and choose my blend
        // Earl Grey
        index2AttrSectyInfo(IDX_chaiBlend, AttrInfo);
        (void) fetch(AttrInfo, NULL_CALLBACK);
        (void) read(AttrInfo, Orig_Blend);

        DataUnit O_blend;
        Asn1Value new_blend;


        Orig_Blend.decode_octets(O_blend);
        chai_debug.print("Read Secretary - chai_Blend is\n");
        Orig_Blend.print(chai_debug);
        if (O_blend != DataUnit("Earl Grey"))
        {
            // Special check to blend only Earl Grey
            DataUnit chai_blend("Earl Grey");

            new_blend.encode_octets(TAG_OCTSTR, chai_blend);
            (void) write(AttrInfo, new_blend);


            (void) store(AttrInfo, NULL_CALLBACK);
         }
           // Go Ahead and Brewchai
           index2ActionSectyInfo(IDX_brewchai, ActionInfo);
    action(ActionInfo, new_blend, NULL_CALLBACK);
    }
    }
}
```

5.10.3 *chai Object Class Definitions*

The example GDMO and ASN.1 definitions for the chai object class are installed into the `$EM_HOME/src/odt/chai` directory when you install the ODT onto your system (SUNWemobj package).

The `chai.gdmo` file defines the following attributes:

- chaiKettleNumber
- chaiBlend
- chaiReady

The `chai.gdmo` file also defines the following action:

- brewChai

5.10.3.1 *Sample* `chai.gdmo` *Definitions File*

```
-- Copyright 03 Apr 1996 Sun Microsystems, Inc. All Rights Reserved.--
-- #pragma ident  “@(#)chai.gdmo1.2 96/04/03 Sun Microsystems”

MODULE “EM Chai Document”

basechai MANAGED OBJECT CLASS
    DERIVED FROM “Rec. X.721 | ISO/IEC 10165-2 : 1992” : top;

    CHARACTERIZED BY
    chaiPackage;
    REGISTERED AS { em-chai-objectClass 0 };

chaiPackage PACKAGE
    BEHAVIOUR chaiPackageDefinition BEHAVIOUR DEFINED AS
    !This managed object class represents the chai
    from the neighbourhood chai shop !;
    ;
    ATTRIBUTES
    chaiKettleNumber GET-REPLACE,
    chaiBlend  GET-REPLACE,
    chaiReady  GET-REPLACE,
    “Rec. X.721 | ISO/IEC 10165-2 : 1992” : discriminatorConstruct
        REPLACE-WITH-DEFAULT
        DEFAULT VALUE Attribute
```

```
ASN1Module.defaultDiscriminatorConstruct
       GET-REPLACE,
    “Rec. X.721 | ISO/IEC 10165-2 : 1992” : administrativeState
       GET-REPLACE,
    “Rec. X.721 | ISO/IEC 10165-2 : 1992” : operationalState
       GET;
    ACTIONS
    brewChai;
    NOTIFICATIONS
    “Rec. X.721 | ISO/IEC 10165-2 : 1992” :
       objectCreation,
    “Rec. X.721 | ISO/IEC 10165-2 : 1992” :
       objectDeletion,
    “Rec. X.721 | ISO/IEC 10165-2 : 1992” :
       attributeValueChange;
    REGISTERED AS { em-chai-package 1 };

chai MANAGED OBJECT CLASS
    DERIVED FROM basechai;
    REGISTERED AS { em-chai-objectClass 1 };

-- Actions

brewChai ACTION
    MODE CONFIRMED;
    WITH INFORMATION SYNTAX Chai-ASN1.ChaiString;
    WITH REPLY SYNTAX Chai-ASN1.ChaiString;
    REGISTERED AS { em-chai-action 1 };

-- Name Bindings

chai-system NAME BINDING
    SUBORDINATE OBJECT CLASS chai;
    NAMED BY
    SUPERIOR OBJECT CLASS “Rec. X.721 | ISO/IEC 10165-2 : 1992” : system;
    WITH ATTRIBUTE chaiKettleNumber;
    BEHAVIOUR chai-rootBehaviour BEHAVIOUR DEFINED AS
    !This name is used to define the chai object
    name binding!;
    ;
    CREATE;
    DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
    REGISTERED AS { em-chai-binding 1 };
```

```
chai-chai NAME BINDING
    SUBORDINATE OBJECT CLASS chai;
    NAMED BY
    SUPERIOR OBJECT CLASS chai;
    WITH ATTRIBUTE chaiKettleNumber;
    BEHAVIOUR chai-systemchai BEHAVIOUR DEFINED AS
    !This name is used to define the chai object
    name binding under system branch!;
    ;
    CREATE;
    DELETE ONLY-IF-NO-CONTAINED-OBJECTS;
    REGISTERED AS { em-chai-binding 2 };

-- Attributes

chaiKettleNumber ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Chai-ASN1.ChaiInteger;
    MATCHES FOR EQUALITY;
    BEHAVIOUR chaiKettleNumberBehaviour BEHAVIOUR DEFINED AS
    !This is the naming attribute for  the chai
    object.!;
    ;
    REGISTERED AS { em-chai-attribute 1 };

chaiBlend ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Chai-ASN1.ChaiString;
    MATCHES FOR EQUALITY;
    BEHAVIOUR chaiBlendBehaviour BEHAVIOUR DEFINED AS
    !This is the blend of chai that is  brewing
    in the current Kettle!;
    ;
    REGISTERED AS { em-chai-attribute 2 };

chaiReady ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Chai-ASN1.ChaiInteger;
    MATCHES FOR EQUALITY;
    BEHAVIOUR chaiReadyBehaviour BEHAVIOUR DEFINED AS
    !If this attribute is true there is chai in
    the Kettle!;
    ;
    REGISTERED AS { em-chai-attribute 3 };

END
```

5.10.3.2 Sample `chai.asn1` Definitions File

```
-- Copyright 03 Apr 1996 Sun Microsystems, Inc. All Rights Reserved.--
-- #pragma ident  "@(#)chai.asn11.2 96/04/03 Sun Microsystems

Chai-ASN1
{iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) sun(42)
     products(2) management(2) em(2) odt(1) em-chai(96)
asn1Module(2) 0}

DEFINITIONS ::=
BEGIN
em-chai OBJECT IDENTIFIER ::=
    {iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) sun(42)
     products(2) management(2) em(2) odt(1) em-chai(96)}

em-chai-objectClass  OBJECT IDENTIFIER ::= { em-chai 3 }
em-chai-package      OBJECT IDENTIFIER ::= { em-chai 4 }
em-chai-binding      OBJECT IDENTIFIER ::= { em-chai 6 }
em-chai-attribute    OBJECT IDENTIFIER ::= { em-chai 7 }
em-chai-action       OBJECT IDENTIFIER ::= { em-chai 9 }

ChaiInteger ::= INTEGER
ChaiString ::= GraphicString
ChaiBoolean ::= BOOLEAN
END
```

5.10.4 Sample PMI Program to Create a New chai Object Instance

```
/*
“This file is generated using Solstice EM (2.0) - Object Development
Tools” Code Generator
*/
```

```
#include <hi.hh>
#include <error.hh>
#include <sys/types.h>
#include <unistd.h>
#ifdef HPUX
#include <sys/param.h>
#else
#include <sys/systeminfo.h>
#endif

void create_chai( DU &dn);

main(int argc, char **argv)
{
    printf(“MODIFY THE GENERATED CODE FILE ./pmi_chai.cc \n”);
    exit(0);

    Platform plat(duEM);

    if (plat.get_error_type() != PMI_SUCCESS)
    {
        printf(“Platform constructor failed...\n”);
        printf(“Reason: %s\n”, plat.get_error_string());
        exit(1);
    }

    // Initialize to the dn of object

    // dn can be a name starting from local root or fully distinguished
name
    DU dn; /* DISTINIGUISHED NAME OF OBJECT HERE*/

    // Connect to the mis running on the local host
    if (!plat.connect(“localhost”, “chai_sample”))
    {
    printf(“Connecting to platform Failed \n”);
        printf(“Reason: %s\n”, plat.get_error_string());
    exit(2);
    }
    // Create the object
    create_chai(dn);

}

void
display_attributes(Image &im)
{

// Get all the attribute names in an Array of DataUnits.
// Perform a get on each attribute to get its value
// Note we have stripped off the document name to make
// the attribute value pairs more readable
// the chp() method of the DataUnit is necessary to null
// terminate the DataUnit.
//

    Array(DU) attr_names = im.get_attr_names();
    fprintf(stdout, “Attribute\tValue\n---------\t-----\n”);
    for (int i=0; i<attr_names.size; i++) {
        DU& name = attr_names[i];
        // note: next to lines use chp() function to convert a
        // DataUnit into char *.
        char *short_name = strrchr(attr_names[i].chp(),’:’);
        fprintf(stdout, “%s: \t%s \n”, ++short_name,
       im.get_str(name,USE_EXPLICIT_CHOICE|OMIT_NEWLINES).chp());
    }
    fprintf(stdout, “\n\n”);
    fflush(stdout);
}

void
create_chai(DU &dn)
{

    Image im;
    im = Image(dn,DU(“chai”));
    if (!im.boot())
    {
    printf(“Image::boot Failed %s\n”,im.get_error_string());
    exit(3);
    }

    /* UNCOMMENT AND MODIFY THE APPROPRIATE LINES IF YOU WANT TO CREATE
     * OBJECT WITH SOME ATTRIBUTE VALUES
    if (!im.set_str(“chaiKettleNumber”, “None”))
    {
    printf(“Image::set_str() failed for chaiKettleNumber: %s %d\n”,
           im.get_error_string(), im.get_error_type());
    exit(4);
    }
    if (!im.set_str(“chaiBlend”, “None”))
    {
    printf(“Image::set_str() failed for chaiBlend: %s %d\n”,
           im.get_error_string(), im.get_error_type());
    exit(4);
    }
    if (!im.set_str(“chaiReady”, “None”))
    {
    printf(“Image::set_str() failed for chaiReady: %s %d\n”,
           im.get_error_string(), im.get_error_type());
    exit(4);
    }
     *
     */
    if (im.get_error_type() != PMI_SUCCESS)
    {
    printf(“Image::set_str Failed %s\n”,im.get_error_string());
    exit(5);
    }

    if (!im.create())
    {
    printf(“Create Failed %s\n”,im.get_error_string());
    exit(6);
    }
    printf(“Created instance %s\n”, dn.chp());
     display_attributes(im);
}
```

5.11 Generated Interfaces and Examples

The following list summarizes the functions or points in the generated code that can be modified by a user of the Object Behavior Interface. For most objects, you will **not** need to supply additional code for every interface point listed here:

- Attribute secretary read function: *className*_`AttrSecty::read`
- Attribute secretary write function: *className*_`AttrSecty::write`
- Attribute secretary read function: *className*_`AttrSecty::fetch`
- Attribute secretary write function: *className*_`AttrSecty::store`
- Action secretary check function: *className*`ActionSecty::action`
- Action secretary perform function: *className*_`InstanceSecty::create_vote`
- Action secretary do not perform function: *className*_`AttributeSecty::destroy_vote`
- Dynamic Loader interface: *className*_`loader`

5.11.1 Example Generated Code in .cc File

The following generated code stub examples are based on the chai example object. A complete scenario for defining the chai object is provided in Section 5.10, “Object Development Scenario Using Chai Object.” The actual code that OCG generates can contain additional comments not reflected in this book.

Note – Throughout these examples and in any code generated by OCG, there are lines that are similar to this:

```
//********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] ************//
```

These lines indicate areas in the code where you can safely add your own code to the generated code to further customize object behaviors.

5.11.1.1 Generated Asynchronous Read Stub Function (FETCH)

Function:

```
chai_AttrSecty::fetch
```

Description:

This function is an asynchronous interface for reading attributes. For every attribute requested in a GET request, MIS Framework calls `fetch()` for that attribute, followed by a `read()` of the same attribute.

Arguments:

This function uses the following arguments:

- `ai` indicates the attribute to be read.
- `cb` identifies the callback routine passed by Framework.

Return Value:

This function returns the following values:

- OK if the attribute is fetched and read successfully.
- NOT_OK to indicate failure in fetching/reading attribute specified by `ai`.

Code Example:

```
Result
chai_AttrSecty::fetch(const AttrSectyInfo &ai, const Callback &cb)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::fetch",
   "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, const Callback &cb
= "
    "0x%lx", (void*)this, &ai, (void*)cb));

#ifdef ODT_EXTENDED
//********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] ************//
#endif
```

```
    TRY
    {
        // Fetch attribute specified by (ai)
        return subfetch(ai,cb);

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [FETCH BEHAVIOUR SPECIALIZATION INSERT ] **//
//***** $ODT_EXT_END   [FETCH BEHAVIOUR SPECIALIZATION INSERT ] **//
#endif

    }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

5.11.1.2 Generated Asynchronous Write Stub Function (STORE)

Function:

```
chai_AttrSecty::store
```

Description:

This function is an asynchronous interface for storing attributes. For every attribute requested in a SET request, MIS Framework calls `write()` for that attribute, followed by a `store()` of the same attribute.

Arguments:

This function uses the following arguments:

- `ai` indicates the attribute to be stored.
- `cb` identifies the callback routine passed by Framework

Return Value:

This function returns the following values:

- OK if the attribute is written and stored successfully.
- NOT_OK to indicate failure in writing/storing attribute specified by `ai`.

Code Example:

```
Result
chai_AttrSecty::store(const AttrSectyInfo &ai, const Callback &cb)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::store",
   "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, const Callback &cb
= "
    "0x%lx", (void*)this, &ai, (void*)cb));

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//*********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] ************//
#endif

    TRY
    {
        // store attribute to DATASTORAGE specified by (ai)
        return substore(ai,cb);

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [STORE BEHAVIOUR SPECIALIZATION INSERT ] **//
//***** $ODT_EXT_END   [STORE BEHAVIOUR SPECIALIZATION INSERT ] **//
#endif

    }
    BEGHANDLERS
    CATCHALL {
```

```
#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

5.11.1.3 Generated Synchronous Read Stub Function (READ)

Function:

```
chai_AttrSecty::read
```

Description:

This function is a synchronous interface for reading attributes. For every attribute requested in a GET request, MIS Framework calls `read()` for that attribute.

Arguments:

This function uses the following arguments:

- `ai` indicates the attribute to be read.
- `av` identifies the Asn1Value of the attribute read (output parameter).

Return Value:

This function returns the following values:

- OK if the attribute is read successfully.
- NOT_OK to indicate failure.

Code Example:

```
Result
chai_AttrSecty::read(const AttrSectyInfo &ai, Asn1Value &av)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::read",
    "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, Asn1Value &av = "
    "0x%lx", (void*)this, &ai, (void*)av));

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//*********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *************//
#endif

    TRY
    {
        int index = AttrSectyInfo2index(ai);

        // Validate passed attribute index
        if(!is_valid_index(index))
        {
            return NOT_OK;
        }

        // Read in-memory value of attribute specified by (ai)
        TRYRES (subread(ai,av));

        // Assign read attribute value
        index2lhsvalue(index) = av ;

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [READ BEHAVIOUR SPECIALIZATION INSERT] ****//
//***** $ODT_EXT_END   [READ BEHAVIOUR SPECIALIZATION INSERT] ****//
#endif
        return(OK);
    }
    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif
#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif


    }
    ENDHANDLERS
}
```

5.11.1.4 Generated Synchronous Write Stub Function (WRITE)

Function:

```
chai_AttrSecty::write
```

Description:

This function is a synchronous interface for writing attributes. For every attribute requested in a SET request, MIS Framework calls `write()` for that attribute.

Arguments:

This function uses the following arguments:

- `ai` indicates the attribute to be written.
- `av` identifies the Asn1Value of the attribute written (output parameter).

Return Value:

This function returns the following values:

- OK if the attribute is written successfully.
- NOT_OK to indicate failure.

Code Example:

```
Result
chai_AttrSecty::write(const AttrSectyInfo &ai,
                      const Asn1Value &av)
{
    TRACE(Tracer TR(chai_trace, "chai_AttrSecty::write",
    "this = 0x%lx, const AttrSectyInfo &ai = 0x%lx, const Asn1Value "
    "&av = 0x%lx", (void*)this, &ai, (void*)av));

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//*********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *************//
#endif

    TRY
    {
        int index = AttrSectyInfo2index(ai);

        // Validate passed attribute index
        if(!is_valid_index(index))
        {
            return NOT_OK;
        }

        // Assign written attribute value
        index2lhsvalue(index) = av ;

        // Execute Write thru
        TRYRES (subwrite(ai,av));

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [WRITE BEHAVIOUR SPECIALIZATION INSERT ] **//
//***** $ODT_EXT_END   [WRITE BEHAVIOUR SPECIALIZATION INSERT ] **//
#endif

        return(OK);
    }

    BEGHANDLERS
    CATCHALL {
```

```
#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

5.11.1.5 Generated Action Stub Function (ACTION)

Function:

```
chai_AttrSecty::action
```

Description:

This function is an asynchronous interface for handling CMIP ACTIONS. The CMIP actions handled are those specified in the Managed Object Class definition in the GDMO.

Arguments:

This function uses the following arguments:

- `ai` indicates the action.
- `av` indicates the Asn1Value specified by the user through the INFORMATION SYNTAX clause in the GDMO ACTION definition.
- `cb` identifies the callback routine passed by Framework to process `cb.exec` (CheckData) where CheckData contains the returned ACTION RESPONSE and ACTION RESULT.

Return Value:

This function returns no values aside from those passed back in the `cb` argument. CheckData return values are:

- Operation completed successfully
  - Set `cd.result` to CHECK_DONE.
  - Set `cd.rv` to encoded Asn1Value of ACTION RESPONSE.
- Operation terminated with an error
  - Set `cd.result` to CHECK_ERROR.
  - Set `cd.rv` to specific ACTION error or to NULL.

Code Example:

```
void
chai_ActionSecty::action(const ActionSectyInfo &ai,
                                  const Asn1Value &input,
                                  const Callback &cb)
{
    TRACE(Tracer TR(chai_trace, "chai_ActionSecty::action",
    "this = 0x%lx, const ActionSectyInfo &ai = 0x%lx, const Asn1Value "
    "&input = 0x%lx, const Callback &cb = 0x%lx",
    (void*)this, &ai, (void*)input, (void*)cb ));

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//*********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] ************//
#endif

    TRY
    {

        CheckData cd;             // CheckData (cd) - Action Response
value //
```

```
              switch(ai.local_value() )
              {
                  case IDX_brewChai:
#ifdef ODT_EXTENDED
//********* $ODT_EXT_START [ACTION IMPLEMENTATION INSERT] *******//
//********* $ODT_EXT_END   [ACTION IMPLEMENTATION INSERT] ********//
#endif
                      break;


              };

#ifdef ODT_DEFAULT
    // Default implementation (returns NULL Action Response & Success)
        cd.result = CheckData::CHECK_DONE;
        cb.exec(&cd);
#endif

        }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_EXTENDED
//****** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] ****//
//****** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] ****//
#endif

    }
    ENDHANDLERS
}
```

5.11.1.6 *Generated Instance Create Stub Function (CREATE)*

Function:

```
chai_AttrSecty::create_vote
```

Description:

This function lets you validate a CMIP CREATE request before the infrastructure creates the object instance. The function is passed a resolved attribute list and FDN, which you can validate before voting OK or NOT_OK.

Arguments:

This function uses the following arguments:

- `fdn` is the FDN (Fully-Distinguished Name) of the object instance to be created.
- `av` identifies the Asn1Value of the resolved Attribute List.

Return Value:

This function returns the following values:

- OK to go ahead and create the object instance.
- NOT_OK to not create the object instance.

Code Example:

```
Result
chai_InstanceSecty::create_vote(const Asn1Value &fdn,
                                        const Asn1Value &av)
{
    TRACE(Tracer TR(chai_trace, "chai_InstanceSecty::create",
    "this = 0x%lx, const Asn1Value &fdn = 0x%lx, const Asn1Value "
    "&av = 0x%lx", (void*)this, &ai, (void*)fdn, (void*)av ));

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//*********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] *************//
#endif

    TRY
    {
```

```
#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [CREATE VOTE SPECIALIZATION INSERT ] ******//
//***** $ODT_EXT_END   [CREATE VOTE SPECIALIZATION INSERT ] ******//
#endif

        return OK;
    }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

5.11.1.7 Generated Instance Destroy Stub Function (DELETE)

Function:

```
chai_AttrSecty::destroy_vote
```

Description:

This function lets you validate a CMIP DELETE request before the infrastructure deletes the object instance.

Arguments:

This function uses no arguments.

Return Value:

This function returns the following values:

- OK to delete the object instance.
- NOT_OK to not delete the object instance.

Code Example:

```
Result
chai_AttrSecty::destroy_vote()
{
    TRACE(Tracer TR(chai_trace, "chai_InstanceSecty::destroy_vote",
    "this = 0x%lx", (void*)this));

#ifdef ODT_EXTENDED
//************ $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//************ $ODT_EXT_END   [LOCAL VARIABLE INSERT] *************//
#endif

    TRY
    {

#ifdef ODT_EXTENDED
//****** $ODT_EXT_START [DELETE VOTE SPECIALIZATION INSERT ] *****//
//****** $ODT_EXT_END   [DELETE VOTE SPECIALIZATION INSERT ] *****//
#endif

        return OK;
    }

    BEGHANDLERS
    CATCHALL {

#ifdef ODT_DEFAULT
        return (NOT_OK);
#endif

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif
```

```
    }
    ENDHANDLERS
}
```

5.11.1.8 Generated Receive Event Stub Function (RECEIVE_EVENT)

Function:

```
chai_AttrSecty::receive_event
```

Description:

This function lets you receive events and notifications specified in `event_type` and `event_info`. You would provide specialized processing depending on `event_type` and `event_info` which have been received as the discriminatorConstruct specified in the object instance.

Note – This function is generated only if you specify the three discriminator attributes (DiscriminatorConstruct, OperationalState, and AdministrativeState) in your GDMO file and specify FILTER_ATTR: DiscriminatorConstruct in your configuration file.

Arguments:

This function uses the following arguments:

- `event_type` is the type of event or notification received
- `event_info` is the Event Info received

Return Value:

This function returns the following values:

- OK if the event or notification was processed successfully.
- NOT_OK to indicate failure in processing event or notification.

Code Example:

```
Result chai_AttrSecty::receive_event(
                    Asn1Value &event_type, Asn1Value &event_info)
{

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [LOCAL VARIABLE INSERT] ************//
//*********** $ODT_EXT_END   [LOCAL VARIABLE INSERT] ************//
#endif

    TRY
    {

#ifdef ODT_DEFAULT
        return OK;
#endif

#ifdef ODT_EXTENDED
//*********** $ODT_EXT_START [EVENT PROCESSING INSERT] **********//
//*********** $ODT_EXT_END   [EVENT PROCESSING INSERT] ***********//
#endif

    }
    BEGHANDLERS
    CATCHALL {

        return (NOT_OK);

#ifdef ODT_EXTENDED
//***** $ODT_EXT_START [ EXCEPTION HANDLING CATCHALL INSERT] *****//
//***** $ODT_EXT_END   [ EXCEPTION HANDLING CATCHALL INSERT] *****//
#endif

    }
    ENDHANDLERS
}
```

5.11.2 Example Generated Code in `.hh` File

The `chai_user.hh` file contains class definitions required for implementing behaviors. The object framework uses the methods defined in this file to perform CMIS operations on the managed object instance. The following secretaries are defined in this file:

- `chai_AttrSecty`
- `chai_ActionSecty`
- `chai_InstanceSecty`

5.11.2.1 Generated Object Definitions

```
//-------------------------------------------------------------//
//          OBJECT ATTRIBUTES ENUMERATION                       //
//-------------------------------------------------------------//
enum chai_ATTR_INDEX{
    IDX_chaiKettleNumber,
    IDX_chaiBlend,
    IDX_chaiReady,
    NUM_chai_ATTR
};

//-------------------------------------------------------------//
//            OBJECT ACTIONS ENUMERATION                        //
//-------------------------------------------------------------//
enum chai_ACTION_INDEX {
    IDX_brewChai,
    NUM_chai_ACTION
};
```

5.11.2.2 Generated OIDs

```
//-------------------------------------------------------------//
//             ATTRIBUTE OBJECT IDENTIFIERS (OID)               //
//-------------------------------------------------------------//
#define OID_chai_chaiKettleNumber
"1.3.6.1.4.1.42.2.2.2.1.96.7.1"
#define OID_chai_chaiBlend       "1.3.6.1.4.1.42.2.2.2.1.96.7.2"
#define OID_chai_chaiReady       "1.3.6.1.4.1.42.2.2.2.1.96.7.3"

//-------------------------------------------------------------//
//              ACTION OBJECT INDENTIFIERS (OID)                //
//-------------------------------------------------------------//
#define OID_chai_brewChai        "1.3.6.1.4.1.42.2.2.2.1.96.9.1"

//-------------------------------------------------------------//
//            NAME BINDING OBJECT IDENTIFIERS (OID)             //
//-------------------------------------------------------------//
#define OID_chai_chai_system
"1.3.6.1.4.1.42.2.2.2.1.96.6.1"
#define OID_chai_chai_chai       "1.3.6.1.4.1.42.2.2.2.1.96.6.2"
```

5.11.2.3 Attribute Class Definition

```
class chai_AttrSecty: public AttrSecty
{
private:
protected:
        // Constructor ODT RESERVED
        chai_AttrSecty(ObjMethMOI &m, const AttrSecty *sp,
                         const AttrSectyTmpl &t);
       // Destructor ODT RESERVED
        ~chai_AttrSecty();
public:
        // Local storage for MOI's Attributes
        Asn1Value chaiKettleNumber;
        Asn1Value chaiBlend;
        Asn1Value chaiReady;

        // User Methods
        Result  read(const AttrSectyInfo &ai, Asn1Value &av);
        Result  write(const AttrSectyInfo &ai, const Asn1Value &av);
        Result  fetch(const AttrSectyInfo &ai, const Callback &cb);
        Result  store(const AttrSectyInfo &ai, const Callback &cb);
        Result  destroy_vote();
        Result  receive_event(Asn1Value &event_type, Asn1Value
&event_info);

      void   action(const ActionSectyInfo &ai, const Asn1Value &input,
                       const Callback &cb);

        static  int     AttrSectyInfo2index(const AttrSectyInfo &ai);
        Result  index2AttrSectyInfo(int index, AttrSectyInfo &ai);
        Result  index2ActionSectyInfo(int index, ActionSectyInfo &ai);
```

```
         // ODT RESERVED METHODS
        static  AttrSecty *new_secty(ObjMethMOI &m, const AttrSecty *sp,
                                     const AttrSectyTmpl &t);
        Asn1Value &index2lhsvalue(int attrindex);
        Result  delete_prepare(DeleteType type);
        static  Result  is_valid_index(int index);
        static  Oid     index2Oid(int attrindex);
        chai_ActionSecty *get_actionsecty()
        {
            chai_MOI *mm = (chai_MOI *)&moi;
            chai_ActionSecty *actionsecty = mm->actionsecty;
            return actionsecty;
        }

#ifdef ODT_EXTENDED
//******* $ODT_EXT_START [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
//******* $ODT_EXT_END   [MEMBER FUNCTION/PROTOTYPE INSERT] ******//
#endif

};
```

5.11.2.4 Action Class Definition

```
class chai_ActionSecty: public ActionSecty
{
protected:
        chai_ActionSecty(ObjMethMOI &m, const ActionSecty *sp,
                     const ActionSectyTmpl &t) ;


public:
        // USER METHODS
      void   action(const ActionSectyInfo &ai, const Asn1Value &input,
                       const Callback &cb);

        // INTRA-OBJECT CONV. METHODS
        Result  read(const AttrSectyInfo &ai, Asn1Value &av);
        Result  write(const AttrSectyInfo &ai, const Asn1Value &av);
        Result  fetch(const AttrSectyInfo &ai, const Callback &cb);
        Result  store(const AttrSectyInfo &ai, const Callback &cb);
        Result    index2AttrSectyInfo(int index, AttrSectyInfo &ai);
       Result    index2ActionSectyInfo(int index, ActionSectyInfo &ai);
```

```
        // ODT RESERVED METHODS
     static ActionSecty *new_secty(ObjMethMOI &m, const ActionSecty *sp,
                                  const ActionSectyTmpl &t)
        {
               return (ActionSecty *)new chai_ActionSecty(m, sp, t);
        }
        static     Result    is_valid_index(int index);
        static     Oid       index2Oid(int attrindex);
        chai_AttrSecty *get_attrsecty()
        {
            chai_MOI *mm = (chai_MOI *)&moi;
            chai_AttrSecty *attrsecty = mm->attrsecty;
            return attrsecty;
        }
        virtual void check(const ActionSectyInfo &ai,
                           const ActionInfo *t,
                           const Asn1Value &av, const Callback &cb);

#ifdef ODT_EXTENDED
//******* $ODT_EXT_START [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
//******* $ODT_EXT_END   [MEMBER FUNCTION/PROTOTYPE INSERT] *****//
#endif

};
```

Debugging EM Code 6

6.1 Introduction

This chapter provides information on how to debug code that uses the EM PMI to communicate with the MIS. It provides the following information:

- Basic EM Architecture
- API Debugging Processes and Procedures
- Using em_debug

6.2 Basic EM Architecture

To easily solve problems associated with how EM interacts with various applications, you need to understand the general approach to how EM works. EM generally consists of several major components:

MIS (Management Information Server)

The MIS is the central process which maintains the repository of network management information and services requests from the client applications about that data.

Protocol adapters

Protocol adapters are the mechanisms that allow proprietary and legacy protocols to integrate with Solstice EM.

Management applications and utilities

Solstice EM provides several applications and utilities that allow network administrators to start using EM to manage their networks.

Application programming interfaces

Solstice EM provides various APIs for you to use when writing applications that interface with EM. For detailed information on the classes, functions, and syntax of the APIs, see the Solstice Enterprise Manager API Syntax Manual.

6.2.1 How It Fits Together

Figure 6-1 illustrates how the Solstice EM pieces fit together to create a network administration environment.

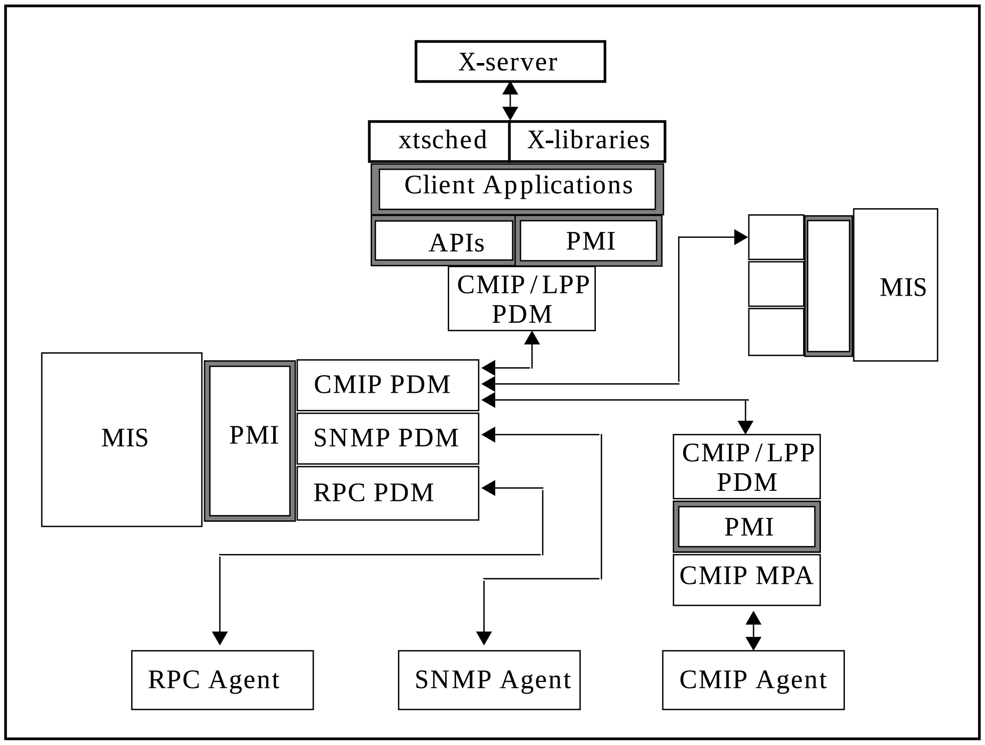


Figure 6-1 MIS Architecture

The items shown in Figure 6-1 with a grayed border indicate areas where you control the interactions. These are also the areas where you may experience problems that you can easily solve.

When you create an application that works on the EM platform, the application basically does two things:

- It connects to the MIS.
- It sends and receives information to and from the MIS.

To put it another way, MIS functions as an agent (actually sits atop real agents, e.g., CMIP and SNMP); Your applications function as a manager.

6.2.2 Tools to Help Identify Problems

EM includes several tools that can help you identify problems in how your applications interact with EM:

- em_debug provides a lot of information about what is going on in the MIS.
- Object Editor/Browser (OBED) lets you view objects in the MIS.
- CMIPtrace is not part of the EM package, but should be part of the standard CMIP environment. CMIPtrace lets you track the network protocol interactions.
- Other EM tools, such as the Object Configuration Tool (OCT) and the Data Viewer provide information about the state of various managed objects in your network. This can sometimes help you determine where a problem is.

6.3 API Debugging Processes and Procedures

If you are new to the arena of application development with Solstice EM, here are some hints to help you determine where your code problems lay:

1. **Verify that your GDMO and/or ASN.1 calls are syntactically valid. For more information about GDMO syntax, see *ITU X.722 ISO/IEC 10165-4 Guidelines for the Definition of Managed Objects (GDMO)*. For more information about ASN.1, see *ITU X.208 ISO/IEC 8824 Specification of Abstract Syntax Notation One*.**

2. **Verify that your code syntax (C++) is correct. In other words, determine whether the problem is in the C++ code or in an API call.**

3. **Verify that you are using the correct C++ compiler (DevPro C++, included with the Solstice Enterprise Manager Developer's Toolkit).**

4. **Compile your application using the -g option with em_debug turned on.**

5. **Use the debugger to step through your code and identify values to narrow down the cause of your problem.**

6. **Identify the specific API with which you are experiencing difficulty and verify that the syntax of your API calls is correct. For information on specific object classes and function calls, see the *Solstice Enterprise Manager API Syntax Manual*.**

6.4 Using em_debug

6.4.1 Overview

The `em_debug` program is a utility you use to communicate with the MIS, to turn on/off the MIS's built-in debug features while MIS is running.

6.4.2 Command Line and Options

The em_debug program uses the following command line format:

```
canton%  em_debug [-oamrmt_debug] -h server -c "[on|off] debug agent"
```

Output from `em_debug` is directed to the window from which you invoked `em_services`. The optional `-oamrmt_debug` flag allows you to receive messages from remote MISes.

The following useful debug objects are defined, as shown in Table 6-1:

Table 6-1 em_debug Debug Objects

Object	Description
`actmsg_debug`	Print PMI Action request and response messages.
`oammsg_debug`	Print PMI non-Action messages, that is, Get, Set, Create, Delete, Cancel-Get and Event requests and responses.
`asn_debug`	Print debug messages on ASN1 modules processing.
`asn_info`	Print detailed debug messages on ASN1 modules processing.
`cmip_debug`	Print CMOT debug messages, including `acse`, `rose`, `lpp` and `tcp`.
`cmip_info`	Print detailed debug messages on CMOT.
`snmp_debug`	Print SNMP PDM debug messages.

Table 6-1 em_debug Debug Objects *(Continued)*

Object	Description
`snmp_info`	Print detailed SNMP PDM debug messages.
`rpc_debug`	Print RPC PDM debug messages.
`rpc_info`	Print detailed RPC PDM messages.

For example, to see all messages transmitted between applications and the MIS, assuming the MIS process, `em_mis`, is running on the machine `canton`, enter:

```
canton% em_debug -h nasdaq -c "on oammsg_debug"
canton% em_debug -h nasdaq -c "on actmsg_debug"
```

Please note that internally the MIS also generates messages and these messages are also displayed, in addition to the messages transmitted between applications and the MIS.

6.4.3 How to use em_debug

To use em_debug, you follow basically follow these steps:

- Turn on em_debug for the specific messages you want to look at
- Initiate trace
- Use a combination of em_debug output, ASN.1 definitions, and OBED to trace what the code is doing

6.4.4 Example Analysis Using em_debug

Appendix C, “em_debug Analysis,” provides a walkthrough of a debugging session. Basically, this walkthrough goes through a PMI platform.connect() call. It follows six functions:

- M-CREATE emApplicationInstance object
- M-SET attribute administrativeState to unlocked
- Receive M-EVENT-REPORT for successful objectCreation event
- Receive M-EVENT-REPORT for successful attributeValueChange event

- M-GET request of the attribute administrativeState of the log object instance
- Before the PMI client application exits, the PMI call platform.disconnect()results in a M-DELETE request to the MIS to delete the corresponding emApplicationInstance object

Scenarios 7

This chapter describes four basic scenarios that deal with problems or situations you might encounter when managing your own network:

- Adding and managing a new class of device
- Using PMI client applications for specialized presentation of information
- Sharing management information among PMI client applications
- Using multiple MISs

These scenarios parallel the design considerations covered in the “Developing EM Solutions” chapter. Each of the scenarios include one or more example solutions for the problem that the scenario describes. The solutions presented for each scenario can be configured or developed using the Solstice EM subcomponents described in the “Enterprise Manager Concepts” chapter. The basic scenarios presented here are general in nature and do not represent the only types of problems you can address using the Solstice EM product. The scenarios presented here simply address some of the problem areas you are likely to deal with when using Solstice EM to help solve your network management problems.

Detailed instructions for setting up Solstice EM application solutions are included as part of the Solstice EM software release. Each scenario described here includes one or more examples. Each example is contained in an example directory and is described in detail by a README file. The example directories can be found under the following directory:

```
$EM_HOME/src/scenario
```

7.1 Scenario 1: Adding and Managing a New Class of Device

A common scenario that you are likely to encounter when managing your network is configuring the Solstice EM product to manage new devices that are added to your network. This solutions provided in this first scenario address the configuration of the Solstice EM product to manage of a new CMIP-manageable device.

If you add a new class of device to your network, you will need to load information about the new device type into the Solstice EM MIS. If the device is a new class of a CMIP-manageable device you will need to load a GDMO definition for the device into the MIS. If the device is a new class of an SNMP-manageable device or is represented by a SunNet Manager object, you will need to convert an SNMP MIB or SunNet Manager schema file into GDMO and then load that definition into the MIS. In addition to loading the device definition, you may also need to perform some additional steps that will help you manage the device.

7.1.1 Scenario Problem Description and Requirements

This first scenario focuses on management of new types of CMIP devices that you add to your network. The example solutions presented for this scenario are configuration activities and do not require the development of software. There are two examples included in this scenario.

7.1.1.1 Example 1 Problem Description and Requirements

Example 1 addresses the following situation:

- You have a new class of CMIP device that you want to add to your network. The GDMO definition for the device will be provided as part of this example.

The fact that it is a new class of device means that either there are no other devices in your network that are managed using the GDMO definition used by this device, or that the definition has not been pre-configured or previously loaded into the Solstice EM MIS.

For this example, a new managed object class, mySystem, will be derived from the system managed object class.

- The mySystem managed object class defines a new type of notification

  The GDMO definition derived includes a new notification type.

  The default log will be extended to log the new notification type, which will also allow the notification type to be used by the NerveCenter.

  A new log record class will be defined as part of the GDMO definition to store information from the new notification. The mapping between the notification and the event log record will also be defined.

- The device can be represented in the Viewer using an existing icon in the Viewer's Object Palette.

7.1.1.2 Example 2 Problem Description and Requirements

Example 2 addresses the following situations:

- You want to use NerveCenter and the Alarm Manager application to help manage the new device. In particular, you want to use NerveCenter to track the new notification defined for the device.

  A NerveCenter Request will be developed to process the new notification defined for the mySystem class.

  The device can operate using the default polling rates and severities defined for the NerveCenter.

- The system object class emits a number of the notifications that are pre-loaded into the Solstice EM MIS. These notifications are also valid notifications for the mySystem managed object class, which is derived from system.

  The system managed object class uses a number of DMI defined notifications allowing it to operate with default log and log record definitions. These notifications are also valid for the mySystem managed object class, but will not be used by the Request Template.

Note – The polling rates for a device can be tuned, possibly providing more efficient management of the device. Polling is not used to drive the Request Template developed here, so poll tuning is not required. In general, most devices should be able to work fine with the range of default values provided by the Request Designer application.

7.1.2 Solution Design Summary

Chapter 4, "Developing EM Solutions," lists a number of design questions and decisions you may want to consider when you add a new device. For most of these decisions, or if you are unsure about some of the answers associated with the design questions you should start by using defaults and pre-configured subcomponents. Later, as you gain more experience with the device and with the Solstice EM product, you can go back if necessary and tune the configuration subcomponents to better manage the device. The examples provided in this scenario use a number of defaults, which simplifies the configuration of a new device. The next two subsections list the subcomponents you will need to work with or develop in order to provide a solution for this type of scenario. The third subsection provides a diagram showing the subcomponents developed for this scenario.

7.1.2.1 Solstice EM Subcomponents Developed or Configured for this Scenario

Example 1 requires the following Solstice EM subcomponents:

- the mySystem GDMO managed object class definition must be created and loaded into the MIS. The GDMO module includes a new notification type and a new log record managed object class definition that matches the new notification type.
- the default log discriminator must be extended to allow logging of the new notification type;
- a new log record object class must be created to store information from the new notification
- a mapping must be defined that specifies that the new notification is to be logged using the new event log record class.

Example 2 requires the following Solstice EM subcomponents:

- a NerveCenter Request template must be created and configured to help manage instances of the mySystem class.

7.1.2.2 *Additional Solstice EM Components Used*

You may also need or want to use the following Solstice EM utilities and components to develop your solution:

- You will need to use the following Solstice EM commands to add the GDMO definitions to the MIS:

  The GDMO compiler: em_gdmo

  The ASN.1 compiler: em_asn1

Note – If you are loading an SNMP MIB definition or SNM schema file you will also need to use one of the following components to produce a GDMO definition: the SNMP Concise MIB to GDMO conversion utility: em_cmib2gdmo; or the SunNet Manager schema file to GDMO conversion utility: em_snm2mib

- You will need to use one of the following Solstice EM commands to create an agent role managed object class in the MIS for the new event log record.

  The object behavior link commands: em_compose_oc or em_compose_poc

Note – You do not need to use the em_load_name_bindings command to load name bindings for the new event log record. The eventLogRecord base class that the new event log record is derived from defines name bindings that also apply to its derived classes.

- You will need to use the Request Designer to develop the NerveCenter Request Template for Example 2.
- You will need to use the Object Editor to add the notification-to-event log record mapping to the event2ObjectClass object.

7.1.2.3 *Overview of Examples*

Figure 7-1 shows the subcomponents needed for Example 1 of the scenario.

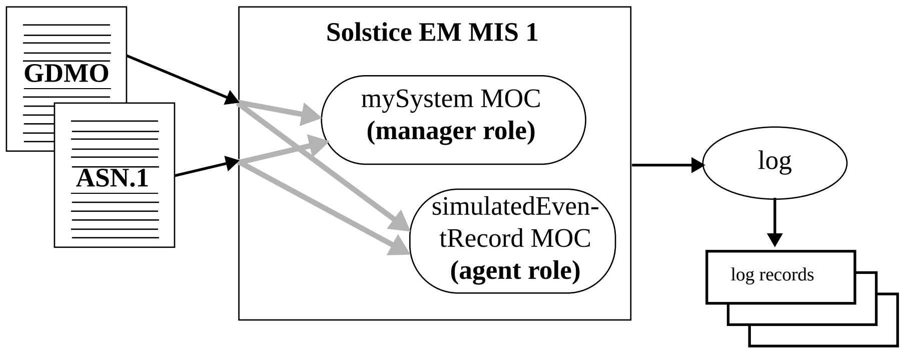


Figure 7-1 Scenario One, Example 1: Managing a New Device

Note – Providing a manager role object is not sufficient in order to access an instance of a managed object class. An agent role object must also exist as a local object or as a remote object in your network. This means that you must have a device in your network that contains an instance of the managed object class or you must create an instance of the managed object class in your network. Typically any CMIP agent or Solstice EM MIS in your network will be represented by an instance of the system managed object class. The creation of an instance of the mySystem managed object class (i.e., an agent role object) will be addressed in Example 5 of Scenario Two.

Figure 7-2 shows the subcomponents that you will need to configure for Example 2 of this scenario.

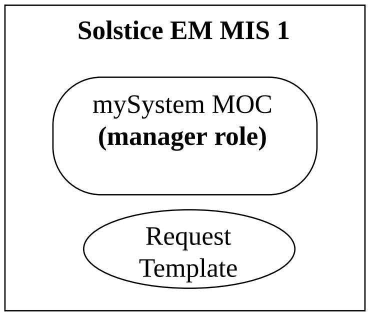


Figure 7-2 Scenario One, Example 2: Request Template for Managing a Device

7.1.3 Examples

The examples developed for Scenario One can be found in the following Solstice EM directories:

```
$EM_HOME/src/scenario/example1
$EM_HOME/src/scenario/example2
```

7.2 Scenario 2: Using PMI Client Applications To Present Information

PMI client applications can be used to present management information in a specialized way. This scenario provides examples of two PMI client applications that are used to present information about the mySystem managed object class used in Scenario One. The first example will use the PMI C++ classes for Platform and Image and display information using standard out (i.e., terminal-based output). The second version of the PMI client application will extend the first example to use X and Motif to display information and will also use the xtsched subcomponent (and also provides an option to use the sched subcomponent for comparison). A third example is provided that uses the GDMO and ASN.1 definitions from Example 1 to create an agent role managed object that the two PMI client applications can examine.

7.2.1 Scenario Problem Description and Requirements

In order to effectively manage your network, you may need to develop some applications to display information in special ways. The basic requirement that the solution presented for scenario addresses is as follows:

- You need to present information about instances of the mySystem managed object class in a specialized fashion that is not possible to achieve using existing Solstice EM components, such as the Viewer or the Alarm Manager. Two versions of information presentation are needed: terminal output; and GUI based presentation.

7.2.2 Solution Design Summary

The examples provided in this scenario will operate with the mySystem managed object class developed for Example 1 of Scenario One. The PMI client applications will display information about instances of the mySystem managed object class. The mySystem managed object class definition will be used to produce a local managed object that the PMI clients can examine.

The next two subsections list the subcomponents you will need to work with or develop to provide a solution for this scenario. The third subsection provides a diagram showing the subcomponents developed as part of the solution.

7.2.2.1 Solstice EM Subcomponents Developed for this Scenario

The following Solstice EM subcomponents were developed for Example 3:

- A PMI client application, which uses:
  - Platform
  - Image
- A client application makefile

The following Solstice EM subcomponents were developed for Example 4:

- The PMI client application developed for Example 3, but extended to use an X-windows-based display
- A client application makefile which can be used to generates an executable file that uses xtsched
- A client application makefile which can be used to generate an executable file that uses sched

No new subcomponents were developed for Example 5.

7.2.2.2 *Additional Solstice EM Components Used*

The example PMI client applications rely on the GDMO and ASN.1 modules developed for Example 1 of Scenario One. You will need to use the following Solstice EM commands to create the local (agent role) managed object for Example 5:

- The GDMO compiler: em_gdmo
- The ASN.1 compiler: em_asn1
- The object behavior link command: em_compose_oc or em_compose_poc
- The load name binding command: em_load_name_bindings

7.2.2.3 *Overview of Examples*

Figure 7-3 shows the example client applications and an instance of the local object class (agent role) developed for this scenario.

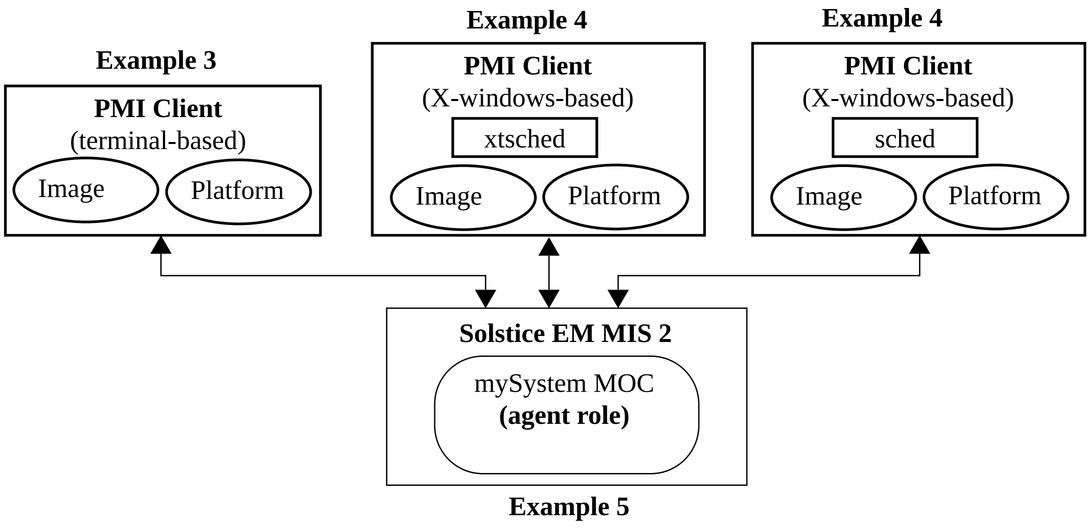


Figure 7-3 Scenario Two: Information Presentation

7.2.3 Examples

The examples developed for Scenario Two can be found in the following Solstice EM directories:

```
$EM_HOME/src/scenario/example3
$EM_HOME/src/scenario/example4
$EM_HOME/src/scenario/example5
```

7.3 Scenario 3: Sharing Information Among PMI Client Applications

Generally when you develop a management application you would like the application to be a distributed application that is coordinated with other applications that you may be running. This scenario shows how local managed objects can be used to share and coordinate management information between PMI client applications.

Sharing management information among PMI clients describes how managed object classes with repository behavior can be used share information among different client applications. This type of solution requires the development of a PMI client application as well as the development of a GDMO managed object class that uses default repository behaviors.

7.3.1 Scenario Problem Description and Requirements

7.3.1.1 Example 6 Requirements

There are a number of requirements for this scenario:

- You need to present information in a specialized fashion that is not possible to achieve using existing Solstice EM components, such as the Viewer or the Alarm Manager. The presentation of information could include specialized presentation windows, GUI-based device front ends, or terminal output.
- You want to provide shared access to MIS information from multiple PMI client applications.
- You want a central repository for at least part of the information processed by the PMI clients.
- You want automated update of information across PMI client applications when instance information is modified in the MIS.

- You want the PMI client applications to be notified when a particular type of notification occurs.

7.3.2 Solution Design Summary

The example provided in this scenario demonstrates how the PMI and local managed objects can be used to pass information between PMI client applications and how this can be done asynchronously. The agent role managed object class configured for Example 5 of Scenario 2 will be used as the central repository in the MIS for sharing and coordinating information. The PMI client applications from Scenario Two will be extended to asynchronously track information changes in the agent role object.

The next two subsections list the subcomponents you will need to work with or develop to provide a solution for this scenario. The third subsection provides a diagram showing the subcomponents developed as part of the solution.

7.3.2.1 Example Subcomponents Developed or Configured for this Scenario

The following Solstice EM subcomponents were developed for Example 6:

- The PMI client application from Example 3 was extended to: track changes in the attributes of an instance of the mySystem object; automatically re-display the instance data when an attribute value changes; and indicate when a notification is generated by an instance of the mySystem class and display the information contained in the notification.
- The PMI client application from Example 4 was extended to: track changes in the attributes of an instance of the mySystem object; automatically update the PMI client Main Window when an attribute value changes; and indicate when a notification is generated by an instance of the mySystem class the type of the notification. This example only uses xtsched; the code related to sched has been removed.
- The PMI client application `simulate_event` was developed to generate the simulatedEvent notification defined for the mySystem object class.
- A makefile for the client applications and the simulate_event program.

7.3.2.2 Additional Solstice EM Components Used

The example PMI client applications rely on and use the local managed object (agent role) developed for Example 5 of Scenario Two.

7.3.2.3 Overview of Example

Figure 7-4 shows the information sharing subcomponents used in Example 6. All components used in this example were previously developed for use in one or more of the five preceding examples.

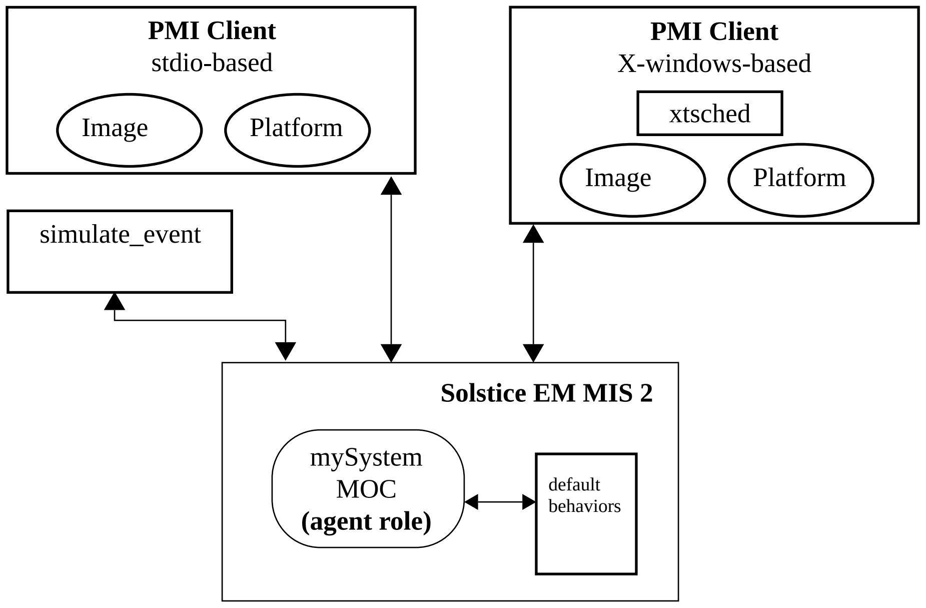


Figure 7-4 Scenario Three, Example 6: Sharing Information

7.3.3 Examples

The examples developed for scenario three can be found in the following Solstice EM directory:

```
$EM_HOME/src/scenario/example6
```

7.4 Scenario 4: Using Multiple MISs

If you are managing a large network, you may need more then one MIS to effectively manage the network. In order to have multiple MISs work together, you will need to define EFDs to pass information between MISs and configure each MIS to allow transparent access to the managed objects which may be controlled by another MIS. A *number of the items presented in this scenario also apply to solutions using a single MIS.* The section includes an example that shows both a single-MIS and multi-MIS application solution.

This section describes some basic mechanisms that can be set up to allow multiple MISs to be set up to work together to manage a network. The configuration steps for this type of solution do not require the development of software.

This class of solution requires that you have a GDMO Managed Object Class definition for the other MIS or other CMIP manager that you will be working with. In this example, the DMI system managed object class is used. The system class will normally be used for this purpose.

The solution also demonstrates how a local managed object can be used to simulate a CMIP agent.

7.4.1 Scenario Problem Description and Requirements

In order to manage large networks you may need to use more than one Solstice EM MIS. The basic requirements for this scenario addresses are as follows:

- You have a large network and want to use two MISs to manage the network.
- Each MIS manages a number of devices with no overlap of devices or overlap of the containment hierarchy (i.e., a device managed by one MIS, does not contain [in the MIT] a device managed by the other MIS).
- Each MIS is represented by the DMI system object. The DMI system object is loaded the first time you initialize the Solstice EM MIS (this happens automatically when the product is installed) and does not need to have its GDMO description loaded again.

7.4.2 Solution Design Summary

Three examples are provided to illustrate this class of solution. Example 7 shows how to set up an EFD on an MIS in order to forward events to another MIS. Example 8 shows how to configure a cmipAgent object on each MIS to allow MIS-to-MIS communication. Example 9 is a comprehensive multi-system solution that uses most of the components developed for each of the previous scenarios. Example 9 also effectively shows how a local managed object can be used to simulate a CMIP agent.

The next two subsections list the subcomponents you will need to work with or develop to develop a solution for this scenario. The third subsection provides diagrams showing how the subcomponents developed or configured for each example.

7.4.2.1 *Example Subcomponents Developed or Configured for this Scenario*

For Example 7, you will need to use the provided file or create a file containing a description for two event forwarding discriminators and then create the discriminators using the Solstice EM `em_objop` command. The first EFD will forward simulatedEvent notifications. The second EFD will forward attributeValueChange notifications that are generated by any instance of the mySystem object class.

For Example 8, you will need to use the Solstice EM command cm_cmipconfig to create the cmipAgent object needed for MIS-to-MIS communication.

No new subcomponents were developed for Example 9, although you will need to use the Solstice EM Viewer to activate the Request Template developed in Example 2 of Scenario One.

7.4.2.2 *Additional Solstice EM Components Used*

Example 7 uses the local object (agent role) created for Scenario Two, Example 5. You will also need to the Solstice EM Object Editor to verify proper operation of the EFD.

Example 8 does not rely on components from previous examples, but does use the Solstice EM Object Editor to verify proper operation of MIS-to-MIS communication.

Example 9 uses most of the subcomponents developed for each of the previous scenarios and examples. It also uses the Solstice EM Viewer, Object Editor, Log Manager and Alarm Manager applications to view information. Both single-MIS and multi-MIS solutions are shown for Example 9.

You might also use the following Solstice EM utilities and components to develop your solution:

- You might want to modify the `init_user` template to add a new object to the Viewer’s Object Palette to represent each MIS in the Viewer. The steps to add an object to the Palette are described in the *Solstice Enterprise Manager Administration Guide*.
- You can also use the following applications to display information about objects contained by either MIS:

- Solstice EM Viewer
- Solstice EM Data Viewer
- Solstice EM Alarm Manager
- Solstice EM Object Editor

7.4.3 Overview of Examples

Figure 7-5 shows the event forwarding discriminators and other components that are used for Example 7.

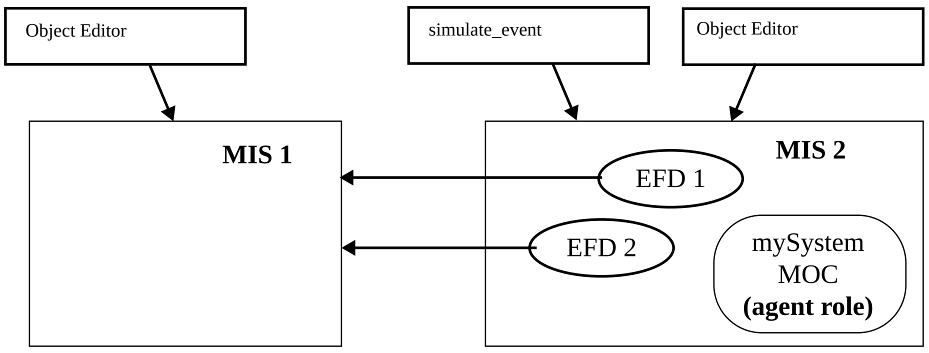


Figure 7-5 Scenario Four, Example 7: Event Forwarding

Figure 7-6 shows the cmipAgent objects and other components used for Example 8. Both Object Editors can access objects on the remote MIS.

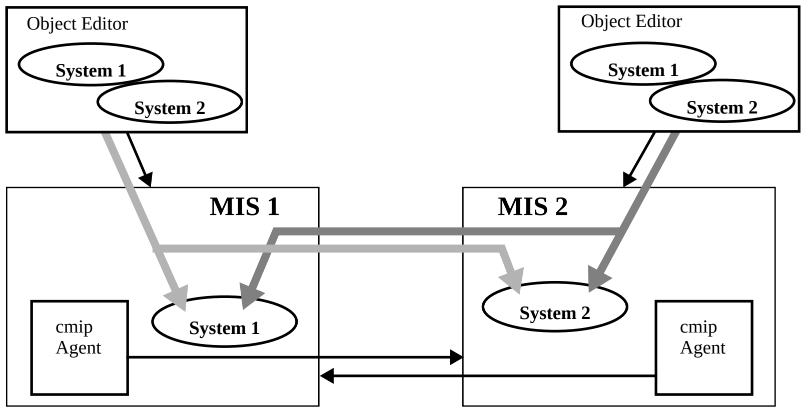


Figure 7-6 Scenario Four, Example 8: MIS-to-MIS Communication

The subcomponents developed for the most of the examples can be put together in a system for testing and operation purposes. Figure 7-7 shows a number of the subcomponents used by Example 9. *It does not show the Solstice EM applications that can also be used as part of this scenario*. Refer to the README file in the Example 9 directory for more information about setting up this scenario solution.

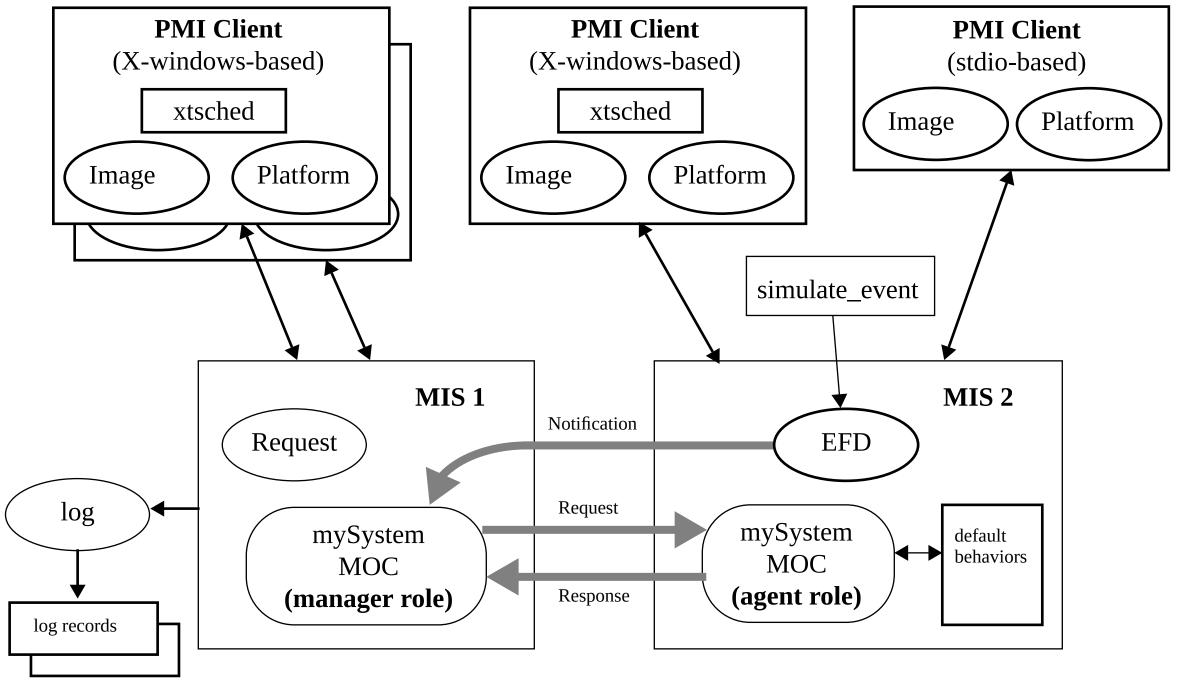


Figure 7-7 Scenario Four, Example 9: Multi-system Overview

The components and subcomponents can also be put together on a single workstation to form a single system solution. Figure 7-8 illustrates a single-system solution for Scenario Four.

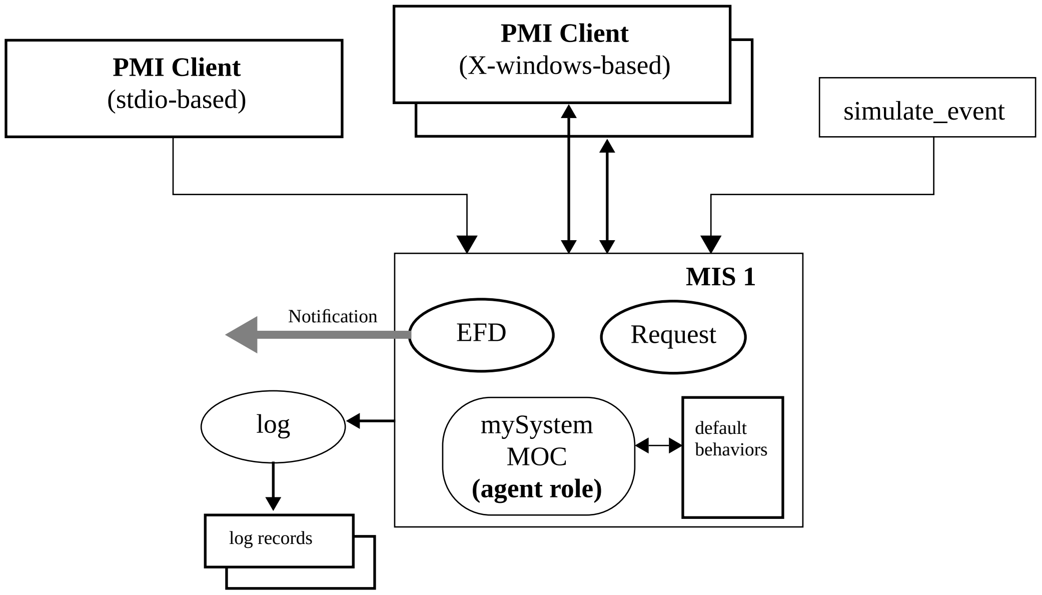


Figure 7-8 Scenario Four, Example 9: Single-system Solution

Figure 7-9 shows the subcomponents from the preceding figures that simulate a CMIP agent and shows the event forwarding discriminator (EFD); the agent role managed object, including the default behaviors that the managed object is linked to (in order create a local object); and also shows the PMI client application used to generate an event report. These subcomponents effectively simulate a CMIP agent. The MIS used in Example 5 includes a subset of these subcomponents and also simulates an agent, but it is not as comprehensive a simulation as the combination of subcomponents shown below.

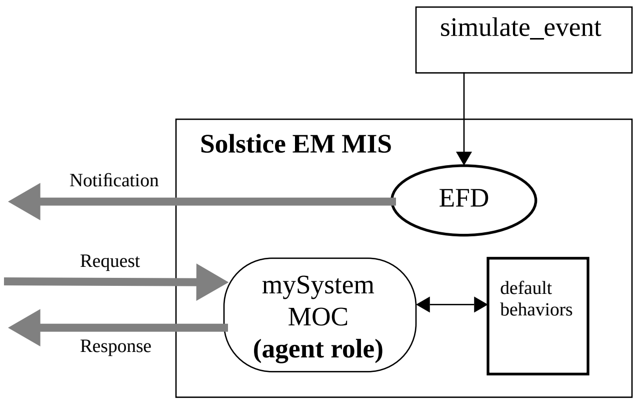


Figure 7-9 Scenario Four, Example 9: CMIP Agent Simulation

7.4.4 Examples

The examples developed for Scenario Four can be found in the following Solstice EM directories:

```
$EM_HOME/src/scenario/example7
$EM_HOME/src/scenario/example8
$EM_HOME/src/scenario/example9
```

API Examples 8

This chapter presents several examples that use the Solstice EM Application Programming Interfaces (APIs). These examples demonstrate use of the protocol-independent functions of the APIs to access and manipulate information about various managed objects known to the Solstice EM MIS.

Because the examples are shipped as part of the product, complete source code for all examples is not included in this chapter. Only one example, *em_cmipconfig Example*, is described in detail.

Note – The names of the directories in which these examples exist are all relative to the `$EM_HOME` directory, where `$EM_HOME` is the directory in which EM is installed and defaults to `/opt/SUNWconn/em`.

For information on the specific syntax of the API classes and functions, see the *Solstice Enterprise Manager API Syntax Manual*.

8.1 Compiling the Examples

The following command illustrates (using the `create.cc` example program) how to compile the examples in this chapter.

```
host_name% CC -I /opt/SUNWconn/em/include -I
/opt/SUNWconn/em/include/pmi -DSYSV -DTESTPGM -DNO_TRACE -noex -g -
L/opt/SUNWconn/em/lib -lpmi -lnsl -lsocket -lgen -o create create.cc
```

You must be using the 4.0.1 version of the SunSoft C++ compiler. To verify the compiler version, enter the following command:

```
hostname% CC -V
```

The output from this command has the following format (dates vary):

```
CC: SC 4.0.1  18 Oct 1995
Usage: CC [ options ] files.  Use 'CC -flags' for details
```

8.2 `em_cmipconfig` Example

`em_cmipconfig` is an example GUI program that uses many aspects of the high-level PMI to configure MIS-to-MIS communication. For a full description of these various areas of high-level PMI functionality, refer to the *Solstice Enterprise Manager API Syntax Manual*. The source code for `em_cmipconfig` is located in the `$EM_HOME/src/pmi_hi/cmipconfig` directory.

8.2.1 API Include Files

All applications that use the Solstice EM APIs must include the header file `hi.hh`. This file is located in the `$EM_HOME/include/pmi` directory. Include the following line near the beginning of your program:

```
#include <pmi/hi.hh>
```

Add the option `-I/opt/SUNWconn/em/include` on the compile line.

8.2.2 Using the `Platform` Class

Every EM application uses the `Platform` class to gain access to an MIS. To use the `Platform` class, define a variable to represent a connection to an MIS:

```
Platform plat;// represents a connection to an MIS
```

Initialize the `Platform` using the platform type `duEM`. This is the *only* supported type of `Platform`. This creates an instance of a connection to one MIS. One instance must be created for each MIS you wish to connect to.

```
plat = Platform(duEM);
```

No connection exists at this point.

Now establish the connection to the MIS:

```
if (!plat.connect(loc, "em_cmipconfig")) {
    cout << "MIS Connection to " << loc << " failed... Exiting." << endl;
    cout << plat.get_error_string() << endl;
    exit(0);
}
```

If the `default_platform` was not previously set, then it is set to `plat`. `loc` specifies the hostname where the MIS resides. The second parameter is the name of the application.

An object in the MIT gets created referring to this application with the attribute:

```
emApplicationType="em_cmipconfig"
```

For example:

```
/systemId=name:"pez"/subsystemId="EM-MIS"/emApplicationID=20
```

8.2.3 Registering for Events

Most EM applications will need to register for events. em_cmipconfig registers for the DISCONNECTED event. This allows the program to exit gracefully when the MIS disconnects. When the specified event happens, the registered callback function will be called. In the following example, the function mis_disconnect_cb() will be called when the MIS exits. Some applications may want to use this function to connect to a backup MIS, or process data before exiting:.

```
plat.when("DISCONNECTED", Callback(mis_disconnect_cb, 0));
```

This can also be referenced using the default_platform() function:

```
(Platform::default_platform()).when("DISCONNECTED", Callback(mis_disconnect_cb, 0));
```

mis_disconnect_cb() is defined as:

```
void
mis_disconnect_cb(Ptr, Ptr)
{
    // the MIS has disconnected.
     //   Nothing more we can do here.
    exit(0);
}
```

8.2.4 Using dispatch_recursive

dispatch_recursive() is used for communicating with the MIS. Most API calls use it internally to wait for responses from the MIS. Many applications may need to call it directly.

This function does a select on all open file descriptors to determine their state. If the parameter is set to TRUE, the select will block, waiting for input. If FALSE, then the select will poll once.

If your application is waiting for events, you will need to set up a routine which calls `dispatch_recursive()`:

```
Boolean
my_dispatcher()
{
    // call the pmi routine which checks for received
                 //  events.
    //   If an event has been received, the Callback
    //     routine specified in plat.when() will get
    //      called directly.
    dispatch_recursive((char)FALSE);
    usleep(100000);
    return FALSE;
}
```

For GUI applications, the GUI needs to have control. You can use the following function to call the dispatcher routine when the application is idle:

```
XtAppAddWorkProc(app_context, (XtWorkProc)my_dispatcher, NULL);
```

When a registered event does come in, the callback function registered by `Platform::when`() will be called directly.

For non-GUI applications, a dispatcher function would look similar to this:

```
       void
my_dispatcher()
{
    while (TRUE) {
        dispatch_recursive(TRUE);
    }
}
```

8.2.5 *Using the* `Image` *Class*

The `Image` class is the local representation of one object. Using the Distinguished Name (DN) and the object class, you can use an `Image` to get to all attributes associated with the DN.

In the em_cmipconfig example program, the function CMIPAgent::get_image() in CMIPAgent.cc takes the Entity Name entered pez, and transforms it into a DN:

```
agentTableType="CMIP"/agentId=id:"pez"
```

Using the object class cmipAgent, it creates an Image:

```
const char *OC = "cmipAgent";
const char *CMIP_TABLE = "agentTableType=\"CMIP\"";

    char fdn[1000];
    sprintf(fdn, "%s/agentId=", CMIP_TABLE);

    // GraphicString
    strcat(fdn, "id:\"");
    strcat(fdn, agentname);
    strcat(fdn, "\"");

    Image im(fdn, OC);
```

8.2.6 Booting an Image

To find out whether the Image object actually exists, call the Image::boot() function. If the Image refers to a creatable object, then boot will succeed whether or not the object actually exists. To make sure the object actually does exist, use Image::exists():

```
if (!im.boot()) {
        cout << "object doesn't boot" << endl;
        cout << im.get_error_string() << endl;
        return 1;
    }
    if (!im.exists()) {
        cout << "object doesn't exist" << endl;
        cout << im.get_error_string() << endl;
        return 1;
    }
```

8.2.7 Getting and Setting `Image` Attributes

Getting and setting attributes of an `Image` object are fairly straightforward. Once an object has been booted, you can get at each attribute using one of the `Image::get` functions. Since `get_str()` returns a `DataUnit`, we use the `DataUnit::chp()` function to get a `char *`.

```
char *aet;
char *psap;

aet = strdup(im.get_str("applicationEntityTitle").chp());
psap = strdup(im.get_str("presentationSelector").chp());
```

You can also use one of the `Image::set` functions. If the `Image` exists, use `Image::store()` to modify existing attributes. If this is a new object, use `Image::create()`.

```
if (!im.set_str("administrativeState", "unlocked")) {
    cout << "can't set attribute" << endl;
    cout << im.get_error_string() << endl;
    return FAILURE;
}

if (!im.set_str("operationalState", "enabled")) {
    cout << "can't set attribute" << endl;
    cout << im.get_error_string() << endl;
    return FAILURE;
}

if (im.exists()) {
    if (!im.store()) {
        cout << "can't store object" << endl;
        cout << im.get_error_string() << endl;
        return FAILURE;
    }
} else {
    if (!im.create()) {
        cout << "can't create object" << endl;
        cout << im.get_error_string() << endl;
        return FAILURE;
     }
 }
```

In many applications, the names of the attributes may not be known. If this is the case, then a function similar to the following may be necessary. This function gets all the attribute names, and prints the attribute name and its value.

```
void
print_image(Image &im)
{
    Array(DU) attr_names = im.get_attr_names();

    U32 maxlen = 0;
    for (U32 attrix = 0; attrix < attr_names.size; attrix++) {
        DU& name = attr_names[attrix];
        if (name.size() > maxlen)
        maxlen = name.size();
    }
    for (attrix = 0; attrix < attr_names.size; attrix++) {
        DU& name = attr_names[attrix];
        printf("%-*s => %s\n", maxlen, name.chp(),
        im.get_str(name).chp());
    }
}
```

8.2.8 *Using the* `Album` *Class*

The `Album` class comprises a set of related objects. These are implemented as a set of `Images`. `Albums` are used for deriving sections of the MIT. A scoped `get` on an `Album` produces the equivalent of booting an `Image` for each object in the tree.

In `em_cmipconfig`, Albums are used to keep track of the Entity List. We create an `Album` with Tracking and AutoImage on. Tracking tells the PMI to automatically update Images as they change. AutoImage tells the PMI to automatically boot the Image.

```
agent_album = new Album("agent_list");
agent_album->set_prop(duTRACKMODE, duTRACK);
agent_album->set_prop(duAUTOIMAGE, duYES);
```

8.2.8.1 Deriving an Album

Once we have set up an instance of an `Album`, we set it up to scope all the children under `'/systemId=<mis_host>/agentTableType="CMIP"'`. Then the call to `derive`() handles the actual booting of `Images`.

```
agent_album->set_derivation("agentTableType=\"CMIP\"/*");
agent_album->derive();
```

8.2.8.2 Registering an Album for Events

For this application, we want to be notified whenever an `Image` is added to this `Album` and when an `Image` is removed. We register a Callback, so when our application receives either of these events, the specified function will be called directly.

```
agent_album->when("IMAGE_INCLUDED", Callback(agent_added_cb, 0));
agent_album->when("OBJECT_DESTROYED", Callback(agent_deleted_cb, 0));
```

In `em_cmipconfig`, the Entity List gets updated to reflect the addition or deletion of objects.

```
void
agent_added_cb(Ptr userdata, Ptr calldata)
{
    CurrentEvent ce(calldata);
    Image im = ce.get_image();
    if (im != NULL) {
        char name[500];
        getAgentId(im, name);// get agent name
                     //  from Image
        XmString str = XmStringCreateSimple(name);
        XmListAddItem(agentlist_list, str, 0);
        XmStringFree(str);
    }
}
```

8.2.8.3 *Using the* `AlbumImage` *Class*

In order to get to the different `Images` in an `Album`, or the different `Albums` in an `Album` (an `Album` of `Albums`), we use the `AlbumImage` class. This is a very simple class, with only two functions: `next_album()`, and `next_image()`.

Using the `Album` defined above, we create an `AlbumImage` using `Album::first_image()`. Then using `AlbumImage::next_image()`, we can step through the `AlbumImage` creating new `Image` objects.

```
// populate list
AlbumImage ai;
for (ai=agent_album->first_image(); ai; ai = ai.next_image()) {
    Image im(ai);
    char name[500];
    name = im.get_str("agentId", OMIT_NEWLINES|OMIT_SPACES).chp();
    printf("agentId=%s",name);
}
```

8.2.9 *Using the* `Morf` *Class*

Morfs are used as wrappers around an abstract base class. Each instance of a Morf contains information which the PMI can decode.

8.2.9.1 *Using* `Coder` *and* `Syntax` *with* `Morfs`

In `em_cmipconfig`, the Managed Object Name field is specified as an FDN, but the actual Object in the MIT stores this value as a SET of SETs.

```
'systemId="pez"/subsystemId="test"'
```

becomes:

```
{{{{systemId,"pez"}},{{subsystemId="test"}}}}
```

In order for this to be done transparently, we can use the `Coder` and `Syntax` classes. Any time an `Image::get_str()` or `Image::set_str()` is done for attributes of type EM-MPA-ASN1.ManagedDNs our functions DNSetCoderData::get_str() and DNSetCoderData::set_str() are called.

```
Syntax dnset_syn = Syntax( plat, "EM-MPA-ASN1.ManagedDNs");
dnset_syn.set_coder( Coder(*new DNSetCoderData(plat)));
```

8.2.9.2 *Splitting Morfs*

The best way to explain the use of a Morf is with an example. In em_cmipconfig, when the Coder DNSetCoderData::get_str() gets called, a Morf representing the Image for the managed DN attribute is passed in. We also get a Morf returned from the Image::get_raw() function.

The following example was modified from `em_cmipconfig`. It gets the value of an attribute of an `Image` object, and uses Morfs to display the information.

```
    image = Image("agentTableType=\"CMIP\"/agentId=\"pez\"");
    image.boot();
    if (!image.exists())
        exit(1);
    Morf m;

    m = image.get_raw("managedDNs");

    printf("MORF=%s\n",m.get_str().chp());

    morf_split(m);


void
morf_split(
        Morf& m
          )
{
        if (m.is_choice()){
                morf_split(m.extract(DU()));
        } else if (m.is_list()) {
                Array(Morf) mm = m.split_array();
                for (int i=0; i<mm.size; i++) {
                        cout << "morf[";
                        cout << i ;
                        cout << "] = ";;
                        cout << mm[i].get_str().chp();
                        cout << endl;
                        morf_split(mm[i]);
                }
        } else {
                cout << endl;
                cout << "---scalar value--->";
                cout << m.get_str().chp();
                cout << endl << endl << endl;
        }
}
```

Using the example above, the Morfs will initially represent:

```
{ { { {
    attributeId "Rec. X.721 | ISO/IEC 10165-2 : 1992":systemId,
    attributeValue name : "pez"
} }, { {
    attributeId "Rec. X.723 | ISO/IEC 10165-5":subsystemId,
    attributeValue "test"
} } } }
```

In order to get to the attributes of the `Morf`, we use the function `split_array()`. For each `split_array`, the newly represented `Array` of Morfs will represent a subset of the first `Morf`. After the first `split_array()`, only three encasing brackets are used.

```
{ { {
    attributeId "Rec. X.721 | ISO/IEC 10165-2 : 1992":systemId,
    attributeValue name : "pez"
} }, { {
    attributeId "Rec. X.723 | ISO/IEC 10165-5":subsystemId,
    attributeValue "test"
} } }
```

If we continue splitting, we eventually get an `Array` of `Morfs` with size 2.

```
morf[0] = "Rec. X.721 | ISO/IEC 10165-2 : 1992":systemId
morf[1] = pez
```

8.3 High-Level PMI Examples

These examples are smaller in scope than those of the Section 8.2, “em_cmipconfig Example.” Each example shows how to perform one or two specific tasks.

This section provides the following information about the examples:

- General descriptions of all of the examples—sorted by task
- Detailed descriptions of a few selected examples

For the detailed descriptions, sometimes the major actions are described, and sometimes all of the source code is provided with comments for almost every action, whichever seems most appropriate. The sample files have some of these descriptions inserted as comments.

The examples described below are in `$EM_HOME/src/pmi_high`.

8.3.1 General Descriptions

The sample files are summarized below, sorted by task.

Table 8-1 High-Level PMI Examples

Task	File	Description
Access control: Define dialog box for querying user passwords.	`passwd_dialog/*`	For more information about access control examples, see Section 8.5, "Access Control Examples."
Access control: Use feature-level access control.	`access_feature/*`	For more information about access control examples, see Section 8.5, "Access Control Examples."
Album: Asynchronously derive an album of level-2 objects. *Alternate*: Asynchronously get all children of an object.	`album_asyn.cc`	Asynchronous version of `album.cc`.
Album: Derive an album of level-2 objects. *Alternate*: Get all children of an object.	`album.cc`	Derives an album of all objects at level two, under the root (/) object. For each object, this prints the object name and each attribute name and value. *Alternate*—It specifies level two, but the source can be changed to specify an object, such as `MO`, or `TOPO`, or `NC_POLL_RATE_CONTAINER`; then it gets all children of the specified object.

Table 8-1 High-Level PMI Examples *(Continued)*

Task	File	Description
Album: Derive an album of log objects.	`album_logObj.cc`	Derives an `album` of all log objects created under the host. In this case, the host is the one where the MIS is running. This album has the following properties: (1) If a new log object is created, an `image` representing it is included in the `album` (2) Some callbacks for tracking events are invoked.
Album: Derive an album of an object, with tracking.	`album_wait.cc`	Derives an album of an object, with tracking. The PMI automatically updates the album when images are added or deleted. The application is notified by `callback` functions which are set prior to album derivation. This program specifies the `nerveCenter` pollrate container or all `cmipsnmpproxy` agents.
Album: Derive an album for RPC groups.	`derive_rpc.cc`	Derives an album for RPC groups, for the specified host and MIS.
Album: Derive an album for SNMP groups.	`derive_snmp.cc`	Derives an album for SNMP MIB II groups, for the specified host and MIS.
Attribute: Asynchronously get object; print attributes.	`get_asyn.cc`	Asynchronous version of `get.cc`.
Attribute: Get object; print attributes.	`get.cc`	Gets an object, then gets and prints attributes. Takes up to three arguments, as follows: `get` *<fdn> <objectClass> <attribute>* If an attribute is specified, it gets and prints that attribute, otherwise, it gets and prints all attributes. If no arguments are specified, it gets the topology database `topoType` object named `Host`, then gets and prints all attributes.
Attribute: Asynchronously set attribute value in object.	`set_asyn.cc`	Asynchronous version of `set.cc`.

Table 8-1 High-Level PMI Examples *(Continued)*

Task	**File**	**Description**
Attribute: Set attribute value in object.	`set.cc`	Sets an attribute value in an object. Example: sets `maxLogsize` in the log object `AlarmLog`. Takes four arguments, as follows: `set` <*fdn*> <*objectClass*> <*attributeName*> <*attributeValue*> By default (if no arguments are specified), it boots the `NerveCenter` PollRate `Fast` object and sets its poll rate value to 777.
Attribute: Get attributes of object.	`image_boot.cc`	Gets attributes of specified object from the MIS. Boots an image that contains attributes. If no attributes are specified, obtains all attributes. If attributes are specified, obtains only specified attributes.
Event: Create an Event Forwarding Discriminator	`efd.cc`	Creates an EFD (Event Forwarding Discriminator) and sets up one MIS (mis_1) to forward events to another MIS (mis_2).
Event: Send notification with time *now.*	`event_send1.cc`	Sends an event notification to the MIS. It sends the event name, event information, and a default time stamp of *now.*
Event: Send notification with *custom* time stamp.	`event_send2.cc`	Sends an event notification to the MIS. It sends the event name, event information, and a *custom* time stamp.
Event: Send notification with ASN.1 time stamp.	`event_send3.cc`	Sends an event notification to the MIS. It sends the event name, event information, and a custom time stamp using ASN.1 notation. It uses `send_event(DU,Asn1Value)` and demonstrates encoding/decoding of `Asn1Values`.

Table 8-1 High-Level PMI Examples *(Continued)*

Task	File	Description
Event: Listen for all events.	`event.cc`	Listens to the MIS for all events. If an event occurs, this program prints out the contents. This program makes the initial connection to the MIS; if the MIS closes the connection, this program receives that event and exits cleanly.
Event: Listen for specified events that happen to all objects.	`event_app1.cc`	Listens for events that match the specified discriminator, then prints out the event contents.
Event: Listen for all events that happen to specified objects.	`event_app2.cc`	Listens for all events that happen to the specified objects. It selects objects by setting the discriminator on all managed object classes that are specified as arguments.
Event: Listen for creations of log objects.	event_create_logObj.cc	Listens to the MIS for log object creation events. If such an event occurs, this program prints out the contents. The information of interest is the list of attribute name and value pairs.
File descriptor: Get file descriptor for MIS connection.	`plat_get_conn_fd.cc`	Gets the file descriptor corresponding to the connection to the MIS. It connects to the MIS, then calls the `Platform::get_connection_fd()` routine to get the file descriptor, and then prints it.
Fully Distinguished Names (FDNS): Convert FDNs to nicknames. *Alternate*: Get and set nicknames.	`nickname/*`	Files in this directory illustrate how to use the nickname service-related API calls to get and set nicknames. For more information, see the `README` file in this directory.
Local object: Count all local objects. *Alternate*: Store FDN entries to hash table for searching.	`object_count.cc`	Demonstrates how to count all local objects. It also shows how to store FDN entries to a hash table and search the entries in the hash table. It also calls an action for an `emFdn` object instance.
Log object: Asynchronously create a log object.	`create_asyn.cc`	Asynchronous version of `create.cc`.

Table 8-1 High-Level PMI Examples *(Continued)*

Task	File	Description
Log object: Create a log object.	`create.cc`	Creates an object that logs specified events (creates it in the MIS).
Log object: Asynchronously delete a log object.	`delete_asyn.cc`	Asynchronous version of `delete.cc`.
Log object: Delete a log object.	`delete.cc`	Deletes a specified object that logs events in the MIS.
MDR object: Invoke object actions.	`mdr_action.cc`	Demonstrates how to invoke MDR object actions. MDR (Meta Data Repository) is a storage area within the MIS for the descriptions of managed objects. A description of every known object is stored in the MDR. MDR supports many actions to provide information about known objects.
MIS-to-MIS communication: GUI program for setting up MIS-to-MIS communication.	`cmipconfig/*`	`em_cmipconfig` is an example GUI program that uses many aspects of the high-level PMI to configure MIS-to-MIS communication. For more information, see Section 8.2, “em_cmipconfig Example.”
Morf: Use Morf to split a compound data attribute value into scalar data values.	morf_topoNode.cc	Demonstrates how to use Morf to split a compound data attribute value to scalar data value. This program uses `env EM_SERVERU` to get remote host name. If EM_SERVER is not set, the program will try to connect to MIS on local host.
topoNodeUserData: Get and set `topoNodeUserData` attribute for `topoNode` object instance.	`topo_user_data/*`	Files in this directory show how to set and get user-defined `topoNodeUserData` attribute for a `topoNode` object instance. For more information, see the `README` file in this directory.

8.3.2 *Details for Selected Examples*

8.3.2.1 `get`

File—`pmi_high/get.cc`

Purpose—Get and print attributes of an object.

Syntax—`get` [<*fdn*> <*ObjectClass*> [<*attribute*>]]

<*fdn*> is the fully distinguished name of the object to retrieve,

<*objectClass*> is the class of the object to retrieve

<*attribute*> is an attribute of the object

Defaults

- If no attribute is specified, get and print all attributes
- If no parameters at all, get and print all attributes of the topology database `topoType` object named `Host`.

Usage

Example 1: Get all attributes of the object named <*titleist*> of class <*system*>.

```
get '/systemId="titleist"' system
```

Example 2: Get the <*operationalState*> attribute of object named <*titleist*>.

```
get '/systemId="titleist"' system operationalState
```

Major Coding Actions

- With the platform object <*plat*>, connect to the host specified in <*hostname*>.

```
if (!plat.connect(hostname, "test_get")) some error processing…
```

- Construct an image of object <*dn*> of class <*class_name*>, save the image in <*im*>.

```
Image im = Image(dn, class_name);
```

- If there is an attribute specified on the command line, then for image <*im*>, get that attribute and print it.

```
if (strlen(attribute_name)) {
        cout << attribute_name;
        cout << "\t" << im.get_str(attribute_name).chp() << endl;
}
```

- If no attribute is specified, get and print all attributes as follows:
    - Create the array <*attr_names*>, an array of type `DataUnits`.
    - For image <*im*>, get all attributes; put them into <*attr_names*>.
    - Print each attribute name and value.

```
Array(DU) attr_names = im.get_attr_names(); // Create attr_names array of data units.
                                       // Get all attributes of im, put them in array.

for (int i=0; i<attr_names.size; i++) {
    DU& name = attr_names[i];                            // Get name as data unit.
    char *short_name = strrchr(attr_names[i].chp(),':'); // Get name as string.
    cout << ++short_name << "\t";                        // Print name.
    cout << im.get_str(name,                             // Get value.
            USE_EXPLICIT_CHOICE|OMIT_NEWLINES).chp();    // Print value.
    cout << endl;
}
```

Remarks for the above Example

- "`Array(`*t*`)` *v*" is a macro to create an array *v* of type *t*.
- The `DU` class is a `typedef` for the `DATAUNIT` class.
- The `im.get_str()`would return a `DATAUNIT`,
  but `im.get_str().chp()` returns a character pointer, *for printing*.
- Note the following error handling code from `get.cc`:

```
    if (!im.boot()) {
      cout << "Failed to boot " << dn << endl;
      cout << im.get_error_string() << endl;
      exit(5);
    }
```

If `im.boot()` fails, the reason and type of error can be obtained by calling functions `im.get_error_string()` and `im.get_error_type()`. The object `im` is marked as in error and by default—following a strict error checking model—all subsequent function calls on `im` will fail.

- The default error checking model can be modified as follows:

```
  Error::error_entry_callback = overlook_previous_error();
```

where `overlook_previous_error()` is defined as follows:

```
void overlook_previous_error(Error *err) {
     err.reset_error()
 }
```

With the above modification, if a derived-from-`Error` object is in error because of a previous call, then, in general, any function called that is a member of that object first calls `overlook_previous_error()`. In the `get.cc` example above, the next function call, `im1.get_str()`, as a first step, calls `overlook_previous_error()`, which resets the error of that object.

8.3.2.2 `album`

File—`pmi_high/album.cc`

Purpose—Derive an album of all objects through level 2 under the root (/) object. It gets an prints attributes of objects below the top of the network and down through level 2.

Syntax—`album [-h` <*HostName*>`]`

<*HostName*> is the name of the host,

Default—If no host name is specified, it uses `localhost`.

Usage

Example 1: Show level 1 and level 2 objects on local host.

```
album
```

Example 2: Show level 1 and level 2 objects on the host <*SomeOtherHost*>.

`album -h` <*SomeOtherHost*>

If you use the alternate coding, the above examples show the children of the specified object. The object is specified in the source code.

Source Code for `album.cc`

The code is listed below and is similar to `pmi_high/album.cc`, except for the comments, and possible updates or bug fixes. The alternate code, for a related purpose, is provided later in this section.

`album.cc` source file

```
// Purpose: Show all objects below the top of the network, down through level 2.
//
// How: Starting at top of network, make the set of all objects that are
//      either 1 level or 2 levels below. That is, derive an album of objects
//      through level 2 under root (/). For each object, print the object name
//      and then print each attribute name and value.

#include <netdb.h>
#include <sys/systeminfo.h>
#include <stdio.h>
#include <hi.hh>

DU NC_POLL_RATE_CONTAINER = "emApplicationType=\"NerveCenter\"/em-name="
                            "\"pollRateContainer\"/LV(1)";
DU MO                     = "/systemId=name:\"titleist\"/internetClassId="
                            "{1 3 6 1 4 1 42 2 2 2 9 2 4 1 0}/*";

int main(
        int argc,
        char **argv )
{
        Album test_album;
        Platform plat = Platform(duEM);

        // Get the host name from the environment variable.
        char *host = getenv("EM_SERVER");
        if (!host) {
                host = new char[MAXHOSTNAMELEN+1];
                sysinfo(SI_HOSTNAME, host, 255);
        }
```

(continued)

`album.cc` source file (*continued*)

```
        if (!plat.connect(host, "test_album")) {    // Connect to the host MIS.
                cout << "Failed to connect to "<< host << endl;
                cout << plat.get_error_string() << endl;
                exit(1);
         } else {
                cout << "Connected. " << endl;
         }
        test_album = Album("myalbum");                // Construct test_album.
         if (test_album.get_error_type() != PMI_SUCCESS) {
                 cout << test_album.get_error_string() << endl;
                 exit(2);
         }
   if (!test_album.set_derivation("/LV(2)")) {          // Derive test_album.
                   cout << test_album.get_error_string() << endl;
                   exit(3);
   }
   // For each image ai in test_album, get the object name, then get attributes.

   AlbumImage ai;
   for ( ai = test_album.first_image(); ai; ai = ai.next_image() ) {
     Image im(ai);                                 // Construct image im from ai.
     if (im.get_error_type() != PMI_SUCCESS) {
        cout << im.get_error_string() << endl;
        exit(5);
     }
     DU objname = im.get_objname();                 // Get object name.
     cout << endl << "Fully Distinguished Name: "; // Print object name.
     cout << objname.chp() << endl;
     //
     // Boot the image im to get all attribute names into im.
     //
     if (!im.boot()) {
         cout << im.get_error_string() << endl;
         exit(6);
     }
     // Get the names from im and put them into the attr_names array.

     Array(DU) attr_names = im.get_attr_names();
```

`album.cc` source file (*continued*)

```
    //
    // For each attribute ai, get name & value as strings, print both.
    //
    for (int i=0; i<attr_names.size; i++) {
         char *name = strrchr(attr_names[i].chp(),':');  // Get name as str.
         cout << ++name;                                  // Print name.
         cout << ": ";
        cout << im.get_str(name).chp() << endl; // Get val as str, print it.
    }
  }
  exit(0);
}
```

Remarks for the above Example

- "`Album test_album;`" `Album` is a class, defined in a header file.
- "`AlbumImage ai;`" `AlbumImage` is a class, defined in a header file.
- "`Array(DU) attr_names =`" `Array(`*t*`)` *v* is a macro to create an array *v* of type *t*.
- "`DU`" The `DU` class is a `typedef` for the `DATAUNIT` class.
- "`plat = Platform(duEM);`"
  - `duEM` is a C++ constant defined to be "`EM`", the only platform supported.
  - `Platform(`*p*`)` is a macro to create an array of type *p*.
- "`Platform plat;`" `Platform` is a class, defined in a header file.
- "`Image im(ai);`" Constructs an image `im` from an image `ai`.

8.3.2.3 event_app2

File—`pmi_high/event_app2.cc`

Purpose—Listen for events that happen to the specified objects.

Syntax—`event_app2` <*moc1*> <*moc2*> … <*mocN*>

Each <*moci*> is a managed object class,

Default—Connects to the local host.

Source Code for event_app2.cc

The code listed below is similar to pmi_high/event_app2.cc, except for the comments, and possible updates or bug fixes.

event_app2.cc source file

```
// Purpose: Listen for all events that happen to the specified objects.
//
// How: 1. This program selects objects by setting the discriminator on all
//         managed object classes that are specified as arguments.
//      2. This program first connects to, and then listens to the MIS.
// Usage:   event_app2 <moc1> <moc2> ... <mocN>
//          where each of the <moci> is a managed object class.
// Notes: 1. This program never exits; it enters an infinite listening loop.
//           Use Control-C to terminate it.
//        2. For more on the discriminator, see the "Log Manager" chapter,
//           in the Solstice EM Reference Manual.

#include <netdb.h>
#include <sys/systeminfo.h>
#include <hi.hh>

void raw_cb(
        Ptr,
        Ptr calldata
           );
int main(  int argc, char **argv
    )
{
        // Setup the connection to the MIS      .

      Platform plat = Platform(duEM);          // Declare plat to be a Platform.
        if (plat.get_error_type()!=PMI_SUCCESS) {
                cout << plat.get_error_string() << endl;
                exit(1);
        }

        char *host = getenv("EM_SERVER");           // Get the host name.
        if (!host) {
                host = new char[MAXHOSTNAMELEN+1];
                sysinfo(SI_HOSTNAME, host, 255);
        }
```

(continued)

event_app2.cc source file (*continued*)

```
        cout << "Connecting to ... " << host << endl;
       if (!plat.connect(host, "event_application")) {    // Connect to the host.
                cout << "Failed to connect to " << host << endl;
                cout << plat.get_error_string() << endl;
                exit(2);
        }
        cout << "Connected." << endl << endl;

        Array(DU)       classes(argc-1);
       for(int i=0; i < argc-1; i++) { // Copy any classes from the command line.
                classes[i] = strdup(argv[i+1]);
        }
        /**********************************************************
        // Alternate: The commented out part below can be used to
        //            filter events based on the type of event,
        //            such as objectCreation, etc.
        // Set the discriminator so only selected events are forwarded.
        Array(DU)       events(2);
        events[0] = "objectCreation";
        events[1] = "objectDeletion";
        if(!em.replace_discriminator_classes(classes, events)) {
                cout << plat.get_error_string() << endl;
                exit(3);
        }
        **********************************************************/

        if(!plat.replace_discriminator_classes(classes)) { // Set discriminator.
                cout << plat.get_error_string();
                exit(4);
        }
       if (!plat.when("RAW_EVENT", Callback(raw_cb, 0))) {// When to call raw_cb.
                cout << plat.get_error_string() << endl;
                exit(5);
        }
        cout << "Waiting for events...... " << endl << endl;

        while(1) {                              // Enter the infinite listen loop.
                dispatch_recursive(TRUE);
        }
        exit(0);
}
```

(continued)

`event_app2.cc` source file (*continued*)

```
void    // Define a function to do something with event notification.
raw_cb(Ptr, Ptr calldata)
{
    // Print interesting things about the event.
    CurrentEvent ce(calldata);
    DU tmp;
    cout << "****** RAW_EVENT received ******" << endl;
    if(tmp=ce.get_event())
        cout << "EVENT = " << tmp.chp() << endl;
        cout << endl;
}
```

Remarks for the above Example

There are two statements that set the discriminator. One is commented out. Use one or the other, but not both. The one that is currently commented out does need the three statements above it, so uncomment carefully.

8.3.2.4 `image_boot`

File—`pmi_high/image_boot.cc`

Purpose—Get attributes of the specified object from the MIS.

Syntax—image_`boot -d` <*dn*> [`-a` <*attr1*> `-a` <*attr2*> … `-a` <*attrN*>]

<*dn*> is the distinguished name of the object

<*attri*> are the specified attribute names.

Defaults

- If no attributes are specified, obtains all attributes.
- Connects to the local host.

Source Code for `image_boot.cc`

The code listed below is similar to `pmi_high/image_boot.cc`, except for the comments, and possible updates or bug fixes.

`image_boot.cc` source file

```
// Purpose: Get attributes of the specified object from the MIS.
//
// How:  This program boots an image that contains attributes,
//       that is, sends a GetReq (get request) to the MIS.
//       It does not display any of the attributes, but to see them,
//       you can use the utility em_debug.
//
// Syntax:  image_boot -d <dn> -a <attr1> -a <attr2> ... -a <attrN>
//
//          If no attributes are specified, it obtains all attributes.
//         If attributes are specified, it obtains only the specified attributes.
//
// Examples:
//
//   % image_boot
//   % image_boot -d 'logId=string:"AlarmLog"'
//   % image_boot -d 'logId=string:"AlarmLog"' -a logFullAction -a currentLogSize
//
// Why: This is useful for tracking event flow between MIS and applications.
//      Used wisely, this method can improve performance, both in terms of time
//      and memory usage by application.

#include <netdb.h>
#include <sys/systeminfo.h>
#include <hi_d3.hh>

int main(int argc, char **argv)
{
    // Get the host name.

    char *host = getenv("EM_SERVER");
    if (!host) {
        host = new char[MAXHOSTNAMELEN+1];
        sysinfo(SI_HOSTNAME, host, 255);
    }

    char      dn[300];
    sprintf(dn, "/systemId=name:\"%s\"", host);
```

(continued)

`image_boot.cc` source file (*continued*)

```
    // Parse the cmd line for options.

    Array(DU) attrs;
    // attrs = Array(DU)(argc - 4);
    int  c = 0;
    int  i = 0;
    attrs = Array(DU)((argc-3)/2);

    if (argc >= 4) {
        while ((c = getopt(argc, argv, "d:a:")) != EOF) {
        switch (c) {
          case 'd':
            strcpy(dn, optarg);
            break;
          case 'a':
            attrs[i] = strdup(optarg);
            i++;
            break;
          case '?':
            exit(1);
        }
        }
    } else if (argc == 3) {
        while ((c = getopt(argc, argv, "d:")) != EOF) {
        switch (c) {
          case 'd':
            strcpy(dn, optarg);
            break;
          case '?':
            exit(2);
        }
        }
    } else if (argc == 1) {
        cout << "No command option using the default value. " << endl;
        cout << "dn = " << dn << " with all attributes. " << endl << endl;
    } else {
        cout << "Usage: boot -d <dn> -a <attr1> -a <attr2> " << endl;
        exit(3);
    }
```

(continued)

`image_boot.cc` source file (*continued*)

```
    Platform plat = Platform(duEM);
    cout << "Connecting to ... " << host << endl;
    if (!plat.connect(host, "boot")) {            // Connect to the host MIS.
        cout << "Failed to connect to "<< host << endl;
        cout << plat.get_error_string() << endl;
        exit(4);
    } else {
        cout << "Connected." << endl << endl << endl;
    }

    Image im(dn);
    if (im.get_error_type() != PMI_SUCCESS) {
        cout << im.get_error_string() << endl;
        exit(5);
    }
    if(argc > 4 )
    {
        // Get selected attributes.

        if(!im.boot(attrs)){       // Boot the image WITH the attribute list.
            cout << im.get_error_string() << endl;
            exit(6);
        }
        cout << "booted with selected attributes. " << endl;
    } else {
        // Get all attributes.

        if(!im.boot()){        // Boot the image WITHOUT the attribute list.
            cout << im.get_error_string() << endl;
            exit(7);
        }
        cout << "booted with all attributes. " << endl;
    }
    Array(DU) attr_names = im.get_attr_names();
       for (i=0; i<attr_names.size; i++) {    // Print the booted attributes.
        char *name = strdup(attr_names[i].chp());
            cout << name;
            cout << ":  ";
            cout << im.get_str(name).chp() << endl;
    }
    exit(0);
}
```

8.4 Low-Level PMI Examples

The following examples are smaller in scope than Section 8.2, “em_cmipconfig Example." Each example shows how to perform one or two specific tasks.

The examples in Table 8-2 are in `/opt/SUNWconn/em/src/pmi_low`.

Table 8-2 Low-Level PMI Examples

File	**Description and Notes**
`album_low.cc`	Sends scoped GetReq to MIS. The high level PMI equivalent is `Album::derive()` with scoping.
`boot_low.cc`	Sends GetReq to MIS with a selected attribute list. The high level PMI equivalent is `Image::boot(attrlist)`.
`create_low.cc`	Sends `CreateReq` to MIS with a selected attribute list. The high level PMI equivalent is `Image::set()` calls followed by `Image::create()`.
`delete_low.cc`	Sends `DeleteReq` to MIS. The high level PMI equivalent is `Image::destroy()`.
event_gen_low.cc	Sends events directly to the event manager in the MIS.
`filter_low.cc`	Sends scoped, filtered `GetReq` to MIS. The high level PMI equivalent is `Album::derive()` with scoping and filtering.
`set_low.cc`	Sends `SetReq` to MIS with a selected attribute list. The high level PMI equivalent is `Image::set()` calls followed by `Image::store()`.
`simple_low.cc`	Sends a simple `GetReq` to MIS for the root object. No attributes are retrieved. There is no high level PMI equivalent.

8.5 Access Control Examples

Starting with the EM 2.0 release, Solstice EM includes access control mechanisms whereby you can control user’s access to entire applications and to individual features within those applications.

To give you a better understanding of how to implement access control for your applications, Solstice EM provides the following examples:

- The files in the `$EM_HOME/src/pmi_hi/passwd_dialog` directory provide an example that illustrates creating a dialog box that queries the user for his password and then starts up the application only if the user is allowed to access that application.

- The files in the `$EM_HOME/src/pmi_hi/access_feature` directory provide an example that illustrates feature-level access control.

The files in these directories are heavily commented, so you should be able to look at the files and understand how access control can be implemented for your applications.

Table 8-3provides a description of the specific files provided and what they contain.

Table 8-3 Access Control Examples

File	Description and Notes

`passwd_dialog/`
`password_dialog.cc`
`password_dialog.hh`
`password_gui.cc`
`password_gui.hh`
	Files provided for developer to create dialog box for verifying user login ID and password.

`passwd_dialog/` `dialog_box.cc`	Demonstrates how to use a dialog box to query password in a GUI-based application and to replace the password query mechanism used by Solstice EM. This demonstrates how to use the password_gui.* and password_dialog.* files.
`access_feature/` `access_feature_level.cc`	Demonstrates how to use feature-level access control in a user-developed application.
`access_feature/` `access_reg_application`	Script to create access control objects in MIS.

8.6 Other API Examples

In addition to the high-level PMI and low-level PMI examples listed in this chapter, Solstice EM includes example files for the other API modules provided with the product. Table 8-4 identifies these examples. All directories are relative to `$EM_HOME/src` (typically, `/opt/SUNWconn/em/src`).

Table 8-4 Other API Examples

Directory	File Name	Description
`app_api`	`app_api/*`	Directory contains files that illustrate how to use the application-to-application API.
`gdmo_example`	`coffee.asn1`	Provides ASN.1 definitions for the coffee object.
	`coffee.cc`	Provides GDMO definitions for the coffee object.

Table 8-4 Other API Examples

Directory	File Name	Description
	`coffee.gdmo`	Creates a coffee object, finds out whether there is any coffee in the coffee pot, and sends a request to brew some if there is none.
`grapher_api`	`grapherAPItest.cc`	Illustrates how to use the functions available in the Grapher API.
`mpa_pdm`	`mpa_pdm/docs/*`	Directory contains ASN.1 and GDMO files for building MPA and PDM examples.
	`mpa_pdm/src/*`	Directory contains source files to build MPA and PDM.
	`mpa_pdm/test_mpa/*`	Directory contain sample programs used to test MPA.
	`mpa_pdm/test_pdm/*`	Directory contains sample programs used to test PDM.
	`mpa_pdm/*`	Directory contains files to ping an MPA.
`nci`	`nci/*`	Directory contains files that illustrate how to use the Nerve Center Interface library.
`objop`	`objop/*`	Directory contains scripts (not C or C++ programming examples) used to create, delete, and set attributes of objects using the `em_objop` utility.
`odt`	`odt/*`	Directory contains examples of how to develop object behaviors for managed object classes in the Solstice EM MIS. For more information about these examples and how to use the Object Development Tools (ODT), see Chapter 5, “Developing Object Behaviors."
`topo_api`	`print_topo.cc`	Program for printing out information about topology nodes using the topology API.
	`topo_events.cc`	Program for creating and modifying attributes related to a topology node and gathering resulting event information through the topology API.
	`traverse.cc`	Program for traversing the nodes of a topology tree using the topology API.
`viewer_api`	`viewer_api/*`	Directory contains files that illustrate how to use the Viewer API.

Protocol and Management Adaptors 9

This chapter reviews some of the important concepts behind the Management Protocol Adaptor (MPA) and Protocol Driver Module (PDM) model. It uses the sample MPA and PDM implementation included in the samples directory to illustrate both the concepts and the specifics of the MPA / PDM interfaces and environments.

9.1 Introduction to EM Protocol Adaptors

The MIS (Management Information Server) is responsible for maintaining the MIT (Management Information Tree) and ensuring that all activity within the MIT is transparent to an application. This transparency allows applications to make requests for information in a normalized fashion without regard for object location or communications protocol. The MIS resolves the request and routes it to an appropriate entity that is capable of making the correct protocol request. These entities are called **protocol adaptors**. Adaptors exist to map information into the MIT maintained by the MIS.

There are two kinds of adaptors:

- Management Protocol Adaptors (MPAs) which exist as separate processes from the MIS
- Protocol Driver Modules (PDMs) which are linked directly to the MIS process

The MIS is shipped with three adaptors: the SNMP PDM, the RPC PDM, and the CMIP MPA. These adaptors provide the protocol translation into the SNMP, RPC, and CMIP domains, respectively.

9.1.1 Differences between Protocol Adaptors and Management Adaptors

The fundamental difference between MPAs and PDMs is that an MPA is a separate process from the MIS where as a PDM links directly to the MIS.

In addition, other differences include:

- MPA requests contain routing information.
- MPAs and PDMs are initialized differently.

MPAs offer some basic advantages over PDMs because they are easier to develop, debug, and support, and they impact the MIS less than PDMs from a stability and performance viewpoint.

The interface between the adaptors and the MIS is well defined and allows for the addition of other user-provided adaptors.

9.1.2 Review of MIS Architecture

The MIS has a modular architecture. Modules are connected by Service Access Points (SAPs). These SAP's provide an asynchronous bidirectional message pipe interface. The core of MIS is the Message Routing Module (MRM) which routes messages to the appropriate modules. New modules can be added to the MIS by attaching a SAP from the new module to the MRM. Each SAP has a unique address which is registered with the MRM when it is attached.

New adaptors are introduced to the system by creating a new SAP to the MRM. (For MPA, no new SAP needs to be created. All MPAs are routed by the same MPA SAP.)

Figure 9-1 illustrates some of the modules within the MIS. The OAM is the Object Access Module. The EMM is the Event Management Module. Logs are managed by the LMM (Log Management Module) and the Persistent Object Store is maintained behind the BS Intf. (Backing Store Interface).

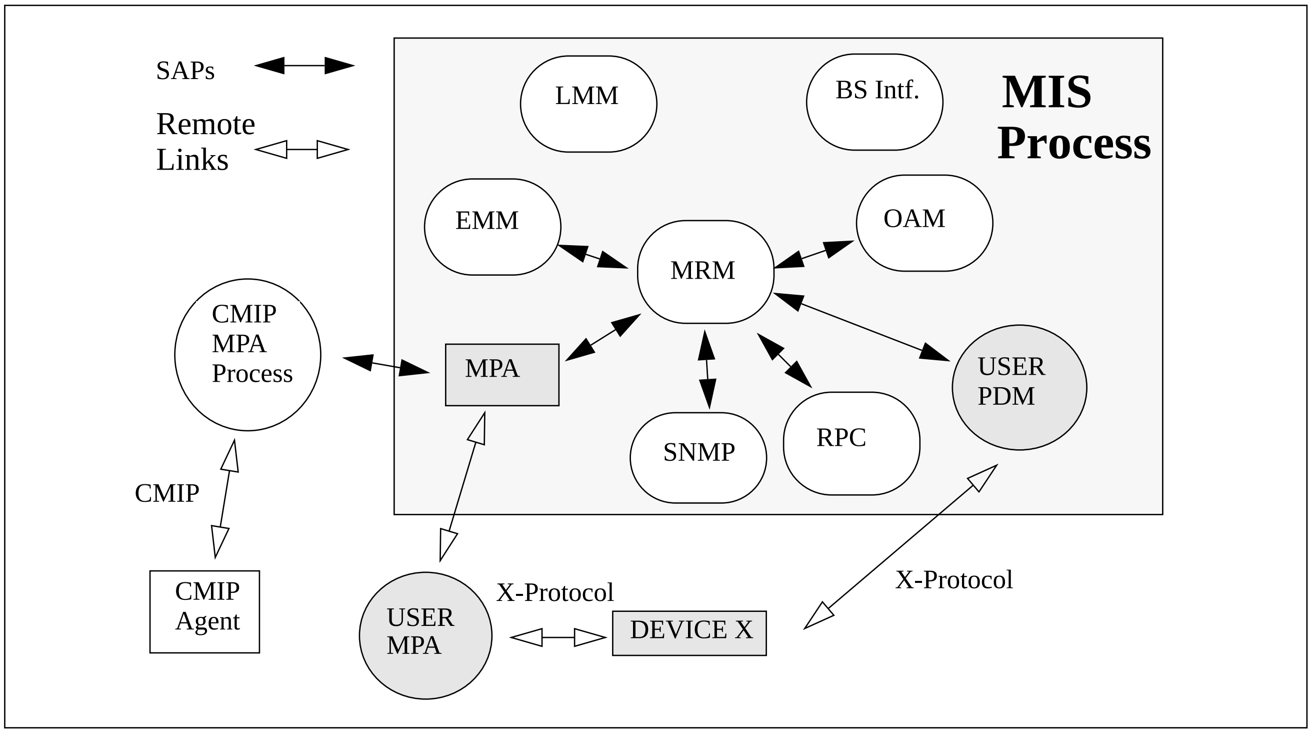


Figure 9-1 Overview of Modules

The shaded components in Figure 9-1 illustrate how users can add new adaptors to communicate to a proprietary device X. The two choices are clearly indicated, MPA or PDM.

All requests are initially sent to the Message Routing Module(MRM). The MRM uses configuration information maintained in the MIT to determine where to satisfy the request. Local requests are sent to the OAM. Remote requests are directed to the appropriate adaptor that has registered to handle that remote portion of the MIT.

Note – Both the MPA and the PDM use SAP to communicate to the MRM. There is very little difference between a MPA and a PDM once the SAP has been created. They both receive identical messages through the exact same interface.

9.2 Initializing an Adaptor

This section describes how Management Protocol Adaptors and Protocol Driver Modules are initialized. It includes the following topics:

- Service Access Points
- Message Protocol Adaptor Initialization
- Protocol Driver Module Initialization

9.2.1 Services Access Points (SAPs)

SAPs provide an asynchronous message passing service. The message set is based on the set of CMIP Protocol Data Units (PDUs). The complete set of messages that can be passed over a SAP is defined in `message.hh`.

All SAP's are C++ class instances derived from the C++ `MessageSAP` class defined in `message.hh`. This class defines the interfaces that allow messages to be sent and received over SAP pairs. An initialized SAP consists of a coupled pair of `MessageSAP` instances.

Note – From here on SAP is meant to mean an initialized SAP pair.

Once a SAP has been initialized, the following functions may be performed:

```
SendResult     send(MessagePtr mp, MTime block_time = INFINITY);
SendResult      send(MessagePtr mp, const Callback &cb,
                             MTime block_time = INFINITY);
Result receive_request(MessagePtr &mp);

Result receive_response(MessId m_id, MessagePtr &mp);
Result receive_response(ResponseHandle rh, MessagePtr &mp);
void    cancel_callback(MessId m_id);
void    cancel_callback(Callback &cb);
```

This set of functions provides the following message services:

Table 9-1 Message Services

Function	**Service Provided**
Send	Send and expect no Response
Send	Send and schedule to Receive a Response
Receive	Receive a message of a particular id
Receive	Receive the first Message on the incoming Q
Cancel	Cancel callbacks for a particular Message
Cancel	Cancel callbacks for a particular Callback

The interface is defined to be asynchronous. This is achieved using callbacks. Callbacks are a C++ class that consist of a static function pointer and user data. The user data is passed as a parameter to the callback function when the function is invoked.

Example 1:

The following example from the fdn_register function in `samp_utils.cc` illustrates a CONFIRMED request message.

```
// Need source so responses can come back.
areq->source.aclass = AC_PRIMITIVE;
areq->source.atag = my_sap_no;
Callback recv_cb((CallbackHandler)pdm_receive_resp, sap);
// Send off the request to add.
TRYRES(sap->send(areq,recv_cb,0));
```

Note – A callback is created with a static function called `pdm_receive_resp` as the callback function pointer. The callback data is the SAP instance pointer. The send function includes the callback `recv_cb` in the send arguments.

When the MIS completes the request, the `pdm_receive_resp` function is called with two parameters. The first parameter is the SAP and the second is a response handle which is used to receive the message response.

Example 2:

The following example excerpt is from the `msgio.cc` module and illustrates the `pdm_receive_resp` function:

```
pdm_receive_resp(Ptr cbh, Ptr rh)
{
Message *mp;
MessageSAP *p_sap = (MessageSAP *) cbh;

Vtry {
    TTRYPROC(p_sap->receive_response(rh, mp));
    // At this point mp points to the received response
    Message:: delete_message(mp);
}
```

SAPs are initialized by creating SAP pairs. When a SAP pair is created, the `init_kernel_msg_sap`() function is used. The `init_kernel_msg_sap`() function is a utility function which couples two SAPs. Once the SAPs have been coupled, the local SAP must have two callback functions initialized: the receive request callback and the detached callback.

The receive request callback handler is the handler which is invoked when message requests are sent to the SAP. The detach callback handler is invoked when the remote SAP is deleted.

9.2.2 Message Protocol Adaptor Initialization

To initialize a Message Protocol Adaptor, the following steps must occur:

- Create listen port and request SAP
- Connect to the MIS and use extract raw SAP
- Lock the application discriminator

9.2.2.1 Creating the Listen Port and Request SAP

The MIS to MPA communication is a form of CMIP over TCP/IP. The MIS creates and manages associations to an MPA on a per request basis. If an association is not used for a period of time the association is released. If an association is already established it will be reused for subsequent requests. Each request has a request identifier which allows for multiple requests to be outstanding at any one time. To facilitate demand based association establishment, the MPA must allocate an IP listen port. An IP listen port is where the MPA listens for association requests from the MIS.

The utility function `init_mpa`() creates both a SAP and a listen port.

```
init_mpa(portnum, &p_sap);
```

where `portnumber` is a port number for an IP socket, and `p_sap` is the address of a pointer to a `MessageSAP` data type.

9.2.2.2 Connecting to the MIS and Using `get_raw_sap()`

Because the underlying transport mechanism for this SAP might not be active (that is, an underlying association is not established), another SAP must be created to send event reports to the MIS. This other SAP is allocated by creating a `platform` instance which is connected to the MIS. This platform instance gives access to an `ApplMessageSAP`. This SAP is based on another `MessageSAP` implementation. Because the implementation is derived from `MessageSAP`, the `ApplMessageSAP` can be used as a `MessageSAP`. This is possible through C++ inheritance and virtual function mechanisms.

Example:

The following code example illustrates how to create a SAP that sends event reports to the MIS and initialize the resultant SAP to receive requests and disconnects.

```
    host = getenv(“EM_MIS_DEFAULT_HOST”);
    if (!host) {
    host = def_host;
    }

    if (emPlatform.connect(host, “SAMPLE_MPA”) == OK) {
    emPlatform.when(“DISCONNECTED”,
        Callback(pdm_test_handle_detach, 0));
    mis_connected = TRUE ;

    } else
    mis_connected = FALSE ;
    ev_sap = (MessageSAP *) emPlatform.get_raw_sap();
    pdm_test_sap->receive_request_cb.handler =

                pdm_receive_request_msgs;
    pdm_test_sap->receive_request_cb.data = (Ptr) pdm_test_sap;
    pdm_test_sap->detach_cb.handler       =
pdm_test_handle_detach;
    pdm_test_sap->detach_cb.data           = (Ptr) pdm_test_sap;
```

Once the platform instance has been created, the `get_raw_sap()` method returns the SAP which can be used for sending event reports to the MIS.

9.2.2.3 Locking the Application Discriminator

This step must be completed to allow for proper operation of the MPA. It is most important as locking the application discriminator has direct impact on system performance. Every platform instance creates an application object instance within the MIS as long as the platform instance is instantiated. Each application object instance contains a discriminator which forwards all platform notifications to the connected application. The MPA typically does not require this feature.

To disable event forwarding, the application instance discriminator must be locked.

Example:

The following example excerpt from the `samp_main.cc` module in the samples directory illustrates this locking mechanism:

```
DU appinst = emPlatform.get_prop(duAPPLICATION_OBJNAME) ;
Image   app_image(appinst);
if( app_image.get_error_type()!=PMI_SUCCESS ||

!app_image.boot() ||
!app_image.set_str(“administrativeState”,”locked”) ||
!app_image.store() ) {

    printf(“Error starting MPA %s\n”,
            app_image.get_error_string());
    exit(1);

}
```

9.2.3 Protocol Driver Module Initialization

Protocol Driver Modules are shared libraries that are loaded at MIS start-up time. The MIS looks in the `$EM_HOME/config/EM_shared_libs` file for a list of libraries to load. Once it finds an entry in that file, it uses the `dlopen`() system call to load that library. Once the library is loaded, the MIS looks for an instance of the DynLoader class that matches the name of the loaded library. Once the Dynloader instance has been located, the entry point is invoked with a D_LOAD command.

Note – The second parameter of the instance of the declared DynLoader must match the name of the shared library. See dyn_lib.cc for an example.

To initialize a Protocol Driver Module, the following steps must occur:

- Create a kernel message SAP pair
- Register the SAP with the Fully Distinguished Name (FDN) table

9.2.3.1 Creating a Kernel Message SAP

A kernel message SAP pair is created using the init_kern_msg_sap() function.

```
    extern Result init_kernel_msg_sap ( Address , MessageSAP ** );
```

This utility function creates a pair of coupled kernel message SAPs. It returns to the caller a pointer to a local MessageSAP and initializes the remote SAP to be attached to the MRM at the SAP Address supplied. This Address must be the same as the Address supplied in the FDN table configuration step. See the section on “Example of Timer Initialization” for more information on Address Format.

Each request (message) contains a destination address field *dest*. The MRM searches in its list of attached SAPs for an Address match. When the destination address of a request matches a registered SAP, the request is routed over that SAP.

Note – The destination field component is completed by the lookup code within the MIS.

9.2.3.2 Registering an FDN table Entry

The utility function `fdn_register()` in `samp_utils.cc` illustrates how to add an entry to the FDN table.

```
Result fdn_register(MessageSAP *, int, Asn1Value &);
```

The `fdn_register()` function creates a table entry which specifies a mapping between a fully distinguished name (FDN) and a specific address. It takes three parameters to create a table entry:

- A SAP over which to send the `addfdn` ACTION request
- An INT which specifies the SAP TAG (unique identifier for a SAP)
- An encoded distinguished name

Example:

The following example from `dyn_libs.cc` illustrates the creation of a table entry:

```
// Check in em_config for TEST SAP number.
// GETENV is a front end MACRO to EM-config file
// Using EM-config forces SAP numbers to be unique
// if not there use default of defined VAL (64)
const char *p_sap_no;

if ( (p_sap_no = GETENV("TEST_PDM_SAP") ) ) {
        pdm_test_addr.atag = atoi(p_sap_no);
} else
        pdm_test_addr.atag   = MY_PDM_SAP;

pdm_test_addr.aclass = AC_PRIMITIVE;
pdm_test_sap = (MessageSAP *) 0;

// Create a kernel message SAP, this binds us to the
// MRM at SAP class AC_PRIMITIVE and SAP number MY_PDM_SAP
// Whenever a request's DN matches a DN in the FDN table
// that request is forwarded to the SAP that matches
// the Address portion of the FDN Table Entry. In this case
// our Address is AC_PRIMITIVE, MY_PDM_SAP.
```

```
if ( init_kernel_msg_sap ( pdm_test_addr, &pdm_test_sap ) != OK )
        Return(NOT_OK);

// Here is where we set up our request handlers.
pdm_test_sap->receive_request_cb.handler =
                                pdm_receive_request_msgs;
pdm_test_sap->receive_request_cb.data    = (Ptr) pdm_test_sap;
pdm_test_sap->detach_cb.handler          =
                                pdm_test_handle_detach;
pdm_test_sap->detach_cb.data             = (Ptr) pdm_test_sap;
ev_sap = pdm_test_sap;

// This is where we register the PDM DN with the
// fdn table. We register an entry with
// DN = /systemId="pdm" and an address of
// of AC_PRIMITIVE and SAP number MY_PDM_SAP

Asn1Value pdm_dn = my_dn(my_name);

// Lets delete first in case we did not unload
// gracefully.
TRYRES(fdn_unregister(pdm_test_sap, MY_PDM_SAP,pdm_dn));

// Now register should work
TRYRES(fdn_register(pdm_test_sap, MY_PDM_SAP,pdm_dn));
```

See Section 9.3.2, “MPA and PDM Addresses” for information on using EM_config. See Section 9.3.1.1, “Address Classes” for more information on the AC_PRIMITIVE class.

9.3 Routing Messages

9.3.1 How Messages are Routed to the Adaptors

The MIS contains a table which is analogous to the NFS mount table that the UNIX kernel maintains. The table is a complete mapping of the remote MIT. The MRM searches this table for complete or partial matches of the Distinguished Name (DN) for every request. If there is a match, the MRM stores the address information found in the FDN table entry in the destination field of the message and forwards the message to the SAP that matches that address.

To route a message to an adaptor the FDN table must previously have been updated with the DN or DNs that the adaptor is responsible for and an address for the adaptor's SAP.

The FDN table is updated by two action requests: `emAddFdnEntry` and `emRemoveFdnEntry`. These action requests are defined in the *em.gdmo* document. The `fdn_register()` function uses these actions to update the FDN table. The CMIP Configuration utility (`em_cmipconfig`) uses the same actions to achieve the same purpose.

The FDN table entry has two components: the DN and the Address. The address component is based on the address C++ class defined in `address.hh`. An address has three components:

- *class*, type of Address
- *tag*, particular instance of a particular class
- *value*, data associated with that address class

9.3.1.1 Address Classes

Four address classes are defined:

- AC_DEFAULT

  Implies default routing based on type, for example, an event report is always routed to the EMM.

- AC_APP

  Is the address class for SAPs that are attached to applications. Each PMI application results in an Application SAP being created. The address class of each application SAP is AC_APP.

- AC_DIR_SERVICE

  Is reserved for directory service management.

- AC_PRIMITIVE

  Is the address class for PDM's.

9.3.1.2 AC_PRIMITIVE Address Tags (SAP number)

Each Address class uses the tag value to uniquely identify SAP instances. For the purposes of developing PDMs, familiarity with AC_PRIMITIVE tags is critical.

The following is a list of the well known AC_PRIMITIVE tags:

```
#define AT_PRIM_OAM              0
#define AT_PRIM_EMM              1
#define AT_PRIM_CMIP_PRES_ADDR   2
#define AT_PRIM_SNMP_ADDR        3

#define AT_PRIM_AET_ADDR         4     // ASN1: AE-title
#define AT_PRIM_MPA_ADDR         5

#define AT_PRIM_AGENT_DN         6     // ASN1: FDN
#define AT_PRIM_RPC_ADDR         7
#define AT_PRIM_CMIP_CONFIG      8     //
String:{psel,ssel,tsel,nsap}
```

Note – The tags listed above are the well known AC_PRIMITIVE tags that are in use by the MIS system. Providers of new PDM's must chose a value outside of this range.

9.3.1.3 Address Data (aval)

The data field of an address can be used for any purpose. It is a DataUnit which is of variable length. Of particular interest is the aval syntax for the AT_PRIM_CMIP_PRES_ADDR tagged AC_PRIMITIVE class. This contains a complete Presentation Address using the following syntax:

```
length byte, Presentation Selector,
length byte Session Selector,
length byte, Transport Selector,
count byte (number of Network Selectors),
        length byte, Network Selector.....
```

This is a series of octets, one octet of length followed by value octets. A length value of -1 (255) indicates a null selector.

9.3.2 MPA and PDM Addresses

9.3.2.1 PDM Address

A PDM Address is defined to have the following values

- *class*, AC_PRIMITIVE
- *tag*, user specified tag, an integer value outside of the range of the well defined set listed above
- *value*, Data Portion is user definable

Example:

The following example from `dyn_lib.cc` serves as an illustration:

```
// Check in em_config for TEST SAP number.
// Using EM-config forces SAP numbers to be unique
// if not there use default of defined VAL (64)
const char *p_sap_no;

if ( (p_sap_no = GETENV("TEST_PDM_SAP") ) ) {
    pdm_test_addr.atag = atoi(p_sap_no);
} else
    pdm_test_addr.atag   = MY_PDM_SAP;

pdm_test_addr.aclass = AC_PRIMITIVE;
pdm_test_sap = (MessageSAP *) 0;
```

Note – A convention exists to ensure that user-supplied PDMs use conflicting PDM SAP numbers. This convention is illustrated above. The convention consists of specifying the PDM number in the $RUNTIME/conf/EM-config file which can be extracted using the GETENV macro.

9.3.2.2 *MPA Addresses*

MPAs have one more level of indirection than PDMs. The additional level allows the MPA to be a separate process that can be run anywhere in the TCP/IP domain. The MPA addresses are defined as:

- *class*, AC_PRIMITVE
- *tag*, AT_PRIM_MPA_ADDR.

The developer need not worry about configuring an MPA Address programmatically since the Address for an MPA is configured using the `em_cmipconfig` utility. The `em_cmipconfig` utility creates an address where the aval portion contains all the relevant configuration information. The MIS extracts this information and presents it to the MPA using the `remote_oi` and `remote` fields of the message request sent to the MPA request routine.

The following fields need to be configured for a custom MPA:

- Entity Name
- *Custom MPA*: The port and hostname fields must be completed.
- *Session Selector*
- *Network SAP*: The Network address of the entity containing the real object should be included.
- FDN

Note – If the Presentation address of the remote entity has a null session selector, the session selector must be configured with some default value (for example, “MPA”).

The example MPA supplied used the following configuration:

```
Entity Name : { 1 2 3 4 5 1 }
Custom MPA  port : 5597
Custom MPA host: “carla”

Session Selector : “Test”
Network SAP : carla:5597
FDN : /pdmId=string:”testMPA”
```

Once the em_cmipconfig configuration has been completed, the MRM attempts to match entries against what has been configured in the FDN table using em_cmipconfig.

Requests for multiple entities can be directed to a custom MPA. The sample source supplied only supports one entity /pdmId="testMPA". A custom MPA can be created to support multiple entities.

The CMIP MPA provided with the MIS is an example of an MPA that is capable of supporting multiple entities. Each CMIP agent in the MIS' management domain is viewed as an entity. For each CMIP agent to be managed, the user must run em_cmipconfig specifying the agent parameters, in particular, the agent's presentation address.

9.3.2.3 *Message* remote_oi *and* remote *Fields*

The values in the remote_oi and remote fields are one of the specific differences between an MPA and a PDM.

The MPA can make explicit use of these fields. The remote_oi contains the DN of the CMIP table entry that was used to route the request to the MPA. The last RDN of this DN can be used to find the AE-Title (Application Entity Table). The remote field contains whatever was configured for that entity's presentation address. This information is what the CMIP MPA uses to establish an association with a remote entity. The aval portion of the remote field contains a string of the format:

```
"{ Pres, Sess, Tsel, Net }"
```

This is an ASCII representation of the remote entities presentation address as configured using em_cmipconfig. See the "Address Data (aval)" section for more information.

Note – A PDM can not make use of the remote_oi or the remote field.

Multiple FDN table entries can be stored which point to a single MPA or PDM. This storage mechanism can be used as a persistent configuration store for each entity supported by the MPA / PDM. This mechanism, in conjunction with the remote_oi and remote fields, allows for easy multiplexing to the real object information.

9.3.3 FDN Table Configuration Options

There are multiple ways to configure the MIS to route messages to adaptors. In deciding which configuration to use, it is important to implement a model view that is appropriate for the problem being solved. Common sense can be a good start. For example, it would not be appropriate to add thousands of entries to the FDN table when the MPA could easily implement an efficient proprietary lookup based on the DN.

To help clarify the process, the following options are examined:

- MPA supporting two remote objects
- PDM supporting two remote objects

These examples are used to indicate the various options open to the developer.

Note – These examples are not the only options and are used for illustrative purposes only.

9.3.3.1 MPA Supporting Two Remote Objects

Two possible strategies for adding entries to the FDN table are:

- Two entries can be added to the FDN table using `em_cmipconfig` with two different entity names specified.
- Two entries can be added to the FDN table using `em_cmipconfig` with only one entity name specified.

The second case illustrates how `em_cmipconfig` allows one entity to support multiple objects. It is assumed that the remote entity that supports these objects is identically addressed, that is, the objects live at the same presentation address.

9.3.3.2 PDM Supporting Two Remote Objects

Two possible strategies for adding entries to the FDN table are:

- Two entries can be added to the FDN table with an identical PDM address.
- Two entries can be added to the FDN table with identical class and tag values in the address with different data value in the address value field.

The first case assumes that the DN is decoded in the request and that the request is routed to the appropriate agent entity.

The second case allows the data that is configured in the value (`aval`) portion of the `dest` field of the message to be used for achieving multiplexing to the remote agent entity.

9.3.4 Source and Destination Fields in the Message

All messages contain `src` and `dest` fields. These fields are C++ instances of Address and are very important for message routing. **The `src` field must be completed for all CONFIRMED requests.** The MPA / PDM developer must complete the `src` address if responses are expected. The `src` field is the same address used when creating the SAP and when specifying the address component of the FDN table registry entry.

The `dest` field is only important if you wish to explicitly route the message.

Note – Since explicit routing is used for passing messages between applications, it should be used carefully.

9.4 MPA/PDM Request Management

The MIS has been designed to be a multi-user server and can handle multiple requests transparently and asynchronously. The SAP interface has been designed to make this asynchronous style of programming easier.

In Figure 9-2, a scenario is illustrated that is typical of a real world environment.

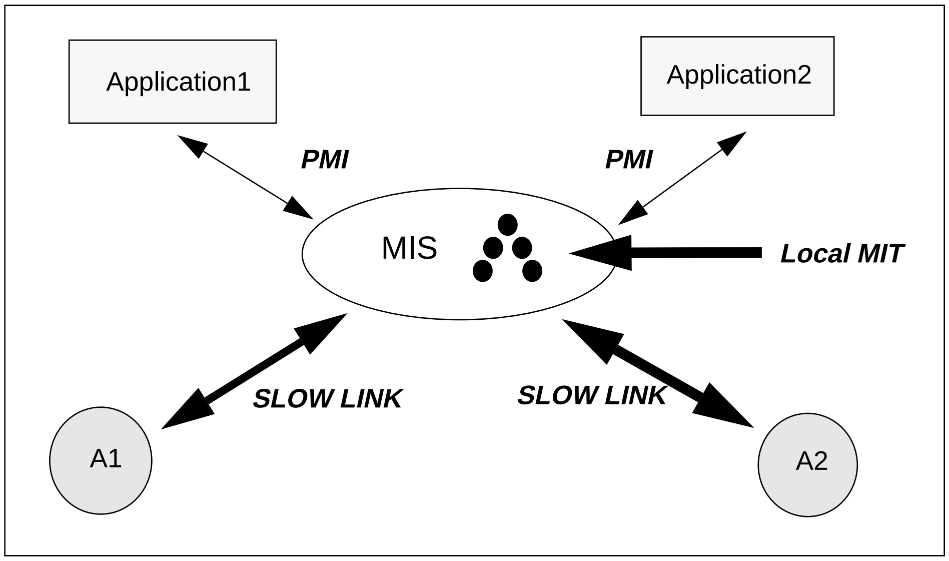


Figure 9-2 Potential Real World Configuration

Application1 makes continuous requests to the local MIT. These requests can be satisfied immediately.

Application2 makes requests to A1 and A2. These agents are at the remote end of very slow links. The architecture must support the scheduling of responses to requests that could take a long time to complete. The design and implementation should also handle cases where requests overlap. If appropriate, multiple overlapping requests should be queued and serviced with a single response.

The MPA / PDM module serving A1 and A2 schedules its responses. It achieves this by using the underlying PMI scheduling services.

The transport mechanism used by the MPA / PDM is normally hidden behind a UNIX file descriptor. The MPA / PDM developer needs to use the underlying scheduling services to interface to the file descriptor. This scheduling service makes use of the underlying *poll* (*select*) system call to determine the following:

- Whether data is available at the file descriptor

- Whether data can be written to the file descriptor
- Whether there is an underlying exception on the file descriptor

To satisfy remote requests, the MPA / PDM code typically opens a file descriptor to the agent. As requests are made to the agent, they are written to the file descriptor. The MPA / PDM code then schedules a callback for whenever data becomes available at the file descriptor. When the data is completely transferred, the MPA / PDM sends a response back to the MIS.

At a high level, the steps can be broken down as shown in Table 9-2.:

Table 9-2 MIS and MPA / PDM Connections

MIS	**MPA/PDM**
Request for MIS	
	Create a connection to remote agent.
	Schedule a callback to indicate connect complete.
	Send request with unique identifier.
	Schedule a callback for response.
	When data callback, check for data complete.
	If data complete, send response to MIS.
MIS receives Response	

Note – Because all requests have identifiers, it is easy to match responses to requests.

9.4.1 *Asynchronous Request Code Specifics*

When managing asynchronous requests, familiarity with some of the underlying C++ classes that facilitate asynchronous request management is critical. The most important C++ class is the `Callback` class. This is used by all of the asynchronous interfaces. There is also a C++ programming style that must be understood. The proposed C++ model is that a C++ class must be built with a class implementation with the following characteristics:

- Original request information, particularly the request id, the src, and dest fields must be stored.

- A static member function must exist that can be used as a callback handler.
- A variable must exist than can be used to count the number of callbacks expected.
- Class instances must be easily found by the request identifier so that CANCEL GET requests can be serviced in a timely manner.
- Responses must be queued for Synchronous requests.

These characteristics determine what can be called a pending request class. This type of class encompasses the functionality required for responding to requests asynchronously.

Example:

The following example excerpt from a class called `pdm_pend_req` in `samp_inc.hh` illustrates some key elements associated with asynchronous requests:

```
class pdm_pend_req : public QueueElem {
    ObjReqMess *orig_msg;
    MessId   req_id;
public:

    // hash of all pending requests based on request id
    Hashdeclare(MessId, pdm_pend_req, hash_MessId,
MessIdcmp,0,0)
    static Hash(MessId, pdm_pend_req) *p_pdm_req_hash;
    I32cbs_pending;
    I32num_in_scope;

    ReqStatus status;
    Result start_req(Callback &req_done);
    // Used in cases where atomic operation requested.
    Queue(RespQElement) resp_q;
}
```

The `pdm_pend_req` class has the following characteristics:

- Maintains a copy of the original request and request id
- Has a callback counter
- Can be queued

- Can be hashed based on a `MessId` type
- Has an asynchronous interface to a `start_req` routine
- Has a callback to be used for invoking operation completions

The function of this class is to remember original request information, to start requests, and to ensure that a final response is sent if required. Requests are completed when there are no more callbacks pending, that is, `cbs_pending == 0`.

The `req_mngt.cc` module supplied in the samples directory illustrates all of these key concepts. The `req_mngt.cc` starts requests, counts the callbacks, and releases resources that are no longer required.

9.4.2 Validating Requests

Each request can be validated. The extent of the validation is dependent on the implementor and the specifics of the implementation. In some cases, it will make no sense to validate requests since much of the intelligence to validate is on the remote agent. The CMIP MPA supplied does little more than store request identifiers and forward the requests to the CMIP agent. The sample source, which is supplied, does some validation for illustrative purposes.

The typical items that can be validated up front are object class and operation types. For illustrations of typical items than can be validated, see the code in `msgio.cc` particularly `pdm_verify_oc()` and `pdm_verify_dn().`

9.4.3 Matching Requests to Responses

Every request is identified with an identifier. To send a response to a particular request, set the response message id to the request id. For completeness the src and dest fields must be completed also.

Example:

The following example excerpt from msgio.cc send_resp. rmp is a pointer to a response message. msg is a pointer to the original request.

```
    rmp->id = msg->id;
    rmp->source = msg->dest;
    rmp->dest = msg->source;
    rmp->qos = msg->qos;
```

9.5 Timer Management

The following timer management services are available to the developer:

- Create and delete a timer
- Start a timer
- Stop a timer

Timers are very useful in the MPA / PDM environment. In particular, they are needed to implement time out strategies. Every request requires a response. In the real world, there are occasions when devices do not respond. The timer facilities are useful to set a timeout callback which can send an TIMED_OUT response to the MIS in these types of situations.

Note – MPAs cannot use TIMED_OUT, DEST_UNREACH or NO_SUCH_DEST messages. PDMs may use these message to signify errors. For error conditions like these, the MPA should return a PROC_FALL message with a Probable Cause specific error which is defined by the implementor.

Timers can also be used to check on device status. If the device status has changed, the MPA / PDM can emit an attribute change notification. Applications can then be written to wait for these notifications and to operate on an event driven basis. Asynchronous applications (applications that do not POLL or that are event driven) have a positive impact on overall system performance.

The MPA / PDM developer should always attempt to use a notification based mechanism to communicate to applications. Timers can be used to schedule these specific notifications. Using GDMO, any type of notification can be designed. Once the GDMO syntax has been loaded into the Meta Data Repository (MDR), the MPA / PDM can emit the notification based on the behavior defined in the GDMO.

9.5.1 Timer Management Interface

The timer interface consist of a set of functions which allow the scheduling of callback handlers based on criteria specified in instances of a timer C++ class. These timer instances specify the following:

- A callback handler
- An interval specifying expiration time before the handler is invoked
- A reload interval value load after the original interval expires

---

Note – These timers are not suitable for real time solutions.

The class definition for timer is defined in `sched.hh` and is listed below as well as the interface functions to the scheduler.

```
class Timer {
public:
    MTime       time;           // expiration time in milliseconds
    MTime        reload;          // reload time after expiration
    Callback     cb;              // callback to post when expired

    Timer() {
    time = 0;
    reload = 0;


    }
    Timer(MTime t, MTime re, CallbackHandler hand, Ptr d) {
    time = t;
    reload = re;
    cb.handler = hand;
    cb.data = d;
    }
    friend int operator==(const Timer &t1, const Timer &t2) {
    return t1.cb == t2.cb;
    }
};

void    post_timer(const Timer &);
    // Post an timer into the scheduler queue

void    purge_timer(const Timer &);
    // Purge any matching timers from the scheduler queue

void    purge_timer_handler(CallbackHandler handler);
    // Purge any timers with matching handler

void    purge_timer_data(Ptr data);
    // Purge any timers with matching data
```

The four functions detailed above provide the interface to the scheduler for enabling and disabling callbacks based on time. The user can choose a purge interface suitable for the implementation.

9.5.1.1 *Example of Timer Initialization*

The following example excerpt `start_timer`() from `rusagobj.cc` uses the `post_timer`() scheduling function to start a timer. The timer that is passed to `post_timer`() is an instance of the C++ class timer.

```
    void start_timer() {
    // Timer Input is in milli secs
    // parms are, timer interval, timer reload value
    // Handler and parm passed to handler

    post_timer(Timer(timerval * 1000, timerval * 1000,
        (CallbackHandler) pdmrusage_timer, (Ptr) this));
    timer_posted = TRUE;
    }
```

The timer constructor takes three parameters

- Timer intervals

- Timer reload interval value
- Callback to be executed

All interval values are specified in milliseconds. The timer instance created above is based on a variable, `timerval`, stored in the `rusageobj` instance. This value is in seconds and is converted to milliseconds by * 1000. The implementation requires that the timer be continuous so a reload value identical to the initial interval is passed for the reload parameter. The callback parameter is passed containing `pdmrusage_timer`() as the function to be called and the rusageobj instance pointer is passed as the Callback user data.

Note – The implementation of `post_timer`() makes a copy of the timer and Callback parameters passed. It is okay to use timers and Callbacks that get destructed

9.5.2 *Stopping a Timer*

Timers are stopped or purged using the `purge_timer`(), `purge_timer_handler`() or `purge_timer_data`() utility functions. The `stop_timer`() method in `rusageobj.cc`, listed below uses the `purge_timer_handler`() function to cancel the `pdmrusage_timer` callback when it is no longer needed, that is, when the *runtime* attribute is set to 0.

```
void stop_timer() {
purge_timer_handler((CallbackHandler) pdmrusage_timer);
timer_posted = FALSE;
}
```

`purge_timer_handler`() searches the list of active timers for any timer that has a callback handler equal to the value passed. It then purges any timers matching from the timer queue.

Note – It is important to remember if timers have been posted. A common error is that users forget to purge timers. The callback then tries to use data that has been deallocated.

9.6 File Descriptor Management

The normal mechanism of communicating outside of a UNIX process is through a UNIX file descriptor. The services provided by the scheduler to interface to file descriptors are important to understand because most MPA / PDMs use file descriptors to communicate to remote devices.

9.6.1 Asynchronous File IO

It is important that the MPA / PDM never blocks (makes a system call that does not return immediately). The underlying UNIX File IO system allows file descriptors to be opened or created in NON-BLOCKING modes. It is most important that any file descriptors be opened in a non-blocking mode.

The most important aspect of non-blocking file IO is that some operations may not complete. The code must handle properly partial reads / writes. By using the underlying operating *poll* system call (*select* on BSD systems), the scheduler facilitates non-blocking IO by providing a set of utility functions to allow for callbacks when particular events happen at a file descriptor. These are based on the standard UNIX set of:

- Data can be read
- Data can be written
- Error or exception

Whenever a user expects to read from a file descriptor the user should use `post_fd_read_callback`(). If data needs to be scheduled to be written, the user should use `post_fd_write_callback`(). To handle error cases there should be a handler for exceptions. This handler can be set using `post_fd_except_callback`().

Callback handlers as discussed above are static functions that are invoked when the event they have been scheduled to service occurs. The file management functions are passed the file descriptor they are managing as parameter two. Parameter one is the data specified when the CallBack was created. Oftentimes parameter one is a pointer to an C++ instance that contains the file descriptor and any status associated with it.

The complete set of function prototypes as defined in `sched.hh` are listed below:

```
void    post_fd_read_callback(int fd, const Callback &cb);
void    post_fd_write_callback(int fd, const Callback &cb);
void    post_fd_except_callback(int fd, const Callback &cb);
void    purge_fd_read_callback(int fd);
void    purge_fd_write_callback(int fd);
void    purge_fd_except_callback(int fd);
void    purge_fd_callbacks(int fd);
```

Note – File descriptor callbacks should be purged if the instance associated with the callback is deleted.

9.6.2 *Example of a Read Callback Implementation*

The example below is taken from the `unixobj.cc` sample source code. There are two items of importance:

- Scheduling the callback
- Callback execution

9.6.2.1 *Scheduling the Callback*

The following example is taken from the unxobj.cc `start_get_req`() routine. Once the pipe has been successfully opened, the code creates a callback and schedules the callback function execution when there is data available from the pipe. The read callback handler is scheduled using the `post_fd_read_callback`() routine.

The following should be noted:

- The FPTR (`fp`) is converted to a file descriptor using `fileno`. (See the man pages for more information.)

- The callback which is created is initialized to have a function pointer `sh_fetch_input` and a data pointer of this. (This is C++ *this* for instance being operated on).

```
// Set up to get called back when Data is
// available from the pipe.
Callback rd_cb((CallbackHandler)sh_fetch_input, this);
post_fd_read_callback(fileno(fp),rd_cb);
```

The routine sh_fetch_input is invoked wherever there is data available at the file descriptor `fileno`(`fp`).

9.6.2.2 *Callback Execution*

The code below is excerpted from `unixobj.cc` `sh_fetch_input`().

The `sh_fetch_input` routine does the following:

- Uses parameter one as a `pdmunixOi` pointer since that is what was defined as the user data parameter when the callback was scheduled.
- Accesses the file descriptor from the instance pointer where it was stored.
- Allocates a temporary dataunit to store the data read.
- Reads the data until there is no more data (for example, EOF). When that happens, it calls the `get_complete`() method of the `pdmunixOi` C++ instance.
- If there is more data, ***it reposts the callback*** having first saved the data read in `sh_data`. The dataunit catenate is used to add any new data to the end of the old data.

Example:

```
static void sh_fetch_input(pdmunixOi *p_obj, int)
{
    int rv;
    DataUnit tmp(RSIZ);

    // Read the data
    if ( !(rv = fread((char *)(const Octet *) tmp,1,
                       RSIZ,p_obj->fp))) {

    // End of pipe, need to encode, save for Persistence
    // and then call the requestor get_complete routine.
    pclose(p_obj->fp);
    p_obj->fp = 0;
    p_obj->get_complete();
    return;

    }
    // Store it into the sh_data Dataunit, just the correct amount
    DataUnit rdata(rv);
    memcpy((void *)(const Octet *) rdata, (const Octet *) tmp,
rv);

    // Need to get rid of old data, so we do not keep growing
sh_data
    if ( p_obj->firstdata == TRUE ) {
    p_obj->firstdata = FALSE;
    p_obj->sh_data = DataUnit();
    }
    p_obj->sh_data = catenate(p_obj->sh_data,rdata);

    // Need to get Called Back again so reschedule
    post_fd_read_callback(fileno(p_obj->fp),
    Callback((CallbackHandler)sh_fetch_input, p_obj));
}
```

This is a very complete example of a read callback handler since it remembers state information (data just read) and reschedules itself.

Note – The callback must be rescheduled if more data is to be read. A common error is the omissions of reposting the callback when the data transfer has not been completed.

9.7 Notifications

Notifications are based on the OSI Management Event Reporting function. All notifications are instances of the CMIP Event Report Request. They contain:

- Object Instance
- Object Class
- Event Time
- Event Type
- Event Information

The ASN.1 syntax for a notification is defined in x711.asn and is listed below:

```
EventReportArgument ::= SEQUENCE {
    managedObjectClass     ObjectClass,
    managedObjectInstance   ObjectInstance,

    eventTime              [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventType               EventTypeId,
    eventInfo              [8] ANY DEFINED BY eventType OPTIONAL
}
```

Note – The event information field is an ASN.1 any defined by value, which is defined by the Event Type field. To generate a notification, the eventInfo must first be constructed according to the syntax defined for that specific notification.

Notifications can be confirmed or unconfirmed. Typically notifications are unconfirmed.

Notifications are generated based on the behavior of the object class being modelled. The standard notification set includes ObjectCreation, ObjectDeletion, and attributeValueChange notifications. New notification types can be defined that are specific to the model being presented to the applications. These new notification types and syntaxes must be loaded into the Meta Data Repository (MDR) using the `em_gdmo` and `em_asn1` utilities.

9.7.1 Creating a Notification

Notifications are created by:

- Allocating an event report request message
- Filling in the message fields
- Sending the completed messages to the MIS

These steps are illustrated in the `pdm_issue_notif`() routine in `msgio.cc`. Excerpts from that routine are used below:

9.7.1.1 Allocating an Event Report Message

Messages are allocated using the `new_message` function. An example of how they are allocated follows:

```
// Allocate Event Report Message
if ((mp = (EventReq *)Message::new_message(EVENT_REPORT_REQ))
    == NULL)

{
    pdm_test_error.print(“pdm_issue_notif:
    Not enough memory for “
    “M-Event-Report message\n”);
Return(NOT_OK);
}
```

9.7.1.2 *Filling in the Event Report Message Fields*

The time and info fields are optional and only need to be competed if the syntax demands that they be completed. All other fields must be filled in. Each of the fields require an encoded Asn1Value:

```
mp->mode = UNCONFIRMED;
mp->oc = oc;
mp->oi = oi;

TRYRES(event.encode_oid(TAG_CONT(6), event_type));
mp->event_type = event;
mp->event_info = info;
getGeneralizedTime(mp->event_time);
```

Each field has a specific encoding. This must be adhered to as defined by the definition of an EventReport. See the “Notifications” section for information on syntax. The object class field needs to be encoded TAG CONTEXT 0. The OI needs to be defined according to the specific encoding rules for ObjectInstance. See the x711.asn1 documentation for more information about ObjectInstance.

9.7.1.3 *Sending a Notification*

All messages are sent to the MIS over an initialized SAP.

```
// Send the Message to the MIS
if (ev_sap->send(mp) != SENT)

{
    // Major System Error
    Message::delete_message(mp);

    pdm_test_error.print(“pdm_issue_notif: Could not send “
    “M-Event-Report message\n”);
Return(NOT_OK);
}
```

Unconfirmed event report request messages are sent using the `send()` function. Confirmed event report requests also use the `send()` function, however, it needs the asynchronous version.

```
SendResult send(MessagePtr mp, const Callback &cb, MTime cd
block_time)
```

The MIS forwards all event report requests to the Event Monitor Module (EMM) where it is discriminated (filtered). The notification is then forwarded to applications that have registered for it.

This routine accepts an already encoded Info parameter. For a complex example of encoding an eventInfo structure, see `pdm_make_attr_chginfo()` in `samp_utils.cc`.

9.8 Sample MPA/PDM Source Code

The source code example provided is meant for illustrative and educational purposes. Much of the example code would not be used by a standard MPA / PDM. The sample source provides additional Logical Object Services that would normally be provided in the remote agent. To avoid dependencies on a specific remote agent, simple Logical Object Services are included in the sample source.

Note – All of the sample source code should be studied in detail.

9.8.1 Files and Configuration

The sample source and the GDMO and ASN.1 files are included in the mpa_samples directory. There is also a set of netperl scripts which can be used to illustrate the functionality of the MPA / PDM.

Note – The GDMO and ASN.1 files must be loaded before any of the example code is used.

The MPA / PDM sample source consists of the files listed in Table 9-3:

Table 9-3 MPA Example Files

Filename	**Description**
Makefile	contains rules
dyn_lib.cc	Portion of Code to create dyn lib entry point and initialization
samp_main.cc	Generates a main for the testmpa program. Attaches to MIS and initializes the MPA event sap.
dynload.hh	Needed by for DynLoader Instance
msgio.cc	Handles message IO from the MIS.
req_mngt.cc	Manages the Requests
samp_inc.hh	Includes and definitions
samp_utils.cc	Utility functions
lroot.cc	Sample logical Root Object
rusageobj.cc	Sample object which reads / proc and generates rusage info. Can be configured to send Attribute Change notifications.
unixobj.cc	Sample that uses the unix popen command to execute unix commands. Illustrates Asyc File IO and Object Create Notifications.

The makefile can be used to generate to files, testmpa and testpdm.so. The testmap is an executable and testpdm.so is a shared library.

9.8.1.1 *Sample MPA Configuration: testmpa*

Testmpa is an executable that binds to 5597 on whatever machine it is run on. It manages a logical MIT that begins at `/pdmId=”testMPA”`. `em_cmipconfig` must be run to address this MPA. The MPA has a DN of `/pdmId=”testMPA”;` it is a CUSTOM MPA that lives at a default port of 5597. You can choose whatever machine name that is needed. Be sure to include a session selector of “test” when using `em_cmipconfg`.

Note – The port number can be overridden by using the environment variable TEST_MPA_DEFAULT_PORT. Make sure that this port is entered in `em_cmipconfig` when configuring the MPA.

9.8.1.2 *Sample PDM Configuration: testpdm.so*

Testpdm.so is a shared library that can be loaded at platform start-up time. To load testpdm.so, edit $EM_HOME/config/EM_shared_libs. There are two methods for loading testpdm.so:

- Provide a complete pathname in the file

  To do so, include the complete pathname of the shared library in EM_shared_libs, e.g. "/opt/ger/pdmsrc/testpdm.so".

- Provide the library name if the library will be placed in $EM_HOME/lib

  To do so, include the libname and place the shared library in $EM_HOME/lib.

The PDM manages a logical MIT that begins at `/pdmId="testPDM"`, it configures itself to be attached to the MRM at an AC_PRIMITIVE SAP type with a SAP number of 64.

Note – PDM SAP numbers can be chosen by the implementor.

To allow for multiple PDMs to exist, this SAP number should be configurable and readable from a file. By convention and default, SAP tag numbers are stored in var/opt/SUNWconn/em/conf/EM-config. The format is Name : Value. For example:

```
TEST_PDM_SAP :  64
```

could be added to the file. This entry can be read using GETENV("TEST_PDM_SAP"). See the source example in `dyn_lib.cc`.

9.9 Steps to Develop an Adaptor

To develop a new adaptor, use the following procedure:

1. Define the management information model
2. Design and implement the request management interface
3. Design and implement the protocol code

9.9.1 Defining the Management Information Model

The information model presented to the MIS must be defined in GDMO. If the existing agent already has some GDMO definitions, the task may be easier. In some cases, the existing GDMO definitions may not be adequately abstracted and therefore, will not be suitable. If the existing GDMO model does not provide a sufficient level of abstraction, new definitions should be defined. The GDMO model defined should make every attempt to fully abstract the management problem being solved.

For example, in a case where a device has 5000 ports, the following activities could be defined:

- The device could be defined as one single object with a set of attributes and ACTIONS that could be used to access the ports.
- Each port could be defined as an object with multiple attributes.

The model that chosen is dependent on the problem being solved. In most cases, the more abstract models can provide for less complex development and better performance for the most common operations. Models can also be optimized to enhance the solution.

The preferred solution in the above scenario would be the single object view. This view would be less complex to implement and require less code. Since there is only one logical object to manage, the overhead in maintaining a logical tree would be minimal and the performance may be better. If the object was defined properly, the applications using the model would also be simpler. They would not have to incur the overhead of managing thousands of objects.

There are no specific rules that can be applied here. Common sense and a clear understanding of the *real* problem and *specific* solution required by the customer are the best guides.

9.9.2 The Request Management Interface

As outlined in the sample source, the interface to the MIS must be completely asynchronous. The interface code must be capable of managing multiple outstanding requests. In addition, it must have minimal impact on the overall system performance. This is critical in the case of the PDM. The request interface can be modeled on the sample source and the `msgio.cc` and `req_mngt.cc` modules can provide the basis for any adaptor.

9.9.3 The Protocol Code

Each device or remote entity must support some type of remote access. Oftentimes there are existing libraries that have already implemented a suitable protocol interface to the devices which are to be managed. These interfaces should be reused as much as possible. The most important consideration when reusing existing protocol stack code is that there be no underlying interface element that could block. All code must be asynchronous. In the case where only synchronous interfaces are provided, a layer needs to be built that provides the asynchronous interface.

Topology Database Service 10

10.1 Summary

Topology Database Service is a module in the MIS that provides topology database functions that can be accessed by the PMI. In this release, the Topology Database Service supports the following new features:

- Alarm Service control
- Device Management

Note – This release replaces the old toponodeDefaultMO and SNMP/RPC MO attributes with the toponodeMOSet attribute.

10.2 Topology Database Service and GDMO

The Topology Database Service is based on the GDMO, whose syntax is in two parts: the formal and the informal. Informal syntax refers to those specified in the “behavior” section of each object. In other words, it is free text, synonymous to behavior. The formal syntax is checked by the GDMO compiler.

The MIS implements the formal syntax. But because the MIS cannot understand the informal syntax, the users must implement it. A secretary is the manifestation of the informal syntax (behavior); this gives rise to the name “object behavior API.”

For example, the following GDMO document contains both formal and informal syntax.

```
topoNodeIncreaseAlarmCount ACTION
    BEHAVIOUR topoNodeIncreaseCountBehaviour BEHAVIOUR DEFINED AS
        !This action increase the counter attribute by 1.  The
        topoNodeSeverity will be updated to the highest severity
        of uncleared alarms, according to the values of severity
        counters.!;
    ;
    WITH INFORMATION SYNTAX EM-TOPO-ASN1.TopoNodeCounter;
    REGISTERED AS { em-topo-action 5 };
```

The formal syntax is provided by the MIS to parse and check the input and output parameters and register the action OID. The informal part (BEHAVIOUR) is implemented in the Topology Database Service to propagate the severity.

10.3 Configuring Topology Nodes for Alarm Management

Topology nodes are created by users to logically model managed objects in their management environments. Each topology node has an attribute topoNodeMOSet, which points to the managed object instances (MOIs) this topology node represents. These MOIs can be SNMP agent objects, RPC agent objects, CMIP agent objects, or any object in the management information tree (MIT).

The topology node can be thought of as a logical placeholder, representing the state of the managed object or MOI. When an alarm comes into the platform, it is against an MOI, not a topology node. A new service called the Alarm Service automatically does a mapping from the MOI to the topology node that has the equivalent MOI in its topoNodeMOSet attribute. When the Alarm Service finds a match, it sets the state of the topology node to the highest severity in the list of alarms against that topology node. For detailed information on the Alarm Service, see the Alarm Service chapter in the *Solstice Enterprise Manager Reference Manual*. An alarm can be generated against an MOI in several ways.

1. An agent or application outside the EM platform detects some exceptional (fault or failure) condition in a managed object and emits an alarm with the specific MOI.
2. A NerveCenter template, which is based on a certain set of states and conditions, can post an alarm against a specific MOI using one of the NerveCenter Request Condition Language (RCL) functions.

Note – Request Designer is one of the EM core applications used for creating NerveCenter request templates. This application allows users to graphically build up an event/poll driven state machine that can be used for resource management. For additional information on the Request Designer, see the *Solstice Enterprise Manager Reference Manual*.

When a topology node can represent multiple managed objects, we need to store the multiple MOIs. This is best illustrated in the case of a link-type topology node. Because a link has two endpoints, the topology node must have a place to store two MOIs, one for each endpoint of the link.

The topoNode object class has an attribute topoNodeMOSet, defined as a set of ObjectInstance (a set of MOIs). There is no limit to the number of MOIs a topology node can represent.

10.4 How Alarm Management Works

Alarm Service maps the alarm condition from MOIs to topology nodes, based on the topoNodeMOSet attribute. When an alarm event is received by the MIS, the alarm event is stored as an alarm log record, based on the alarm log discriminator. (Refer to Alarm Management chapter in the *Solstice Enterprise Manager Reference Manual*.) The alarm service is notified with the creation of an alarm log record. It maps the managed object instance value to the corresponding topology node and updates the alarm counters based on severity. This triggers the severity propagation of topoNode in two ways:

1. Propagate to parents, according to topoNodeParents. (You can have multiple parents,) This applies if topoNodePropagateUp is set to true.
2. Propagate to peers, according to attribute topoNodePropagatePeers.

In addition, the Alarm Service keeps the topoNodeSeverity synchronized so that it represents the highest (most critical) value assigned to it by all uncleared alarm log records that are against this topology node. When there are no longer any uncleared alarms posted against a topology node, then the topoNodeSeverity returns to its “normal” value of cleared.

10.4.1 Discover

Discover creates topology nodes for each device that it finds on the network, such as hosts, routers, networks, and links. Discover also updates attributes like the toponodeMOSet to enable actual management of these devices and their interfaces. In addition, the toponodeUserdata is updated with some of the actual interface and/or ip-address information.

When a network is monitored by Discover, a communications alarm can be issued (see the section on monitoring alarms in the *Solstice Enterprise Manager Reference Manual*) when a host is unreachable.

Discover has a mapping file that allows the user to map information discovered to topology-specific data. For example, if Discover finds a device that has the SNMP sysLocation filled in, the user might want sysLocation to be part of the Host topology node.

Since topology nodes now store vital information such as interfaces, physical addresses, and multiple IP addresses, Discover does some simple link management. Discover periodically runs through the topology database and queries all the links and interfaces for reachability. If it finds one down, it generates a communications alarm. When a downed interface comes back up it generates a communications alarm with perceivedSeverity set to cleared.

Discover periodically retrieves the IpAddress table and IfTable from a host or router and checks for changes in physical and IP address mappings.

10.4.2 Generic Configuration of Complex Devices in the Topology Database

Topology database configuration involves configuration of complex devices such as routers and links into the topology database to enable effective management. Discover updates a special attribute (topoNodeMOSet) to contain references to the interfaces of the Router or Link.

This attribute enables other components (Auto Manager, Alarm Service, and NerveCenter) to use this attribute in order to reference interfaces for configuration and fault management.

The topology database allows users to model their management environment by creating topology nodes with specific topology types, such as host, router, and device. Each topology node is distinguishable from the next by its name, type, and location in the topology hierarchy. If a topology node has other distinguishing characteristics, the only way to configure or show these characteristics is by querying the underlying managed objects that this topology node represents. This solution is not optimal, since managed objects usually reside in remote agents and the data in question is usually static. A few examples of topology nodes are:

- Routers with multiple interfaces
- Switches with multiple circuits
- Topology nodes that represent a set of applications

Topology nodes have some generic user-configurable method for storing device specific data. This not only allows users to add useful data to a topology node, but it also allows vendors to add their own topology types and their own user data fields. topoNodeUserData is used as a generic placeholder for specific topology type data.

The GDMO and ASN.1 have been defined for any topology type that uses the topoNodeUserData attribute. Not all topology types defined are required to use this attribute. By default the value of this attribute is null.

The topoType attribute topoTypeUserDataAttrs contains the GDMO attribute-to-topology type mapping.

For each topoType, the topoTypeUserDataAttrs is defined to have syntax of `SET OF AttributeId`. For example, the topoType Router has the value

```
'{globalForm:"EM Topology":topoRouterData}'
```

Link has the value:

`'{globalForm:"EM Topology":topoLinkData}'`

Any tool that creates a topology node or sets the topoNodeUserData attribute through the PMI consults this attribute to get the syntax of the topoNodeUserData attribute. This syntax must be used when setting the attribute. The syntax for a GDMO attribute can be retrieved by the PMI Syntax class.

Because the TopoNodeUserData attribute is defined in ASN.1 as a `SET OF SEQUENCE` construct, it is possible to have a set of more than one ASN.1 type of data. For example, in the Router entry above, two attribute syntaxes present linkData and myData. When multiple GDMO parameters are present for a topology type, both attributes must be present in the topoNodeUserData attribute when created or set

The topoNodeUserData attribute in the topoNode class is of the ASN.1 type.

```
TopoNodeUserData::= CHOICE {
    null NULL,
    value SET OF AttributeValueAssertion
}
```

The AttributeValueAssertion is defined as:

```
AttributeValueAssertion::= SEQUENCE{
    attributeId OBJECT IDENTIFIER,
    attributeValue ANY DEFINED BY attributeId
}
```

In order to use an `ANY DEFINED BY` construct in the EM platform the user must define both the ASN.1 and GDMO documents. ASN.1 must be defined for the specific data and the GDMO attribute template must reference that ASN.1 syntax. Following is a sample GDMO attribute template definition referencing a LinkPair ASN.1 construct.

```
-- GDMO--
linkData ATTRIBUTE
    WITH ATTRIBUTE SYNTAX Link-ASN1.LinkPair;
    MATCHES FOR EQUALITY;
    BEHAVIOUR linkDataBehaviour BEHAVIOUR DEFINED AS
    !This attribute contains linkname address pairs for Routers!;
REGISTERED AS { 1 2 3 4 5 6 7 5 };

--ASN.1--

        Link ::= SEQUENCE {
            linkname GraphicString,
            address  GraphicString
        }
        LinkPair ::= SEQUENCE {
            endpoint1 Link,
            endpoint2 Link
        }
```

10.4.3 Using OCT for Managed Object Instances

The Object Configuration Tool (OCT) is the tool within the Viewer used to configure topology nodes. When users create or modify a topology node, OCT allows them to enter the name of the managed object and configure protocol-specific information for RPC, CMIP, or SNMP.

When users elect to enter their own MOIs, OCT presents a Data Viewer type interface for browsing the MIT and choosing which MOIs to populate the topoNodeMOSet with. OCT sets the topoNodeMOSet to the cmipsnmpProxyAgent object by default.

OCT reads the GDMO attribute corresponding to the topology type being created from the TopoTypeUserDataAttrs file. The ASN.1 syntax for the attribute is retrieved by the PMI and the user is presented with the type fields of the ASN.1 construct and allowed to enter values for each type field. OCT then stores the specific user entered data in the topoNodeUserData attribute in the topology node.

In the case of an ASN.1 `SEQUENCE` construct, OCT allows the user to enter a value for each type definition in the `SEQUENCE`. In the case of an ASN.1 `SET OF` construct, OCT allows the user to enter from 0 to N entries of whatever types or constructs are in the `SET OF`.

When parsing and displaying ASN.1 types to the user, OCT displays the lowest level of expansion. For example, with the ASN.1 constructs, LinkPair defined above the display gives the user a place to enter two link names and two corresponding addresses.

When OCT displays an existing topology node, it does not look up the GDMO attribute for topoNodeUserData in the TopoTypeUserDataAttrs file. OCT displays the attribute based on the ASN.1 type or types returned by the PMI. If users change the topology type of a managed object, OCT warns them that the current topoNodeUserData is not meant for that topology type, and allows them to enter new topoNodeUserData corresponding to the new type.

10.4.4 Configuring Complex SNMP Devices in the Topology Database

Each SNMP agent the EM platform communicates with is represented by a cmipsnmpProxyAgent object in the local MIS. This managed object contains the agentId, the IP address, and the MIBs that the agent supports. When a request for a DN is generated, which translates into a managed object in the agents MIB, the request is translated to SNMP and forwarded to the agent.

Note – The IIMC provides a standard mapping of SNMP Internet MIBs to GDMO object classes.

10.4.5 Configuring Complex RPC Devices in the Topology Database

Each SNM RPC agent the EM platform communicates with is represented by an rpcAgent object in the local MIS. This managed object contains the agentId, the IP address, and the schemas that the agent supports. When a request for a DN is received that translates into a managed object in the agent’s schema, the request is translated to RPC and forwarded to the agent.

Note – This translation is similar to the IIMC translation from CMIP to SNMP. This translation is a proprietary definition to allow SunNet Manager RPC schemas to translate to the GDMO model.

When a request is started the `em_autod` passes as arguments to the `nci_request_start` all the MOIs in the topoNodeMOSet. For additional information, see the section on the NerveCenter in *Solstice Enterprise Manager Reference Manual*.

10.4.5.1 Scenario

Suppose user X wants to model, within the topology, a host and five critical applications that run on that host. User X follows this procedure.

1. **X creates a topology type `Host` and, in the same view, 5 topology-type Application objects surrounding the host.**
   X sets up the system to manage not only the host but also the five applications.

2. **If an application fails, X wants to color the applications glyph.**
   If the host fails, X wants not only the host to be colored, but all the applications. For this to occur there must be some way to define propagation relationships between topology nodes in the same view.

An additional attribute, topoNodePropagatePeers, defined for the topoNode object class, consists of a `SET OF topoNodeId`.

When the Viewer checks for propagation of a topoNodeSeverity, it also checks for the topoNodePropagatePeers attribute. If this attribute is set, the Viewer propagates the severity to the peer topology nodes.

Writing RPC Agents for EM 11

Solstice Enterprise Manager supports agents written to the Site / SunNet / Domain Manager (SNM) interfaces and libraries. SNM agents can communicate with the Enterprise Manager MIS via the EM MIS RPC interface. The EM MIS RPC interface translates from SNM ONC RPC to the Enterprise Manager PMI. The role of SNM agents and applications in the Solstice EM architecture is described in the *Solstice Enterprise Manager Reference Manual*.

In this chapter, "SNM" refers to the 2.2 or later release of SunNet Manager or releases of Solstice Site Manager or Solstice Domain Manager.

This chapter provides guidance for software developers who wish to develop SNM agents. For complete information on writing agents, see the *Site/SunNet/Domain Manager Application and Agent Development Guide*.

11.1 Manager-Agent Model

The SNM design is based on the manager/agent model in the Open Systems Interconnection (OSI) management framework. The manager is a process started by the user (for example, the EM MIS). The agent is a process that collects data from the managed resource and reports it to the manager.

The Manager/Agent Services libraries provide the management infrastructure and handle the communication services. The agent and manager need not be concerned with the underlying networking involved in their communication. The agent process need be concerned only with collecting data from the managed resource. The manager and agent processes make use of the Services through Application Programming Interfaces (APIs).

Another aspect of the manager/agent model involves the definition of management data. Open management standards (for example, OSI and the Simple Network Management Protocol (SNMP)) specify that agents abstract the properties (or attributes) of managed resources into data items (for example, “how busy a CPU is” becomes a value between 0 and 100). In SNM, the attributes for a managed resource are described in the agent *schema*. The agent is able to respond to the manager’s request, because both use the same data definitions for the managed resource.

11.2 Types of Agents

All SNM agents communicate with the manager in the manner just described. Agent types differ in the relationship with their respective managed resources.

Agents can directly or indirectly access managed resources. Most of the SNM agents provided with Solstice EM manage resources on the Sun workstations where they are installed (for example, the hostmem agent uses the same mechanism as `netstat -m` to get memory utilization data).

The second type of agent provides the ability to manage objects that are not directly accessible. Such agents are called *proxy agents*. Proxy agents run on Sun workstations, called *proxy systems,* and use protocol translation mechanisms to provide the necessary access to the managed resources. The proxy system may be the workstation on which the EM MIS is running or another workstation on the network.

The ping proxy agent provides the ability to test the reachability of Internet Protocol (IP) devices, translating manager requests into Internet Control Message Protocol (ICMP) echo requests. Similarly, the hostperf proxy agent uses the *rstat* protocol to gather host statistics.

11.3 Steps for Writing an Agent

From a high-level viewpoint, the steps involved in implementing an agent are as follows:

1. Access to the managed resource must exist (that is, code must be written to get the required management data).

Note – The prefixes `NETMGT`, `Netmgt`, and `netmgt` are reserved for network management functions.

2. Assign a name to each discrete management data item—*attribute*. For example, if the input packet count is an attribute, `ipkts` would be a good name.
3. Determine the data type for each attribute. In the example, `ipkts` is an integer.
4. Use the attribute information to form the agent schema file, which will be specific to the particular agent.

Note – The conversion of SNM schema files into GDMO files for use with the EM MIS requires that only alphanumeric characters be used for names. Therefore, you should use special characters with caution.

5. Expand the original code written for accessing the managed resource to incorporate the agent schema definitions.
6. Write the code that uses the SNM Agent Services library. This includes code for agent initialization, request handling, and error reporting.
7. Incorporate any agent-specific error messages into the agent schema file.
8. Test and integrate the completed agent code and schema file with Solstice EM.

For complete information on writing agents, see the *Site/SunNet/Domain Manager Application and Agent Development Guide*.

11.4 Solstice EM Integration

Once you are satisfied your agent is performing correctly, install it on the systems where you want it to run, then integrate it with the Solstice EM MIS database.

11.4.1 Install the Agent

For *each system* where you want your agent to run,

- Copy your agent to the directory where the other SNM agents are installed. If no SNM agents have been installed on the system, first run the SNM utility `getagents.`
- Add the following entry to `snm.conf`:

```
na.<my-agent-name>0
```

  This will set your agent security level on this host to zero (no security checking). Optionally, you can set your agent to any value between 1 and 5.

- Add an entry for your agent to `/etc/inetd.conf` to allow `inetd` to automatically start your agent when a manager sends a request to your agent. Here's a summary of the `inetd.conf`(5) file format.

```
na.<agent-name>/10 tli rpc/udp wait root  <agent program absolute pathname> <agent-name> <arguments>
```

For example, the entry for the `iproutes` agent (on a Solaris 2.x machine) is:

```
na.hostmem/10 tli rpc/udp wait root /opt/SUNWconn/snm/agents/na.iproutes na.iproutes
```

- Force `inetd` to reread its configuration file by sending it a `SIGHUP` signal:

```
host% kill -HUP  <inetd's process ID>
```

11.4.2 *Update the EM MIS Database*

Solstice EM uses GDMO descriptions, rather than the schema files used by SNM products, to represent the managed object classes available to the management database. Thus, the Solstice EM Schema compiler (`em_snm2gdmo`) is used as the first step in converting the SNM schema files into GDMO descriptions used in Solstice EM.

To translate an SNM schema file, run the Schema compiler with the file to be translated as an argument (<*filename*>).

```
hostname% em_snm2gdmo < <filename>
```

The output of the translation is a GDMO document, with a name of the form <*filename*>`.gdmo`, and an ASN.1 document, with a name of the form <*filename*>`.asn1`. You must then pass the GDMO file through the GDMO compiler.

For more information on the GDMO and schema-to-GDMO compilers, refer to Chapter 2, “Utilities,” in the *Solstice Enterprise Manager Reference Manual*.

Terminology References A

The ISO specifications provide precise technical definitions for a number of terms used through out this document. This Appendix provides a road map into the ISO specifications for terminology definitions.

Table A-1 ISO Specifications for Terminology Definitions

Term	**Defined In**
action	X.720 ISO / IEC 10165-1
agent	X.701 ISO / IEC 10040
agent role	X.701 ISO / IEC 10040
Application Entity Title (AE-Title)	ISO 7498-3
Application Program Title (AP-Ttile)	ISO 7498-3
attribute	X.710 ISO / IEC 9595
attribute identifier	X.720 ISO 10165-1
attribute type	X.720 ISO 10165-1
attribute value assertion	X.720 ISO 10165-1
constructed encoding	X.209 ISO 8825
containment	X.720 ISO 10165-1
destination (see distination Attribute syntax))	X.734 ISO 10164-5
discriminator	X.734 ISO 10164-5

Table A-1 ISO Specifications for Terminology Definitions

Term	**Defined In**
distinguished name	X.720 ISO 10165-1
end-of-contents octets	X.209 ISO 8825
event forwarding discriminator (see eventForwardingDiscriminator Managed Object Class syntax)	X.734 ISO 10164-5
event report management function	X.734 ISO 10164-5
filter (see CMISfilter syntax)	X.711 ISO / IEC 9596-1
functional unit	X.710 ISO / IEC 9595
identifier octets	X.209 ISO 8825
length octets	X.209 ISO 8825
local distinguished name (see ObjectInstance syntax)	X.711 ISO / IEC 9596-1
managed object	ISO 7498-4
managed object class	X.701 ISO / IEC 10040
management information	X.701 ISO / IEC 10040
managed information base	ISO 7498-4
manager	X.701 ISO / IEC 10040
manager role	X.701 ISO / IEC 10040
naming binding	X.720 ISO 10165-1
naming tree	X.720 ISO 10165-1
notification	X.701 ISO / IEC 10040
notification type	X.701 ISO / IEC 10040
open system	X.200 ISO 7498
primitive encoding	X.209 ISO 8825
relative distinguished name	X.720 ISO 10165-1
systems management	X.200 ISO 7498
top managed object class	X.721 ISO / IEC 10165-2

Access to Data in the MIS B

B.1 Overview

The central function of the Solstice EM MIS is to make data regarding managed objects available to its client services. The term *managed objects* includes both the physical resources connected to the network, such as hosts, bridges, and routers—as well as conceptual objects, such as lines, circuits, and queues. It also includes any objects an application may choose to create, such as requests, views, counters, lists, or collections.

All managed objects are accessed through the MIT. There is one global tree, and that tree provides a single naming scheme for all data. The MIT is constructed according to rules provided by OMNIPoint. The tree has a single root (called *root*). The shape of the tree is arbitrary, and may vary considerably from one Solstice EM MIS to another. Generally speaking, the tree's structure represents containment relationships. That is, below an object, one finds those objects that in some sense are related to it.

Data from managed resources (including MIB data) are also a part of the MIT. The Solstice EM MIS makes this data accessible both through the naming conventions that come from the managed resources it describes, and through a resource-independent naming convention, in which identifiers are specified using Fully Distinguished Names (FDNs). It is this mechanism that enables the Solstice EM MIS to achieve transparency of location.

B.2 Object Orientation

Object orientation is central to the Solstice EM product, which takes advantage of object-oriented design in two complementary ways:

- It describes managed objects in terms of OMNIPoint and ISO terminology;
- It uses C++ objects for internal storage and manipulation of network data.

The Solstice EM MIS uses one or more C++ classes to correspond directly to the classes used in the formal descriptions. Within this document the following terminology is used:

- *class* refers to a C++ class as described in *The Annotated C++ Reference Manual*[1].
- *instance* refers to the memory which is allocated for the instantiation of a C++ class according to its definition. A variable name is usually associated with a particular instance of a class.
- *managed resource* refers to an actual physical device or entity that exists in a network or system. This is consistent with the OSI / NM Forum definition of this term.
- *managed object* (MO) refers to a set of services and attributes that describes a type of managed resource. Again, this is consistent with the OSI / NM Forum definition for this term.

1. *Annotated C++ Reference Manual*, Margaret Ellis and Bjarne Stroustrop. Addison-Wesley Publishing Co., Reading, MA. Copyright 1990 by AT&T Bell Telephone Laboratories.

- *managed object class* (MOC). This term refers to the internal representation of a managed object (as defined by OSI/NM Forum). A MOC is typically the GDMO description, but might also be an SNMP MIB description. Put another way, the MOC represents the attributes and behaviors for particular types of manageable objects. The MOC defines the type of data stored, and the behaviors that can be taken, but does not represent actual data for any managed object. The MOC is the internal representation used by the Solstice EM MIS.
- *managed object instance* (MOI) relates to a managed object class in the same way as an instance relates to a class. The MOC determines the type of attributes and behaviors available to operate on an object of this type; an MOI refers to actual data that represents an object which is being managed by the MIS. An MOI is also an internal representation used by the Solstice EM MIS.

B.3 Representation of Objects in the MIT

Every object known to the MIS is represented in the Management Information Tree (MIT), including all types of objects. Such objects:

- Represent physical devices, such as hosts, bridges, and routers
- Represent parts of a device, such as ports and network access cards
- Are primarily conceptual, such as lines and functional groups
- Are primarily the creation of software, such as queues
- Are entirely the creation of client services, such as counters, tables, and lists

Each object in the MIS has either an entry—consisting of a Fully Distinguished Name—in the Management Information Tree or, if it is outside of the local MIS, an entry in the FDN Table, which contains a pointer to where the object resides. The FDN Table maps a remote object’s Distinguished Name to its Presentation Address. When an application requests information of an object, if the object is in the FDN Table, the MIS forwards the application’s request to agent or manager at the address listed in the table.

Storage for local objects is shown in Figure B-1.

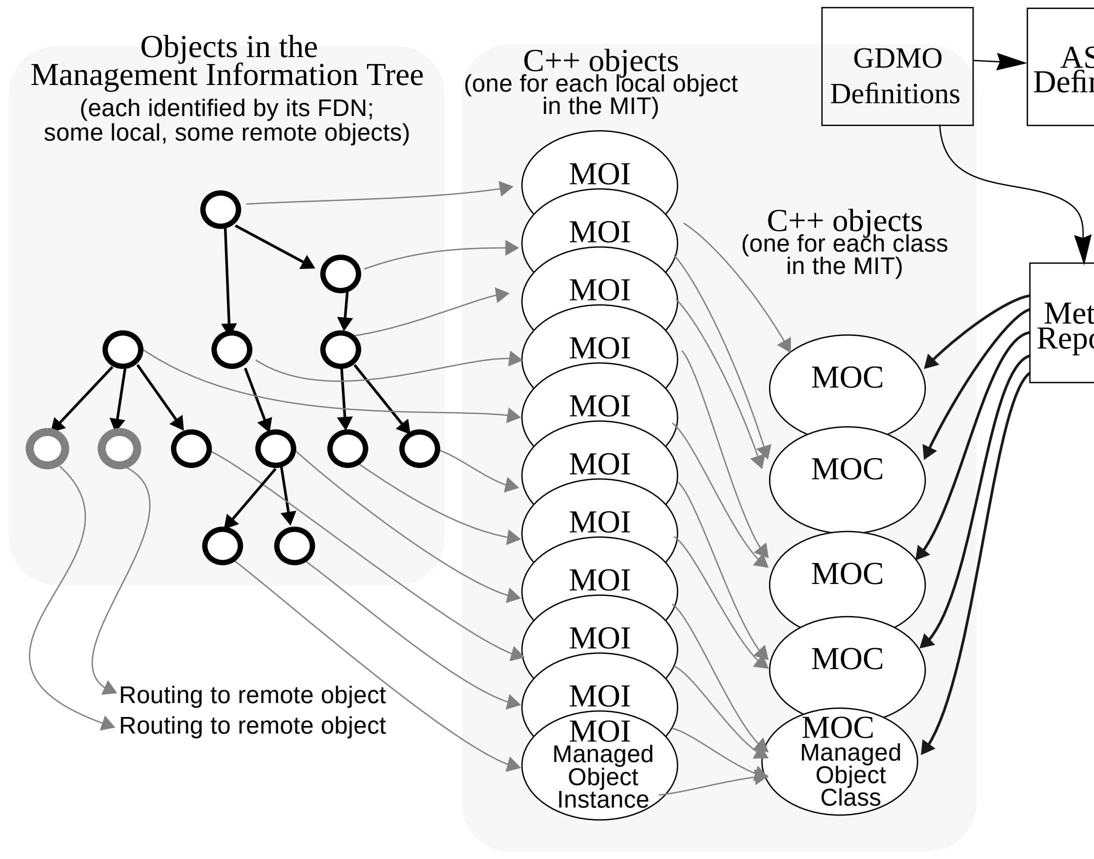


Figure B-1 Objects in the Management Information Tree and C++ Objects

For each local object, the MIS contains a corresponding C++ object. Such C++ objects are either a Managed Object Instance (MOI), or a class derived from a MOI.

Each MOI includes a reference to its class. Each class description is an MOC: that is, an instance of the C++ class called MOC (Managed Object Class).

Each MOC instance describes the characteristic of a class of managed objects, and is constructed by referring to an entry in the MetaData Repository (which contains the GDMO form of its description).

Descriptions of managed resources in the MetaData Repository, and also MOC and MOI definitions, can refer to ASN.1 definitions, so all of them make reference to the MIS's set of ASN.1 definition files.

B.4 Identifying an Object in the MIT

Each managed object is uniquely identified by its fully distinguished name, or FDN, which is the path through the MIT required to reach the object. Paths are written in a format similar to paths in the UNIX file system. That is, a path beginning with `/` starts from the root of the tree. Each successive level of the tree starts with another `/`.

Each segment of the path identifies both the syntactic category and the individual name of an object in that category. In this regard, paths through the MIT are more complex than the names of UNIX files. The following is an example of an FDN:

```
/systemId=name:"myhost"/systemId=NAME:"EM"
```

B.5 Object Behavior Framework

Much of the MIS's power comes from the ability not just to describe objects but to specify the actions they can take—their behaviors. When an object's behavior is specified, it becomes possible to elicit the behavior. The MIS allows you to specify behaviors of objects through the ACTION clause of a GDMO description and through the behaviors already defined in the MIS.

B.6 ASN.1 Definitions

Both in its CMIS-like messages and in data storage for managed objects, the Solstice EM MIS makes extensive use of an ISO standard description of the way data is represented. The description is called ASN.1 (Abstract Syntax Notation One). A message that uses ASN.1 must include not only the encoded data, but also information about the type of the encoded information. A process that represents a value by an ASN.1 encoding, or that extracts a value from an ASN.1 encoding, must be able to access the definition of the particular encoding employed. The GDMO descriptions of new objects also make reference to ASN.1 descriptions of their data.

To provide effective access to the needed ASN.1 definitions, the Solstice EM MIS keeps a set of files containing type definitions and encoding rules. When the definitions of new objects make reference to new ASN.1 types, files containing the new ASN.1 definitions must be included in the MIS's `/var/opt/SUNWconn/em/data/ASN.1` directory. At startup and periodically, the Solstice EM MIS loads these ASN.1 type files into the MDR. Refer to the "Utilities" chapter in the *Solstice Enterprise Manager Reference Manual* for detailed information about ASN.1 documents and how they are compiled.

B.7 Transparent Access to the Distributed Storage of Managed Objects

A managed resource can reside anywhere in the network. The MIS's MIT serves as a map of all the information in the network. It contains an entry for every managed object known to the Solstice EM MIS. For each managed object, the MIT contains data about the route by which the MIS can get the managed information, including objects whose information is stored in the Solstice EM MIS itself, referred to as *local* objects.

To access objects whose information is stored outside the local MIT, the MIS uses the FDN Table. The FDN Table contains the Presentation Addresses of objects not in the local MIT. Whenever the local MIS receives a request for the value of attributes for a remote object, a request must be sent to the remote entity (agent or MIS) to fetch the current attribute value.

Access to a non-local object might require a protocol translation. Where a managed object is reported by an agent that is limited to SNMP, the MIT manager software creates a *proxy* agent. Although the proxy accepts CMIP-based requests and returns CMIP-based responses, it uses SNMP to communicate with the remote agent. Because CMIP provides facilities that are not directly supported in SNMP, the proxy agent is considerably more than a translator. For example, to support CMIP scoping, the proxy may have to generate multiple requests and coordinate multiple responses. To handle some CMIP notifications, the proxy may have to poll the SNMP agent.

An Solstice EM application can refer to any object that has an entry in the MIT. The application does not need to know how the MIS communicates with the managed object. In particular, it addresses remote objects or proxies in the same way as other managed objects.

Figure B-2 illustrates the mechanism by which Solstice EM applications can communicate with remote managed objects.

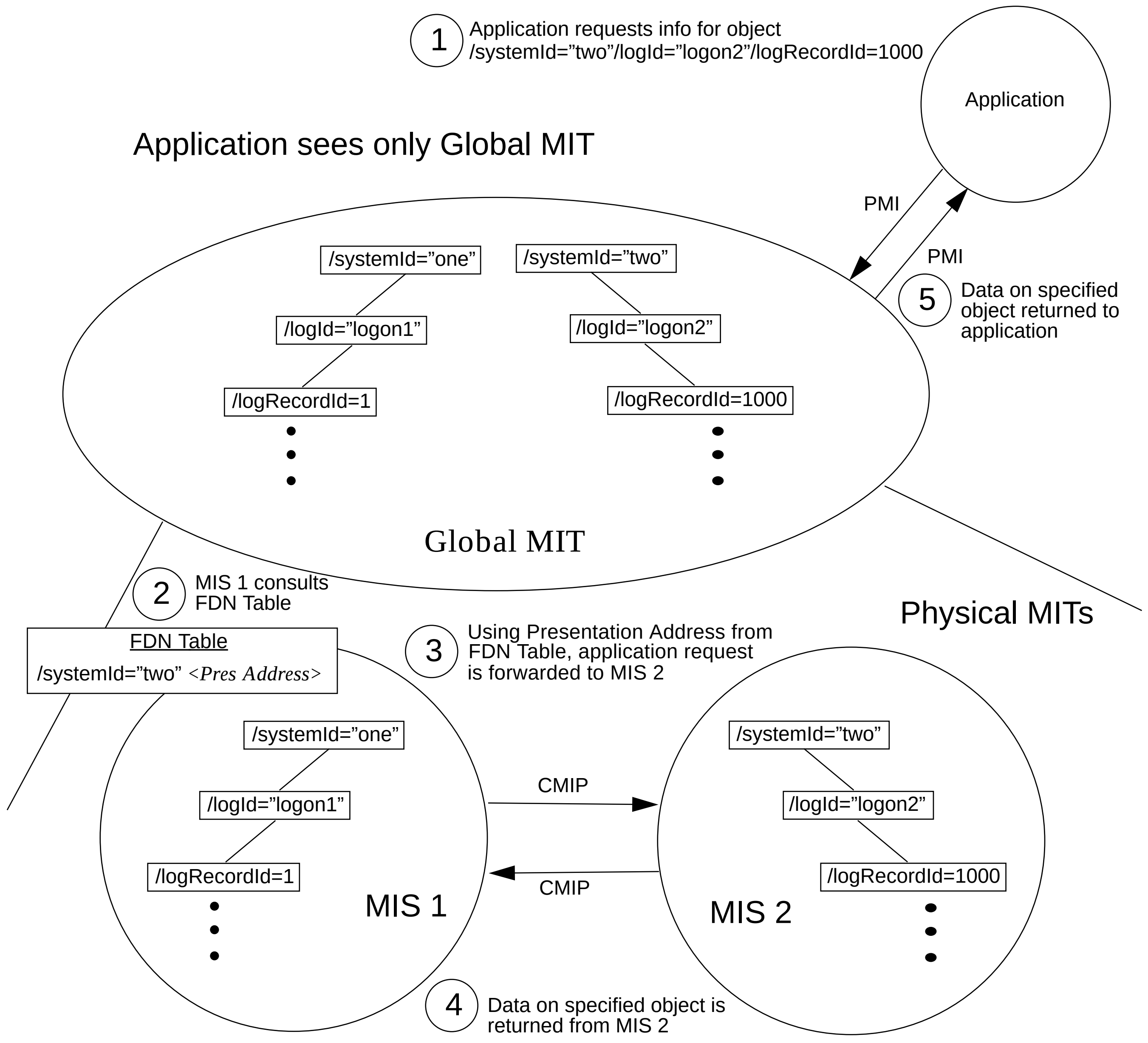

1
Application requests info for object /systemId="two"/logId="logon2"/logRecordId=1000
Application
Application sees only Global MIT
PMI
PMI
5
Data on specified object returned to application
/systemId="one"
/logId="logon1"
/logRecordId=1
/systemId="two"
/logId="logon2"
/logRecordId=1000
Global MIT
2
MIS 1 consults FDN Table
FDN Table
/systemId="two" <Pres Address>
3
Using Presentation Address from FDN Table, application request is forwarded to MIS 2
Physical MITs
/systemId="one"
/logId="logon1"
/logRecordId=1
MIS 1
CMIP
CMIP
/systemId="two"
/logId="logon2"
/logRecordId=1000
MIS 2
4
Data on specified object is returned from MIS 2

Figure B-2 Access to Remote Managed Objects

Note the sequence illustrated in Figure B-2:

1. An application, using the PMI, makes a request for information about the object with the FDN of `/systemId="two"/logId="logon2"/logRecordId=1000`.
2. As with each incoming request, MIS 1 consults its FDN Table, where it finds an entry for `/systemId="two"`.
3. MIS 1 forwards the request to MIS 2 over a CMIP connection, using the Presentation Address found in the FDN Table entry.
4. MIS2 returns Data for specified object to MIS 1, over a CMIP connection.
5. MIS 1 returns data for specified object to application, by the PMI.

em_debug Analysis C

C.1 EM_DEBUG Analysis of Trace on Object Access Module Object

The following command was used to turn on the em_debug feature:

```
em_debug “on oam*”
```

PMI client program /opt/SUNWconn/em/src/pmi1_1/boot.cc was used to initiate the trace by executing the following command:

```
./boot/systemID=<hostname>/logid=\”AlarmLog”
administrativeState
```

C.1.1 M-CREATE Request

The PMI call platform.connect() results in a creation of emApplicationInstanceobject. This happens when the pmi client issues a M-CREATE request to the MIS. The RDN of this object is

subsystemId=”EM-MIS”/emApplicationID=5.

This call also results in a M-SET request issued by the PMI client to set the attribute administrativeState to unlocked for this emApplicationInstance.

These two events result in MIS sending 2 M-EVENT-REPORT requests to notify the PMI client of an objectCreation [emApplicationInstance object] event and an AVC [administrative attribute of this emApplicationInstance changed to unlocked] event.

```
#
# PLAT_CONNECT
#
# oammsg_debug:
*************************************************************
oammsg_debug:   received by OAM: create request
oammsg_debug:   message type = create request
```

M-Create Indication is received by Object Access Manager. The format for an M-Create Indication from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

is as follows:

```
CreateArgument ::= SEQUENCE {
        managedObjectClass      ObjectClass,
        CHOICE  {
                managedObjectInstance   ObjectInstance,
                superiorObjectInstance  [8] ObjectInstance
        } OPTIONAL,
        accessControl           [5] AccessControl OPTIONAL,
        referenceObjectInstance [6] ObjectInstance OPTIONAL,
        attributeList           [7] IMPLICIT SET OF Attribute
OPTIONAL
}
```

Thus the M-Create Indication (CreateArgument) will contain the following (in strict order):

- managedObjectClass
- optionally managedObjectInstance or superiorObjectInstance
- optionally accessControl
- optionally referenceObjectInstance

- optionally attributeList

```
oammsg_debug:   id = 7
oammsg_debug:   source =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   dest =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   mode = CONFIRMED
```

The mode is CONFIRMED. i.e. M-Create is a confirmed operation. The sender of the M-Create expects a response.

```
oammsg_debug:   app_context = Du: no data unit allocated
oammsg_debug:   oc =
oammsg_debug:    Tag  Length                  Value
oammsg_debug:    C0   c             1.3.6.1.4.1.42.2.2.2.1.3.2
```

The above, `oc`, is the managedObjectClass of the target of this operation. That is, the Object Class of the new object that is to be created. From earlier we see that: managedObjectClass is of type ObjectClass. And from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectClass is defined as follows:

```
ObjectClass ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Enterprise Manager always uses the global Form CHOICE as this is guaranteed to be unique. We can see that globalForm is of type OBJECT IDENTIFIER. Em_debug always prints out the encoded value of CMIP Protocol Data Units. The general form for an encoded value is:

Tag Length Value

where Tag indicates the ASN.1 type that exists in the Value field. In the above definition of ObjectClass we have [0] which indicates that the Tag for globalForm is a Context Specific 0 [C0]. This tag replaces the Tag that would be used, OID, is the [0] IMPLICIT was not in this definition. So the managedObjectClass has the following encoded form:

Tag	Length	Value
C0	Length of Value	<Object Identifier>

In this specific case we see:

```
oammsg_debug:   oc =
oammsg_debug:    Tag     Length                    Value
oammsg_debug:   C0      c             1.3.6.1.4.1.42.2.2.2.1.3.2
```

Thus, the OBJECT IDENTIFIER of the managedObjectClass is

`1.3.6.1.4.1.42.2.2.2.1.3.2`

The easy way to find out what this means is to use the Object Editor to issue an action against the Meta Data Repository.

1. **Select the /systemId=name:"hostname"/metaName="MDR" object**

2. **Select Object->Action->getOidName**

3. **When the Action dialog box appears enter the OID in curly braces as follows:**

   ```
   { 1 3 6 1 4 1 42 2 2 2 1 3 2 }
   ```

4. **Click OK**

5. **The Output Window will then be displayed. For this example it will be:**

```
{
    oid “EM GMI”:emApplicationInstance,
    name “““EM GMI””:emApplicationInstance”
    }
```

Thus, we can see that this M-Create is for a new object with Object Class emApplicationInstance.

```
oammsg_debug:   oi = NULL
oammsg_debug:   superior_oi =
oammsg_debug:    Tag    Length          Value
oammsg_debug:    C4     13
oammsg_debug:    Tag    Length          Value
oammsg_debug:    SET    11
oammsg_debug:    Tag    Length          Value
oammsg_debug:    SEQ    f
oammsg_debug:    Tag    Length          Value
oammsg_debug:    OID    5 2.9.3.5.7.11
oammsg_debug:    Tag    Length          Value
oammsg_debug:    GRPH   6               “EM-MIS”
```

We can see from the earlier definition of CreateArgument that the ObjectInstance (the name of the object that we are creating) can be specified in one of two ways:

```
CHOICE  {
                managedObjectInstance    ObjectInstance,
                superiorObjectInstance  [8] ObjectInstance
        } OPTIONAL,
```

That is, we can specify either the name of the object to be created (managedObjectInstance) or the name of the object that contains the object to be created (superiorObjectInstance).

em_debug calls managed ObjectInstanceoi, and calls superiorObjectInstance superior_oi. So in this case the superiorObjectInstance is being used. To decode the message further, we start with the definition of superiorObjectInstance:

```
superiorObjectInstance  [8] ObjectInstance
```

from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectInstance is defined as follows:

```
ObjectInstance ::= CHOICE {
        distinguishedName       [2] IMPLICIT DistinguishedName,
        nonSpecificForm         [3] IMPLICIT OCTET STRING,
        localDistinguishedName  [4] IMPLICIT RDNSequence
}
```

Note the context specific tag of each member of the choice. The first tag in the em_debug output is:

```
oammsg_debug:   superior_oi =
oammsg_debug:    Tag  Length        Value
oammsg_debug:    C4   13
```

Thus what follows in localDistinguishedName form. Again, localDistinguishedName is of ASN.1 Type RDNSequence.

From the InformationFramework

`[/opt/SUNWconn/em/etc/asn1/infofw.asn1]`

RDNSequence is defined as follows:RDNSequence ::= SEQUENCE OF RelativeDistinguishedName and that:

RelativeDistinguishedName ::= SET OF AttributeValueAssertion and that:

```
AttributeValueAssertion ::=  SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
}
```

Putting this all together we get:

ObjectInstance is the

```
        localDistinguishedName  [4] IMPLICIT
                                    SEQUENCE OF
                                    SET OF
                                    SEQUENCE
{
                                attributeId    OBJECT IDENTIFIER,
                                    attributeValue  ANY DEFINED
BY attributeId
                                      }
```

Note – When a type defined in a CHOICE e.g. RDNSequence has a context specific Tag e.g. [4], this tag replaces the TAG of the type that follows (like the OBJECT IDENTIFIER example earlier). So in this case the TAG C4 replaces SEQUENCE OF in the encoded value of localDistinguishedName.

Thus, superiorObjectInstance will have the following form for its value:

```
Tag             Length                                Value
C4              A
Tag             Length                                Value
SET             B
Tag             Length                                Value
SEQUENCE        C
Tag             Length                                Value
OID             D
Tag             Length                                Value
X               E
```

The final Tag is X because it depends on the value of the previous OID. Compare this with the em_debug output:

```
oammsg_debug:   superior_oi =
oammsg_debug:    Tag       Length                          Value
oammsg_debug:    C4        13
oammsg_debug:    Tag       Length                          Value
oammsg_debug:    SET       11
oammsg_debug:    Tag       Length                          Value
oammsg_debug:    SEQ       f
oammsg_debug:    Tag       Length                          Value
oammsg_debug:    OID       5                              2.9.3.5.7.11
oammsg_debug:    Tag       Length                          Value
oammsg_debug:    GRPH      6                               "EM-MIS"
```

We note that the final part:

```
oammsg_debug: Tag       Length               Value
oammsg_debug: OID       5                    2.9.3.5.7.11
oammsg_debug  Tag       Length               Value
oammsg_debug: GRPH      6                    "EM-MIS"
```

is actually

```
AttributeValueAssertion ::=  SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
```

So we need to first look up the OID. Using the Object Editor we find that 2.9.3.5.7.11 is subsystemId. Now we can lookup the syntax of subsystemId. This will help us resolve the ANY DEFINED BY attributeId. Using the Object Editor issue an action against the MDR as before. This time the action will be getAttribute. The parameter is as before:

{ 2 9 3 5 7 11 }

The output window will display the syntax:

```
{
    "Rec. X.723 | ISO/IEC 10165-5":subsystemId,
    {
        {
            document "Rec. X.723 | ISO/IEC 10165-5",
            label "subsystemId"
        }
    },
    defined : {
        module "",
        name "GraphicString"
    }
}
```

We see that subsystemId has a syntax or ASN.1 type of GraphicString. Thus we expect the second part of the sequence to be:

Tag	Length	Value
Graphic	String	X

Now compare this with the real encoded value:

```
oammsg_debug:   Tag    Length    Value
oammsg_debug:   GRPH   6         “EM-MIS”
```

Thus the superiorObjectInstance is:

`subsystemId=”EM-MIS”`

Because we are using localDistinguishedName form, this name is relative to

`/systemId=name:”hostname”`

and we are specifying that the Object to be created will be contained by:

`/systemId=name:”hostname”/subsystemId=”EM-MIS”`

This is where the object instances that represent EM application are always found.

```
oammsg_debug:   access = NULL
oammsg_debug:   reference_oi = NULL
```

We saw earlier in the definition of a CreateArgument that:

```
        accessControl           [5] AccessControl OPTIONAL,
        referenceObjectInstance [6] ObjectInstance OPTIONAL,
```

That is, both accessControl, and referenceObjectInstance are optional. em_debug represents the absence of a parameter by NULL.

The final field in the M-Create Indication is the attributes that are to be set in the new object instance and the initial values of these attributes. This is the attributeList parameter:

```
       attributeList          [7] IMPLICIT SET OF Attribute OPTIONAL
```

From ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1])`

Attribute is as follows:

```
Attribute ::=  SEQUENCE {
        attributeId     AttributeId,
        attributeValue  ANY DEFINED BY  attributeId
}
```

Thus, attributeList is defined as follows:

```
attributeList           [7] IMPLICIT SET OF
                               SEQUENCE {
                               attributeId     AttributeId,
                               attributeValue  ANY DEFINED BY
attributeId
                               }
```

And AttributeId is defined as:

```
AttributeId ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Remembering that [7] replaces SET OF (as in the earlier example) we have the following encoding for attributeList:

```
Tag                    Length                                Value
C7                     A
Tag                    Length                                Value
SEQUENCE               B
Tag                    Length                                Value
C0 or C1               C
Tag                    Length                                Value
?                      D
Tag                    Length                                Value
SEQUENCE               E
Tag                    Length                                Value
C0 or C1               F
Tag                    Length                                Value
?                      G
```

and so on. One of the Tags is ? because it depends on the value of AttributeId.

Note – The AttributeId can be either: globalForm tagged as C0 : type is OBJECT IDENTIFIER OR localForm tagged as C1 : type is INTEGER.

Now, we can decode the attributeList as follows using the above and the Object Editor:

```
oammsg_debug:   attr_list =
oammsg_debug:    Tag    Length        Value
oammsg_debug:    C7     7b
```

We have attributeList:

```
oammsg_debug: Tag     Length                   Value
oammsg_debug: SEQ     11
oammsg_debug: Tag     Length                   Value
oammsg_debug: C       c               1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag     Length                   Value
oammsg_debug:         1 =                      5U
```

First attribute (SEQuence of AttributeId, AttributeValue). The AttributeId is in globalform (Tag C0). As this is an OID, it decodes to:

emApplicationID

Next, we must find the syntax of emApplicationID. The getAttribute action yields:

```
{
    “EM GMI”:emApplicationID,
    {
        {
            document “EM GMI”,
            label “emApplicationID”
        }
    },
    defined : {
        module “EM-GMI-ASN1”,
        name “EMApplicationID”
    }
}
```

The syntax is EMApplicationID defined in module EM-GMI-ASN1. We can use the Object Editor MDR Action getAsn1Module with parameter EM-GMI-ASN1 to recover all the ASN.1 types in the module. Looking down we see that:

```
 {
           name “EMApplicationID”,
           type integer : {
           }
```

That is, EMApplicationID ::= INTEGER Thus, the next TAG should be INT. From em_debug we see:

```
oammsg_debug:                        Tag         Length       Value
oammsg_debug:                        INT         1    =       5U
```

Thus, the value is 5 (INTEGER).

And, finally, this attribute and value are:

emApplicationID=5

```
oammsg_debug Tag      Length        Value
oammsg_debug:SE       1a
oammsg_debug:Tag      Length        Value
oammsg_debug:C0       c             1.3.6.1.4.1.42.2.2.2.1.7.2
oammsg_debug Tag      Length        Value
oammsg_debug:GRPH     a             “Test_album”
```

emApplicationType=”Test_album”

```
oammsg_debug: Tag      Length       Value
oammsg_debug: SEQ      2c
oammsg_debug: Tag      Length       Value
oammsg_debug: C0       c            1.3.6.1.4.1.42.2.2.2.1.7.6
oammsg_debug: Tag      Length       Value
oammsg_debug: SEQ      1c
oammsg_debug: Tag      Length       Value
oammsg_debug: SET      1a
oammsg_debug: Tag      Length       Value
oammsg_debug: SEQ      18
oammsg_debug: Tag      Length       Value
oammsg_debug: OID      c            1.3.6.1.4.1.42.2.2.2.1.7.7
oammsg_debug: Tag      Length       Value
oammsg_debug: GRPH     8            “sudhakar”
```

This one is more complex. Let us examine it step by step.

AttributeId = emUserID

AttributeValue has the syntax of emUserID. We can resolve this to typeDistinguishedName which the EM-GMI-ASN1 module imports from the InformationFramework:

DistinguishedName ::= RDNSequence

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET OF AttributeValueAssertion

AttributeValueAssertion ::=

```
SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
}
```

Thus, DistinguishedName has the following syntax:

```
SEQUENCE OF SET OF SEQUENCE {
                             attributeId     OBJECT IDENTIFIER,
                       attributeValue  ANY DEFINED BY attributeId
                             }
```

Thus, DistinguishedName will have the following form for its encoded value:

```
Tag                 Length                          Value
SEQ                 A
Tag                 Length                        Value
SET                 B
Tag                 Length                        Value
SEQUENCE            C
Tag                 Length                        Value
OID                 D
Tag                 Length                        Value
X                   E
```

The final Tag is X because its value depends on the value of the previous OID. Thus, we have:

```
emUserID=  { { { emUserLogin, "sudhakar" } }}
```

or, if you prefer:

```
emUserID= /emUserLogin="sudhakar"
```

```
oammsg_debug: Tag    Length         Value
oammsg_debug: SEQ    10
oammsg_debug  Tag    Length         Value
oammsg_debug: C0     c              1.3.6.1.4.1.42.2.2.2.1.7.11
oammsg_debug: Tag    Length         Value
oammsg_debug: SET    0
```

```
emSpecialEvents = {}
```

(the empty set)

```
oammsg_debug: Tag       Length         Value
oammsg_debug: SEQ       a
oammsg_debug: Tag       Length         Value
oammsg_debug: C0        5              0x59 03 02 07 1f
oammsg_debug: Tag       Length         Value
oammsg_debug: ENUM      1     =        0
```

Here the OID has not been decoded for us. Thus, we need to do so manually:

0x59 03 02 07 1f = 2.9 3 2 7 31 = 2.9.3.2.7.31 = administrativeState

The syntax for administrativeState from dmi.asn1 is:

`AdministrativeState ::= ENUMERATED`

```
{
    locked              (0),
    unlocked            (1),
    shuttingDown        (2)
}
```

Thus, we have:

administrativeState=locked

```
oammsg_debug:   ************************************************
```

We have now completed the M-Create Request.

```
oam_debug:      resolve_attr: attr 1.3.6.1.4.1.42.2.2.2.1.7.1
resolved to supplied

oam_debug:      resolve_attr: attr 1.3.6.1.4.1.42.2.2.2.1.7.2
resolved to supplied

oam_debug:      resolve_attr: attr 1.3.6.1.4.1.42.2.2.2.1.7.6
resolved to supplied

oam_debug:      resolve_attr: attr 1.3.6.1.4.1.42.2.2.2.1.7.11
resolved to supplied

oam_debug:      resolve_attr: attr 2.9.3.2.7.31 resolved to
supplied

oam_debug:      resolve_attr: attr 2.9.3.2.7.35 resolved to
magical

oam_debug:      resolve_attr: attr 2.9.3.2.7.50 resolved to
magical

oam_debug:      resolve_attr: attr 2.9.3.2.7.56 resolved to
default

oam_debug:      resolve_attr: attr 2.9.3.2.7.63 resolved to
magical

oam_debug:      resolve_attr: attr 2.9.3.2.7.65 resolved to
magical

oam_debug:      resolve_attr: attr 2.9.3.2.7.66 resolved to
magical
```

The MIS now resolves those attributes in the M-Create Request and creates a new emApplicationInstance object.

oam_info: DiscriminatorAttrSecty:write val idx=1DiscriminatorAttrSecty:write val idx=2DiscriminatorAttrSecty:write val idx=0DiscriminatorAttrSecty:read val idx=2

```
oamnot_debug:   M-Event-Report Message =
oamnot_debug:   message type = event_report request
oamnot_debug:   id = 14915424
oamnot_debug:   source =
oamnot_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oamnot_debug:   dest =
oamnot_debug:     aclass = DEF, atag = 1
  aval = Du: no data unit allocated
oamnot_debug:   remote =
oamnot_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
```

C.1.1.1 M-Event Report

The MIS now generated an M-Event-Report.

```
oamnot_debug:   mode = UNCONFIRMED
oamnot_debug:   app_context = Du: no data unit allocated
```

The mode is unconfirmed. The MIS does not expect a response. The syntax of an Event Report is (from ITU X.711):

```
EventReportArgument ::= SEQUENCE {
managedObjectClass      ObjectClass,
        managedObjectInstance   ObjectInstance,
      eventTime           [5] IMPLICIT GeneralizedTime OPTIONAL,
        eventType               EventTypeId,
      eventInfo             [8] ANY DEFINED BY eventType OPTIONAL
}
```

```
oamnot_debug:   oc =
oamnot_debug:    Tag  LengthValue
oamnot_debug:    C0     c 1.3.6.1.4.1.42.2.2.2.1.3.2
```

The above, oc, is the managedObjectClass of the source of this operation (M-Event-Report). That is, the Object Class of the new object that was just created. From earlier we see that: managedObjectClass is of type ObjectClass. And from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectClass is defined as follows:

```
ObjectClass ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Thus, from the em_debug trace we see that once again we use the globalFormCHOICE and that managedObjectClass = emApplicationInstance.

```
camnot_debug: oi =
camnot_debug: Tag       Length                Value
camnot_debug: C2        3d
camnot_debug: Tag       Length                Value
camnot_debug: SET       13
camnot_debug: Tag       Length                Value
camnot_debug: SEQ       11
camnot_debug: Tag       Length                Value
camnot_debug: OID       5                     2.9.3.2.7.4
camnot_debug: Tag       Length                Value
camnot_debug: GRPH      8                     “dynamics”
camnot_debug: Tag       Length                Value
camnot_debug: SET       11
camnot_debug: Tag       Length                Value
camnot_debug: SEQ       f
camnot_debug: Tag       Length                Value
camnot_debug: OID       52.9.3.5.7.11
camnot_debug: Tag       Length                Value
camnot_debug: GRPH      6                     “EM-MIS”
camnot_debug: Tag       Length                Value
camnot_debug: SET       13
camnot_debug: Tag       Length                Value
camnot_debug: SEQ       ll
camnot_debug: Tag       Length                Value
camnot_debug: OID      C              1.3.6.1.4.1.42.2.2.2.1.7.1
camnot_debug: Tag       Length                Value
camnot_debug: INT       1  =                   5U
```

We can see from the earlier definition of EventReportArgument that the managedObjectInstance is of type ObjectInstance.

From ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectInstance is defined as follows:

```
ObjectInstance ::= CHOICE {
        distinguishedName       [2] IMPLICIT DistinguishedName,
        nonSpecificForm         [3] IMPLICIT OCTET STRING,
        localDistinguishedName  [4] IMPLICIT RDNSequence
}
```

Note the context specific tag of each member of the choice. Looking at the em_debug output we can see the first tag is:

```
oammsg_debug:    Tag         Length             Value
oammsg_debug:    C2          13
```

Thus, what follows in DistinguishedName form. From the InformationFramework

`[/opt/SUNWconn/em/etc/asn1/infofw.asn1]`

is defined as follows: DistinguishedName ::= RDNSequence

RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

RelativeDistinguishedName ::= SET OF AttributeValueAssertion

```
AttributeValueAssertion ::=  SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
}
```

Thus, DistinguishedName has the following syntax:

```
SEQUENCE OF SET OF SEQUENCE {
                             attributeId     OBJECT IDENTIFIER,
                       attributeValue  ANY DEFINED BY attributeId
                             }
```

Thus, DistinguishedName will have the following form for its encoded value:

```
Tag          Length                    Value
SEQ          A
Tag          Length                    Value
SET          B
Tag          Length                    Value
SEQ          C
Tag          Length                    Value
OID          D
Tag          Length                    Value
X            E
```

The final Tag is X because it depends on the value of the previous OID. Now, compare this with the em_debug output:

AttributeId : emApplicationID

```
camnot_debug: oi =
camnot_debug: Tag       Length        Value
camnot_deubg: C2        3d
DistinguishedNameform
camnot_debug: Tag       Length        Value
camnot_debug: SET       13
camnot_deubg: Tag       Length        Value
camnot_debug: SET       13
camnot_debug: Tag       Length        Value
camnot_debug: SEQ       11
camnot_debug: Tag       Length        Value
camnot_debug: OID       5             2.9.3.2.7.4
camnot_debug: Tag       Length        Value
camnot_debug: GRPH      8             ”dynamics”
AttributeId    :  systemId
AttributeValue:  ”dynamics” (uses the name CHOICE)
camnot_debug: Tag       Length         Value
camnot_debug: SET       11
camnot_debug: Tag       Length         Value
camnot_debug: SEQ       f
camnot_debug: Tag       Length         Value
camnot_debug: OID       5               2.9.3.5.7.11
camnot_debug: Tag       Length          Value
camnot_debug: GRAPH     6               ”EM-MIS”
AttributeId    :  subsystemId
AttributeValue:  ”EM-MIS”
camnot_debug: Tag       Length          Value
camnot_debug: SET       13
camnot_debug: TAG       Length          Value
camnot_debug: SEQ       11
camnot_debug: Tag       Length          Value
camnot_debug: OID      c              1.3.6.1.4.1.42.2.2.2.1.7.1
camnot_debug: Tag       Length          Value
camnot_debug: INT       1  =             5U
AttributeId    :  emApplicationID
AttributeValue:  5
```

Thus, the managedObjectInstance is:

`/systemId=name:”dynamics”/subsystemId=”EM-MIS”/emApplicationID=5`

```
oamnot_debug:   event_type =
oamnot_debug:    Tag      Length           Value
oamnot_debug:    C6       5                 0x59 03 02 0a 06
```

From earlier we have:

eventType EventTypeId,

and from X.711 EventTypeId has the following syntax:

```
EventTypeId ::= CHOICE {
        globalForm      [6] IMPLICIT OBJECT IDENTIFIER,
        localForm       [7] IMPLICIT INTEGER
}
```

As the first Tag from the em_debug trace is C6, we are using globalForm, and the value of type OBJECT IDENTIFIER. Thus, eventType= 0x59 03 02 0a 06 = 2.9.3.2.10.6 = objectCreation

```
oamnot_debug:   event_time =
oamnot_debug:    Tag      Length          Value
oamnot_debug:    C5       e               “19960513135218”
```

From earlier we have:

eventTime [5] IMPLICIT GeneralizedTime OPTIONAL,

Thus, the Tag is C6 and the value is of type GeneralizedTime. Thus eventTime= 13th May 1996 13:52:18.

```
oamnot_debug:   event_info = NULL
```

There is NO eventInfo field in this M-Event-Report (eventInfo is defined as OPTIONAL).

```
oammsg_debug:   ***********************************************
```

The M-Event-Report is now complete.

```
oammsg_debug:   sent by OAM: create response
oammsg_debug:   message type = create response
oammsg_debug:   id = 7
oammsg_debug:   source =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   dest =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   linked = FALSE
```

C.1.1.2 M-Create Response

The MIS now sends a response to the original M-Create Indication. As this is the only response, linked is set to FALSE. The M-Create Response is defined from ITU X.711 as follows:

```
CreateResult ::= SEQUENCE {
        managedObjectClass      ObjectClass OPTIONAL,
        managedObjectInstance   ObjectInstance OPTIONAL,
        currentTime             [5] IMPLICIT GeneralizedTime
OPTIONAL,
        attributeList           [6] IMPLICIT SET OF Attribute
OPTIONAL
}
```

```
oammsg_debug:   oc =
oammsg_debug:    Tag   Length  Value
oammsg_debug:    C0    c        1.3.6.1.4.1.42.2.2.2.1.3.2
```

The above, oc, is the managedObjectClass of the source of this operation (M-Create Response). i.e. the Object Class of the new object that was just created. From earlier we see that: managedObjectClass is of type ObjectClass. And from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectClass is defined as follows:

```
ObjectClass ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Thus, from the em_debug trace we see that once again we use the globalForm CHOICE and that managedObjectClass = emApplicationInstance.

```
cammsg_debug: oi =
oammsg_debug: Tag         Length                     Value
oammsg_debug: C2          3d
oammsg_debug: Tag         Length                     Value
oammsg_deubg: SET         13
oammsg_debug: Tag         Length                     Value
oammsg_debug: SEQ         11
oammsg_debug: Tag         Length                     Value
oammsg_debug: OID         5                          2.9.3.2.7.4
oammsg_debug: Tag         Length                     Value
oammsg_debug: GRPH        8                          "dynamics"
oammsg_debug: Tag         Length                     Value
oammsg_debug: SET         11
oammsg_debug: Tag         Length                     Value
oammsg_debug: SEQ         f
oammsg_debug: Tag         Length                     Value
oammsg_debug: OID         5                          2.9.3.5.7.11
oammsg_debug: Tag         Length                     Value
oammsg_debug: GRPH        6                          "EM-MIS"
oammsg_debug: Tag         Length                     Value
oammsg_debug: SET         13
oammsg_debug: Tag         Length                     Value
oammsg_debug: SEQ         11
oammsg_debug: Tag         length                     Value
oammsg_debug: OID         c             1.2.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag         Length                     Value
oammsg_debug: INT        1  =                      5U
```

We can see from the earlier definition of EventReportArgument that the managedObjectInstance is of type ObjectInstance.

From ITU X.711
`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectInstance is defined as follows:

```
ObjectInstance ::= CHOICE {
        distinguishedName       [2] IMPLICIT DistinguishedName,
        nonSpecificForm         [3] IMPLICIT OCTET STRING,
        localDistinguishedName  [4] IMPLICIT RDNSequence
}
```

Note the context specific tag of each member of the choice. Looking at the em_debug output we can see the first tag is:

```
oammsg_debug:    Tag  Length      Value
oammsg_debug:    C2   13
```

Thus, what follows in DistinguishedName form. From the InformationFramework

`[/opt/SUNWconn/em/etc/asn1/infofw.asn1]`

is defined as follows: DistinguishedName ::= RDNSequence RDNSequence ::= SEQUENCE OF RelativeDistinguishedName. RelativeDistinguishedName ::= SET OF AttributeValueAssertion

```
AttributeValueAssertion ::=  SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
}

Thus DistinguishedName has the following syntax:

SEQUENCE OF SET OF SEQUENCE {
                             attributeId     OBJECT IDENTIFIER,
                       attributeValue  ANY DEFINED BY attributeID
             }
```

Thus, DistinguishedName will have the following form for its encoded value:

Tag Length Value

```
Tag          Length         Value
SEQ          A
Tag          Length         Value
SET          B
Tag          Length         Value
SEQUENCE     C
Tag          Length         Value
OID          D
Tag          Length         Value
X            E
```

The final Tag is X because it depends on the value of the previous OID. Now compare this with the em_debug output:

```
camnot_debug: oi =
camnot_debug: Tag       Length      Value
camnot_debug: C2        3d
LocalDistinguishedName form
camnot_debug: Tag       Length      Value
camnot_debug: SET       13
camnot_debug: Tag       Length      Value
camnot_debug: SEQ       11
camnot_debug: Tag       Length      Value
camnot_debug: OID       5            2.9.3.2.7.4
camnot_debug: Tag       Length      Value
camnot_debug: GRPH      8            “dynamics”
camnot_debug: Tag       Length      Value
camnot_debug: AttributeId  :  systemId
camnot_debug: AttributeValue: “dynamics” (uses the name CHOICE)
camnot_debug: Tag       Length      Value
camnot_debug: SET       11
camnot_debug: Tag       Length      Value
camnot_debug: SEQ       f
camnot_debug: Tag       Length      Value
camnot_debug: OID       5            2.9.3.5.7.11
camnot_debug: Tag       Length      Value
camnot_debug: GRPH      6            “EM-MIS”
AttrubuteId  :  subsystemId
AttributeValue: “EM-MIS”
camnot_debug: Tag       Length      Value
camnot_debug: SET       13
camnot_debug: Tag       Length      Value
camnot_debug: SEQ       11
camnot_debug: Tag       Length      Value
camnot_debug: OID       c         1.3.6.1.4.1.42.2.2.2.1.7.1
camnot_debug: Tag       Length      Value
camnot_debug: INT       1  =         5U
AttributeId  :  emApplicationID
AttributeValue: 5
```

Thus, the managedObjectInstance is:

`/systemId=name:”dynamics”/subsystemId=”EM-MIS”/emApplicationID=5`

```
oammsg_debug:    curr_time = NULL
```

currentTime is OPTIONAL, and is not present

```
oammsg_debug: attr_list  =
oammsg_debug: Tag      Length          Value
oammsg_debug:C6        c0
oammsg_debug: Tag      Length          Value
oammsg_debug: SEQ      11
oammsg_debug: Tag      Length          Value
oammsg_debug: C0       c         1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag      Length          Value
oammsg_debug: INT      1    =           5U
oammsg_debug: Tag      Length          Value
oammsg_debug: SEQ      1a
oammsg_debug: Tag      Length          Value
oammsg_debug: C0       c         1.3.6.1.4.1.42.2.2.2.1.7.2
oammsg_debug: Tag      Length          Value
oammsg_debug: GRPH     a                "Test_album"
oammsg_debug: Tag      Length          Value
oammsg_debug: SEQ      2c
oammsg_debug: Tag      Length          Value
oammsg_debug: C0       c         1.3.6.1.4.1.42.2.2.2.1.7.6
oammsg_debug: Tag      Length          Value
oammsg_debug: SEQ      1c
oammsg_debug: Tag      Length          Value
oammsg_debug: SET      1a
```

```
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      18
oammsg_debug: Tag      Length           Value
oammsg_debug: OID      c         1.3.66.1.4.1.42.2.2.2.1.7.7
oammsg_debug: Tag      Length           Value
oammsg_debug: GRPH     8                 “sudhakar”
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      10
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       c         1.3.6.1.4.1.42.2.2.2.1.7.11
oammsg_debug: Tag      Length           Value
oammsg_debug: SET      0
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      a
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       5         0x59 03 02 07 1f
oammsg_debug: Tag      Length           Value
oammsg_debug: ENUM     1     =           0
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      a
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       5         0x59 03 02 07 23
```

```
oammsg_debug: Tag      Length           Value
oammsg_debug: ENUM     1      =          1
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      9
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       5          0x59 03 02 07 38
oammsg_debug: Tag      Length           Value
oammsg_debug: C9       0
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      15
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       5          0x59 03 02 07 3f
oammsg_debug: Tag      Length           Value
oammsg_debug: OID      c          1.3.6.1.4.1.42.2.2.2.1.6.3
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      15
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       5          0x59 03 02 07 41
oammsg_debug: Tag      Length           Value
oammsg_debug: C0       c          1.3.6.1.4.1.42.2.2.2.1.3.2
```

The final parameter is the attributeList. This is defined in X.711 as follows:

attributeList [6] IMPLICIT SET OF Attribute OPTIONAL

And Attribute is defined as:

```
Attribute ::=  SEQUENCE {
        attributeId     AttributeId,
        attributeValue  ANY DEFINED BY  attributeId
}

and AttributeId is defined as:

AttributeId ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Remembering that [6] replaces SET OF (as in the earlier example), we have the following encoding for attributeList:

```
Tag         Length                Value
C6          A
Tag         Length                Value
SEQUENCE    B
Tag         Length                Value
C0 or C1    C
Tag         Length                Value
?           D
Tag         Length                Value
SEQUENCE    E
Tag         Length                Value
C0 or C1    F
Tag         Length                Value
?           G
```

and so on. One of the Tags is ? because it depends on the value of AttributeId. Also, the AttributeId can be either: globalForm tagged as C0 : type is OBJECT IDENTIFIER OR localForm tagged as C1 : type is INTEGER. Thus, we can now decode the attributeList as follows using the above and the Object Editor:

```
oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       11
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        c          1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag       Length                 Value
oammsg_debug: INT       1    =                  5U
oammsg_debug: Tag       Length                 Value

emApplicationID=5

oammsg_debug: SEQ       1a
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        c          1.3.6.1.4.1.42.2.2.2.1.7.2
oammsg_debug: Tag       Length                 Value
oammsg_debug: GRPH      a                       ”Test_album

emApplicaionType=”Test_album”

oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       2c
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        c          1.3.6.1.4.1.42.2.2.2.1.7.6
oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       1c
oammsg_debug: Tag       Length                 Value
oammsg_debug: SET       1a
oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       18
oammsg_debug: Tag       Length                 Value
oammsg_debug: OID       c          1.3.6.1.4.1.42.2.2.2.1.7.7
oammsg_debug: Tag       Length                 Value
oammsg_debug: GRPH      8                       “sudhakar”

emUserID= {{{emUserLogin, “sudhakar”}}}

or, if you prefer:
emUserID= /emUserLogin=”sudhakar”
```

```
oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       10
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        c          1.3.6.1.4.1.42.2.2.2.1.7.11
oammsg_debug: Tag       Length                 Value
oammsg_debug: SET       0
```

```
emSpecialEvents = {}

(the empty set)

oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       a
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        5                      0x59 03 02 07 1f
oammsg_debug: Tag       Length                 Value
oammsg_debug: ENUM      1       =              0

administrativeState=locked

oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       a
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        a                      0x59 03 02 07 23
oammsg_debug: Tag       Length                 Value
oammsg_debug: ENUM      1        =              1

operationalState=enabled

oammsg_debug: Tag       Length                 Value
oammsg_debug: SEQ       9
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        5                      0x59 03 02 07 38
oammsg_debug: Tag       Length                 Value
oammsg_debug: C9        0
```

discriminatorConstruct=

The syntax for discriminatorConstruct is:

DiscriminatorConstruct ::= discriminatorConstruct

```
CMISFilter ::= CHOICE {
        item    [8] FilterItem,
        and     [9] IMPLICIT SET OF CMISFilter,
        or      [10] IMPLICIT SET OF CMISFilter,
        not     [11] CMISFilter
}
```

So, the encoded value is:

and : {}

Thus,

discriminatorConstruct= and : {}

```
oammsg_debug: Tag       Length          Value
oammsg_debug: SEQ       15
oammsg_debug: Tag       Length          Value
oammsg_debug: C0        5               0x59 03 02 07 3f
oammsg_debug: Tag       Length          Value
oammsg_debug: OID       c              1.3.6.1.4.1.42.2.2.2.1.6.3

naemBinding=emApplicationInstance-emKernel

oammsg_debug: Tag       Length          Value
oammsg_debug: SEQ       15
oammsg_debug: Tag       Length          Value
oammsg_debug: C0        5               0x59 03 02 07 41
oammsg_debug: Tag       Length          Value
oammsg_debug: C0       c               1.3.6.1.4.1.42.2.2.2.1.3.2

objectClass=emapplicationInstance

oammsg_debug:  **********************************************
```

The M-Create Response is complete

```
oammsg_debug:   received by OAM: set request
oammsg_debug:   message type = set request
```

C.1.1.3 M-Set Indication

The MIS now receives an M-Set Indication. From ITU X.711 this has the following syntax:

```
SetArgument ::= SEQUENCE {
        COMPONENTS OF           BaseManagedObjectId,
        accessControl           [5] AccessControl OPTIONAL,
       synchronization [6] IMPLICIT CMISSync DEFAULT bestEffort,
        scope                   [7] Scope DEFAULT baseObject,
        filter          CMISFilter DEFAULT and : { },
        modificationList        [12] IMPLICIT SET OF SEQUENCE {
             modifyOperator [2] IMPLICIT ModifyOperator DEFAULT
replace,
                attributeId     AttributeId,
             attributeValue  ANY DEFINED BY attributeId OPTIONAL
                        --  absent for setToDefault
        }
}
```

and where BaseManagedObjectId is defined as follows:

```
BaseManagedObjectId ::= SEQUENCE {
        baseManagedObjectClass          ObjectClass,
        baseManagedObjectInstance       ObjectInstance
}
```

```
oammsg_debug:   id = 5
oammsg_debug:   source =
oammsg_debug:     aclass = APP, atag = 5
  aval =
“[0xff][0xff][0x2][0xc6][0x1a][0x1][0x4][0x7f][00][00][0x1]”
oammsg_debug:   dest =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   mode = CONFIRMED
oammsg_debug:   app_context = Du: no data unit allocated
oammsg_debug:   oc =
oammsg_debug:    Tag  LengthValue
oammsg_debug:    C0     5 0x59 03 04 03 2a
```

The above is the ManagedObjectClass. As before, this is in globalForm. The value decodes once again to emApplicationInstance.

The above oc is the baseManagedObjectClass of the source of this operation (M-Set Indication); that is. the Object Class of the object that the M-Set is to be applied to.

From earlier we see that baseManagedObjectClass is of type ObjectClassAnd from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectClass is defined as follows:

```
ObjectClass ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Thus, from the em_debug trace see see that once again we use the globalForm CHOICE and that managedObjectClass = emApplicationInstance.

```
oammsg_debug: oi =
oammsg_debug: Tag        Length                   Value
oammsg_debug: C4         28
oammsg_debug: Tag        Length                   Value
oammsg_debug: SET        11
oammsg_debug: Tag        Length                   Value
oammsg_debug: SEQ        f
oammsg_debug: Tag        Length                   Value
oammsg_debug: OID        5                         2.9.3.5.7.11
oammsg_debug: Tag        Length                   Value
oammsg_debug: GRPH       6                         “EM-MIS”
oammsg_debug: Tag        Length                   Value
oammsg_debug: SET        13
oammsg_debug: Tag        Length                   Value
oammsg_debug: SEQ        11
oammsg_debug: Tag        Length                   Value
oammsg_debug: OID        c             1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag        Length                   Value
oammsg_debug: INT        10x05
```

We can see from the earlier definition of SetArgument that the baseManagedObjectInstance is of type ObjectInstance.

From ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectInstance is defined as follows:

```
ObjectInstance ::= CHOICE {
        distinguishedName       [2] IMPLICIT DistinguishedName,
        nonSpecificForm         [3] IMPLICIT OCTET STRING,
        localDistinguishedName  [4] IMPLICIT RDNSequence
}
```

Note the context specific tag of each member of the choice. Looking at the em_debug output we can see the first tag is:

```
oammsg_debug:   superior_oi =
oammsg_debug:    Tag  Length         Value
oammsg_debug:    C4   13
```

Thus, what follows in localDistinguishedName form. Again we can see that localDistinguishedName is of ASN.1 Type RDNSequence. From the InformationFramework

`[/opt/SUNWconn/em/etc/asn1/infofw.asn1]`

RDNSequence is defined as follows: RDNSequence ::= SEQUENCE OF RelativeDistinguishedName and that: RelativeDistinguishedName ::= SET OF AttributeValueAssertion and that:

```
AttributeValueAssertion ::=  SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
}
```

Putting this all together we get:

```
ObjectInstance is the
        localDistinguishedName  [4] IMPLICIT
                                    SEQUENCE OF
                                    SET OF
                                    SEQUENCE {
                               attributeId    OBJECT IDENTIFIER,
                                    attributeValue  ANY DEFINED
BY attributeId
                                      }
```

Note – According to ASN.1, when a type defined in a CHOICE e.g. RDNSequence has a context specific Tag e.g. [4], this tag replaces the TAG of the type that follows (like the OBJECT IDENTIFIER example earlier). So in this case the TAG C4 replaces SEQUENCE OF in the encoded value of localDistinguishedName.

Thus, superiorObjectInstance will have the following form for its value:

```
Tag           Length                    Value
C4            A
Tag           Length                    Value
SET           B
Tag           Length                    Value
SEQUENCE      C
Tag           Length                    Value
OID           D
Tag           Length                    Value
X             E
```

The final Tag is X because it depends on the value of the previous OID. Now, compare this with the em_debug output:

```
oammsg_debug: oi =
oammsg_debug: Tag          Length               Value
oammsg_debug: C4           28
localDistinguishedName
oammsg_debug: Tag          Length               Value
oammsg_debug: SEt           11
oammsg_debug: Tag          Length               Value
oammsg_debug: SEQ          1
oammsg_debug: Tag          Length               Value
oammsg_debug: OID          5                    2.9.3.5.7.11
oammsg_debug: Tag          Length               Value
oammsg_debug: GRPH         6                    ”EM-MIS”
subsystemId=”EM-MIS”
oammsg_debug: Tag          Length               Value
oammsg_debug: SET          13
oammsg_debug: Tag          Length               Value
oammsg_debug: SEQ          11
oammsg_debug: Tag          Length               Value
oammsg_debug: OID          c           1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag          Length               Value
0x05
emApplicationID=5
```

Thus, the baseManagedObjectInstance is: subsystemId=”EM-MIS”/emApplicationID=5. Because we are using localDistinguishedName form, this name is relative to

```
/systemId=name:"hostname"
```

and we are specifying that the Object to be created will be contained by:

```
/systemId=name:"hostname"/subsystemId="EM-MIS"/emApplicationID=5
```

This is the object that was created by the M-Create received by the MIS earlier.

```
oammsg_debug:   scope = BASE_OBJECT
oammsg_debug:   filter = NULL
oammsg_debug:   access = NULL
oammsg_debug:   sync = BEST_EFFORT
```

The scope is Base Object Only. No filter or access is defined. The synchronization is best effort.

```
oammsg_debug:   modify_list =
oammsg_debug: Tag       Length             Value
oammsg_debug: C12       c
oammsg_debug: Tag       Length             Value
oammsg_debug: SEQ       a
oammsg_debug: Tag       Length             Value
oammsg_debug: C0        5                  0x59 030207 1f
oammsg_debug: Tag       Length             Value
oammsg_debug: ENUM      1                  "[0x1]"
```

The final parameter is the list of attributes to modify and the new values for these attributes (modificationList). From earlier this is defined as follows:

```
 modificationList
[12] IMPLICIT SET OF SEQUENCE {
             modifyOperator [2] IMPLICIT ModifyOperator DEFAULT
replace,
               attributeId     AttributeId,
             attributeValue  ANY DEFINED BY attributeId OPTIONAL
                        }
```

From X.711 ModifyOperator, AttributeId are defined as follows:

```
ModifyOperator ::= INTEGER {
        replace         (0),
        addValues       (1),
        removeValues    (2),
        setToDefault    (3)
}
```

and the earlier definition indicates that if this parameter is not present the default is replace.

Thus, we can decode the em_debug output as follows:

```
oammsg_debug:   modify_list =
oammsg_debug: Tag         Length                Value
oammsg_debug: C12         c

modificationList ( C12 of SEQUENCE)

oammsg_debug: Tag         Length                Value
oammsg_debug: SEQ         a
oammsg_debug: Tag         Length                Value
oammsg_debug: C0          5                    0x59 03 02 07 1f

attributeId = globalForm = administrativeState

oammsg_debug: Tag         Length                Value
oammsg_debug: ENUM        1                      "[0x1]"

attributeValue = 1 = unlocked

This modificationList consists of a single attribute and value

administrativeState = unlocked
```

```
oammsg_debug:
***************************************************************
oam_debug:      MODOP = 0, stat = 0
oam_info:       DiscriminatorAttrSecty:read val idx=1
oam_info:       DiscriminatorAttrSecty:write val
idx=1oamnot_debug:
```

C.1.1.4 M-Event-Report

M-Event-Report Message =

```
oamnot_debug:   message type = event_report request
oamnot_debug:   id = 15003840
oamnot_debug:   source =
oamnot_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oamnot_debug:   dest =
oamnot_debug:     aclass = DEF, atag = 1
  aval = Du: no data unit allocated
oamnot_debug:   remote =
oamnot_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
```

The MIS set the administrativeState sttribute to unlocked and now generated an M-Event-Report.

```
oamnot_debug:   mode = UNCONFIRMED
oamnot_debug:   app_context = Du: no data unit allocated
```

The mode is unconfirmed. The MIS does not expect a response. The syntax of an Event Report is (from ITU X.711): EventReportArgument ::=

```
SEQUENCE {
        managedObjectClass      ObjectClass,
        managedObjectInstance   ObjectInstance,
      eventTime            [5] IMPLICIT GeneralizedTime OPTIONAL,
        eventType               EventTypeId,
      eventInfo             [8] ANY DEFINED BY eventType OPTIONAL
}
```

```
oamnot_debug:   oc =
oamnot_debug:    Tag      Length             Value
oamnot_debug:    C0       c          1.3.6.1.4.1.42.2.2.2.1.3.2
```

The above, oc, is the managedObjectClass of the source of this operation (M-Event-Report). That is, the Object Class of the object for which the administrativeState sttribute was just set. From earlier we see that :managedObjectClass is of type ObjectClass. And from ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectClass is defined as follows:

ObjectClass ::=

```
CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Thus, from the em_debug trace we see that once again we use the globalForm CHOICE and that managedObjectClass = emApplicationInstance.

```
oamnot_debug:   oi =
oamnot_debug: Tag           Length              Value
oamnot_debug: C2            3d
oamnot_debug: Tag           Length              Value
oamnot_debug: SET           13
oamnot_debug: Tag           Length              Value
oamnot_debug: SEQ           11
oamnot_debug: Tag           Length              Value
oamnot_debug: OID           5                   2.9.3.2.7.4
oamnot_debug: Tag           Length              Value
oamnot_debug: GRPH          8                   “dynamics”
oamnot_debug: Tag           Length              Value
oamnot_debug: SET           11
oamnot_debug: Tag           Length              Value
oamnot_debug: SEQ           1
oamnot_debug: Tag           Length              Value
oamnot_debug: OID           5                   2.9.3.5.7.11
oamnot_debug: Tag           Length              Value
oamnot_debug: GRPH          6                   “EM-MIS”
oamnot_debug: Tag           Length              Value
oamnot_debug: SET           13
oamnot_debug: Tag           Length              Value
oamnot_debug: SEQ           11
oamnot_debug: Tag           Length              Value
oamnot_debug: OID          c          1.3.6.1.4.1.42.2.2.2.1.7.1
oamnot_debug: Tag           Length              Value
oamnot_debug: INT           1     =             5
```

We can see from the earlier definition of EventReportArgument that the managedObjectInstance is of type ObjectInstance.

From ITU X.711

`[/opt/SUNWconn/em/etc/asn1/x711.asn1]`

ObjectInstance is defined as follows:

```
ObjectInstance ::= CHOICE {
        distinguishedName       [2] IMPLICIT DistinguishedName,
        nonSpecificForm         [3] IMPLICIT OCTET STRING,
        localDistinguishedName  [4] IMPLICIT RDNSequence
}
```

Note the context specific tag of each member of the choice. Looking at the em_debug output, we can see the first tag is:

```
oammsg_debug:    Tag   Length            Value
oammsg_debug:    C2    13
```

Thus, what follows in DistinguishedName form. From the InformationFramework

[/opt/SUNWconn/em/etc/asn1/infofw.asn1]

is defined as follows:

```
DistinguishedName ::= RDNSequence RDNSequence ::= SEQUENCE OF
RelativeDistinguishedName
RelativeDistinguishedName ::= SET OF AttributeValueAssertion
AttributeValueAssertion ::=  SEQUENCE {
        attributeId     OBJECT IDENTIFIER,
        attributeValue  ANY DEFINED BY attributeId
}
```

Thus, DistinguishedName has the following syntax:

```
SEQUENCE OF SET OF SEQUENCE {
                             attributeId     OBJECT IDENTIFIER,
                       attributeValue  ANY DEFINED BY attributeId
                             }
```

Thus, DistinguishedName will have the following form for its encoded value:

```
Tag               Length                  Value
SEQ               A
Tag               Length                  Value
SET               B
Tag               Length                  Value
SEQUENCE          C
Tag               Length                  Value
OID               D
Tag               Length                  Value
X                 E
```

The final Tag is X because it depends on the value of the previous OID. Compare this with the em_debug output:

```
camnot_debug: oi =
camnot_debug: Tag        Length                 Value
camnot_debug: C2         3d

DistinguishedName form

camnot_debug: Tag        Length                 Value
camnot_debug: SET        13
camnot_debug: Tag        Length                 Value
camnot_debug: SEQ        11
camnot_debug: Tag        Length                 Value
camnot_debug: OID        5                      2.9.3.2.7.4
camnot_debug: Tag        Length                 Value
camnot_debug: GRPH       8                      "dynamics"

AttributeId   : systemId
AttributeValue: "dynamics" (uses the name CHOICE)

camnot_debug: Tag        Length                 Value
camnot_debug: SET        11
camnot_debug: Tag        Length                 Value
camnot_debug: SEQ        f
camnot_debug: Tag        Length                 Value
camnot_debug: OID        5                      2.9.3.5.7.11
camnot_debug: Tag        Length                 Value
camnot_debug: GRPH       6                      "EM-MIS"
```

```
AttributeId   : subsystemId
AttributeValue: “EM-MIS”

camnot_debug: Tag        Length                  Value
camnot_debug: SET        13
camnot_debug: Tag        Length                  Value
camnot_debug: SEQ        11
camnot_debug: Tag        Length                  Value
camnot_debug: OID        c            1.3.6.1.4.1.42.2.2.2.1.7.1
camnot_debug: Tag        Length                  Value
camnot_debug: INT        1       =               5U

AttributeId   : emApplicationID
AttributeValue: 5
```

Thus, the managedObjectInstance is:

`/systemId=name:”dynamics”/subsystemId=”EM-MIS”/emApplicationID=5`

```
oamnot_debug:   event_type =
oamnot_debug:    Tag  Length    Value
oamnot_debug:    C6     5               2.9.3.2.10.1
```

From earlier we have:

eventType EventTypeId,

and from X.711 EventTypeId has the following syntax:

```
EventTypeId ::= CHOICE {
        globalForm      [6] IMPLICIT OBJECT IDENTIFIER,
        localForm       [7] IMPLICIT INTEGER
}
```

As the first Tag from the em_debug trace is C6, we are using globalForm and the value of type OBJECT IDENTIFIER. Thus, eventType= 2.9.3.2.10.1 = attributeValueChange

```
oamnot_debug:   event_time =
oamnot_debug:    Tag       Length            Value
oamnot_debug:    C5        e                 “19960513135218”
```

From earlier we have:

eventTime [5] IMPLICIT GeneralizedTime OPTIONAL,

Thus, the Tag is C6 and the value is of type GeneralizedTime. Thus, eventTime= 13th May 1996 13:52:18

```
oamnot_debug:   event_info =
camnot_debug: Tag           Length                Value
camnot_debug: SEQ           21
camnot_debug: Tag           Length                Value
camnot_debug: ENUM          1       =             1
camnot_debug: Tag           Length                Value
camnot_debug: C1            7
camnot_debug: Tag           Length                Value
camnot_debug: C0           5                     0x59 03 02 07 1f
camnot_debug: Tag           Length                Value
camnot_debug: SET           13
camnot_debug: Tag           Length                Value
camnot_debug: SEQ           11
camnot_debug: Tag           Length                Value
camnot_debug: C0           5                     0x59 03 02 07 1f
camnot_debug: Tag           Length                Value
camnot_debug: C1            3
camnot_debug: Tag           Length                Value
camnot_debug: ENUM          1         =           0
camnot_debug: Tag           Length                Value
camnot_debug: C2            3
camnot_debug: Tag           Length                Value
camnot_debug: ENUM          1        =            1
```

For an attributeValueChange notification, the contents of the eventInfo field are defined as (from /opt/SUNWconn/em/etc/asn1/dmi.asn1):

```
AttributeValueChangeInfo ::= SEQUENCE
{
    sourceIndicator                     SourceIndicator OPTIONAL,
    attributeIdentifierList     [1]     AttributeIdentifierList
OPTIONAL,
    attributeValueChangeDefinition
AttributeValueChangeDefinition,
    notificationIdentifier              NotificationIdentifier
OPTIONAL,
    correlatedNotifications     [2]     CorrelatedNotifications
OPTIONAL,
    additionalText                      AdditionalText OPTIONAL,
    additionalInformation       [3]     AdditionalInformation
OPTIONAL
}
```

Thus, we can decode the AttributeValueChangeInfo as follows:

```
oamnot_debug:   event_info =
oamnot_debug:   Tag             Length          Value
oamnot_debug:   SEQ             21
AttributeValueChangeInfo
oamnot_debug:    Tag            Length           Value
oamnot_debug:    ENUM           1       =        1
```

The first component in the sequence has Tag ENUMERATE and, therefore, must be sourceIndicator. SourceIndicator is defined as:

```
SourceIndicator ::= ENUMERATED
{
    resourceOperation   (0),
    managementOperation (1),
    unknown             (2)
}
```

Thus, sourceIndicator = managementOperation

```
oamnot_debug:  Tag  Length         Value
oamnot_debug:  C1   7
oamnot_debug:  Tag  Length         Value
oamnot_debug:  C0     5             0x59 03 02 07 1f
```

The next item in the Sequence has a Context Specific Tag of 1. Thus, we have attributeIdentifierList [1] AttributeIdentifierList OPTIONAL. From dmi.asn1 the AttributeIdentifierList is defined as follows:

AttributeIdentifierList ::= SET OF AttributeId

And AttributeId (from X.71) is:

Thus, the general encoded form will be:

```
AttributeId ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

```
Tag     Length         Value
C1      A
Tag     Length         Value
C0/C1   B
```

From em_debug:

```
oamnot_debug:     C0     5               0x59 03 02 07 1f
```

We have the globalForm choice. Thus, attributeIdentifierList = 0x59 03 02 07 1f = administrativeState, which is the attribute we just set.

Note that AttributeValueChangeDefinition is not OPTIONAL. It is defined in DMI as:

```
AttributeValueChangeDefinition ::= SET OF SEQUENCE
{
    attributeID                      AttributeId,
    oldAttributeValue   [1]      ANY DEFINED BY attributeID
OPTIONAL,
    newAttributeValue   [2]      ANY DEFINED BY attributeID
}
```

Thus, we have:

What follows is attributeID, oldAttributeValue, newAttributeValue

```
camnot_debug: Tag          Length                  Value
camnot_debug: SET          13
camnot_debug: Tag          Length                  Value
camnot_debug: SEQ          11
```

```
camnot_debug: Tag          Length                  Value
camnot_debug: CO          5                       0x59 03 02 07 1f

AttributeId (globalForm) = administrativeState

camnot_debug: Tag          Length                  Value
camnot_debug: C1           3
camnot_debug: Tag          Length                  Value
camnot_debug: ENUM         1     =                 0

oldAttributeValue = locked

camnot_debug: Tag          Length                  Value
camnot_debug: C2           3
camnot_debug: Tag          Length                  Value
camnot_debug: ENUM         1        =              1

Tag C2 = newAttributeValue = unlocked

oammsg_debug:   ************************************************
```

The M-Event-Report is now complete.

```
oammsg_debug:   sent by OAM: set response
oammsg_debug:   message type = set response
oammsg_debug:   id = 5
oammsg_debug:   source =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   dest =
oammsg_debug:     aclass = APP, atag = 5
  aval =
"[0xff][0xff][0x2][0xc6][0x1a][0x1][0x4][0x7f][0x0][0x0][0x1]"
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
```

C.1.1.5 M-Set Response

The MIS now sends a response to the M-Set. This is defined in X.711 to be:

```
SetResult ::= SEQUENCE {
        managedObjectClass      ObjectClass OPTIONAL,
        managedObjectInstance   ObjectInstance OPTIONAL,
        currentTime             [5] IMPLICIT GeneralizedTime
OPTIONAL,
        attributeList           [6] IMPLICIT SET OF Attribute
OPTIONAL
}
```

```
oammsg_debug:   linked = FALSE
```

This is the only response. That is, it is not linked with others.

```
oammsg_debug:   oc =
oammsg_debug:    Tag   Length      Value
oammsg_debug:    C0    c           1.3.6.1.4.1.42.2.2.2.1.3.2
```

Once again: managedObjectClass = emApplicationInstance

```
oammsg_debug:   oi =
oammsg_debug: Tag      Length           Value
oammsg_debug: C2       3d
oammsg_debug: Tag      Length           Value
oammsg_debug: SET      13
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      11
oammsg_debug: Tag      Length           Value
oammsg_debug: OID      5                2.9.3.2.7.4
oammsg_debug: Tag      Length           Value
oammsg_debug: GRPH     8                "dynamics"
oammsg_debug: Tag      Length           Value
oammsg_debug: SET      11
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      f
oammsg_debug: Tag      Length           Value
oammsg_debug: OID      5                2.9.3.5.7.11
oammsg_debug: Tag      Length           Value
oammsg_debug: GRPH     6                "EM-MIS"
oammsg_debug: Tag      Length           Value
oammsg_debug: SET      13
oammsg_debug: Tag      Length           Value
oammsg_debug: SEQ      11
oammsg_debug: Tag      Length           Value
oammsg_debug: OID      c             1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag      Length           Value
oammsg_debug: INT      1                0x05
```

Once again: managedObjectInstance =

```
/systemId=name:"dynamics"/subsystemId="EM-
MIS"/emApplicationInstance:5
```

```
oammsg_debug:   curr_time = NULL
```

currentTime is NOT present (NULL)

The attributeList is defined as:

```
oammsg_debug:   attr_list =
oammsg_debug: Tag       Length           Value
oammsg_debug: C6        c
oammsg_debug: Tag       Length           Value
oammsg_debug: SEQ       a
oammsg_debug: Tag       Length           Value
oammsg_debug: C0        5                 0x59 03 02 07 1f
oammsg_debug: Tag       Length           Value
oammsg_debug: ENUM      1       =         1
```

attributeList [6] IMPLICIT SET OF Attribute OPTIONAL. Thus:

```
oammsg_debug: Tag       Length         Value
oammsg_debug: C6        c

attributeList

oammsg_debug: Tag       Length         Value
oammsg_debug: SEQ       a
oammsg_debug: Tag       Length         Value
oammsg_debug: C0        5              0x59 03 02 07 1f

administrativeState =

oammsg_debug: Tag       Length         Value
oammsg_debug: ENUM      1     =        1

unlocked

oammsg_debug:   ***********************************************
```

The M-Set operation is now complete

C.1.1.6 M-GET Request

The PMI call image.boot(attr=administrativeState) call results in the PMI client issuing a M-GET request of the attribute administrativeState of the log object instance /systemId=<hostname>/logid=\"AlarmLog\".

```
#
#
#  IMAGE_BOOT
#
#
#
# oammsg_debug:
****************************************************************
oammsg_debug: received by OAM: get request
oammsg_debug: message type = get request
oammsg_debug: id = 14
oammsg_debug: source =
oammsg_debug:  aclass = APP, atag = 5 aval = “[0xff] [0xff] [0xc6]
[0x1s] [0x1] [0x4] [0x7f] [00] [00] [0x1]”
oammsg_debug: dest =
oammsg_debug:  aclass = DEF, atag =0 aval = Du: no data unit is
allocated
oammsg_debug: remote =
oammsg_debug:  aclass = DEF, atag =0 aval= Du: no data unit
allocated
oammsg_debug: mode = CONFIRMED
```

The MIS receives an M-Get indication, and a response is expected. X.711 defines the M-Get Request/Indication as follows:

```
GetArgument ::=  SEQUENCE {
        COMPONENTS OF           BaseManagedObjectId,
        accessControl           [5] AccessControl OPTIONAL,
       synchronization [6] IMPLICIT CMISSync DEFAULT bestEffort,
        scope                   [7] Scope DEFAULT baseObject,
        filter          CMISFilter DEFAULT and : {},
       attributeIdList        [12] IMPLICIT SET OF AttributeId
OPTIONAL
}
```

and where BaseManagedObjectId is defined as:

```
BaseManagedObjectId ::= SEQUENCE {
        baseManagedObjectClass          ObjectClass,
        baseManagedObjectInstance       ObjectInstance
}
```

```
oammsg_debug:   app_context = Du: no data unit allocated
oammsg_debug:   oc =
oammsg_debug:    Tag  Length      Value
oammsg_debug:    C0   5            0x59 03 04 03 2a
```

By a similiar method as before: baseManagedObjectClass = 0x59 03 04 03 2a = 2.9.3.4.3.42 = actualClass. ActualClass is a wildcard. This indicates to the MIS that the application does not know the ObjectClass.

```
oammsg_debug:   oi =
oammsg_debug: Tag        Length        Value
oammsg_debug: C2         2a
oammsg_debug: Tag        Length        Value
oammsg_debug: SET        13
oammsg_debug: Tag        Length        Value
oammsg_debug: SEQ        11
oammsg_debug: Tag        Length        Value
oammsg_debug: OID        5              2.9.3.2.7.4
oammsg_debug: Tag        Length        Value
oammsg_debug: GRPH       8              "dynamics"
oammsg_debug: Tag        Length        Value
oammsg_debug: SET        13
oammsg_debug: Tag        Length        Value
oammsg_debug: SEQ        11
oammsg_debug: Tag        Length        Value
oammsg_debug: OID        5              2.9.3.2.7.2
oammsg_debug: Tag        Length        Value
oammsg_debug: GRPH       8             "AlarmLog'
```

By a similiar method as before: baseManagedObjectInstance is in distinguishedName form

`= /systemId=name:"dynamics"/logId="AlarmLog"`

```
oammsg_debug:   scope = BASE_OBJECT
oammsg_debug:   filter = NULL
oammsg_debug:   access = NULL
oammsg_debug:   sync = BEST_EFFORT
```

Scope is the base object only. No filter is present. No access is present. The synchronization is Best Effort.

```
oammsg_debug:   attr_id_list =
oammsg_debug:    Tag       Length          Value
oammsg_debug:    C12        7
oammsg_debug:    Tag       Length           Value
oammsg_debug:   C0        5                    0x59 03 02 07 1f
```

From earlier the attributeIdList is defined as attributeIdList [12] IMPLICIT SET OF AttributeId OPTIONAL, and AttributeId is defined as:

```
AttributeId ::= CHOICE {
        globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
        localForm       [1] IMPLICIT INTEGER
}
```

Thus, the encoded form will be:

```
Tag          Length                  Value
C12          A
Tag          Length                  Value
C0/C1        B
Tag          Length                  Value
C0/C1        C
```

Thus, we have:

```
oammsg_debug:             C0      5 0x59 03 02 07 1f
```

the AttributeId in globalForm = 0x59 03 02 07 1f = administrativeState

```
oammsg_debug:   ************************************************
DiscriminatorAttrSecty:read val idx=1
oammsg_debug:   ************************************************
```

```
oammsg_debug:   sent by OAM: get response
oammsg_debug:   message type = get response
oammsg_debug:   id = 14
oammsg_debug:   source =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   dest =
oammsg_debug:     aclass = APP, atag = 5
  aval =
“[0xff][0xff][0x2][0xc6][0x1a][0x1][0x4][0x7f][0x0][0x0][0x1]”
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   linked = FALSE
```

C.1.1.7 M-Get Response

The M-Get is received by the MIS, and it issues an M-Get Response. There is only a single response so linked = FALSE. From X.711 the structure of the response is:

```
GetResult ::=  SEQUENCE {
        managedObjectClass      ObjectClass OPTIONAL,
        managedObjectInstance   ObjectInstance OPTIONAL,
        currentTime             [5] IMPLICIT GeneralizedTime
OPTIONAL,
        attributeList           [6] IMPLICIT SET OF Attribute
OPTIONAL
}
```

```
oammsg_debug:   oc =
oammsg_debug: Tag       Length                 Value
oammsg_debug: C0        5                       0x59 03 02 03 06
```

By a similiar method as before: managedObjectClass = 0x59 03 04 03 2a = 2.9.3.4.3.42 = actualClass. actualClass is a wildcard. This indicates to the application that it should not care about this field.

```
oammsg_debug:   oi =
oammsg_debug: Tag        Length           Value
oammsg_debug: C2         2a
oammsg_debug: Tag        Length           Value
oammsg_debug: SET        13
oammsg_debug: Tag        Length           Value
oammsg_debug: SEQ        11
oammsg_debug: Tag        Length           Value
oammsg_debug: OID        5                2.9.3.2.7.4
oammsg_debug: Tag        Length           Value
oammsg_debug: GRPH       8                “dynamics”
oammsg_debug: Tag        Length           Value
oammsg_debug: SET        13
oammsg_debug: Tag        Length           Value
oammsg_debug: SEQ        11
oammsg_debug: Tag        Length           Value
oammsg_debug: OID        5                2.9.3.2.7.2
oammsg_debug: Tag        Length           Value
oammsg_debug: GRPH       8                “AlarmLog”
```

By a similiar method as before: baseManagedObjectInstance is in distinguishedName form

`= /systemId=name:”dynamics”/logId=”AlarmLog”`

```
oammsg_debug:   curr_time = NULL
```

current time is NOT present.

```
oammsg_debug:   attr_list =
oammsg_debug: Tag         Length             Value
oammsg_debug: C6          c
oammsg_debug: Tag         Length             Value
oammsg_debug: SEQ         a
oammsg_debug: Tag         Length             Value
oammsg_debug: C0          5                  0x59 03 02 07 1f
oammsg_debug: Tag         Length             Value
oammsg_debug: ENUM        1       =          0
```

By a similiar method as before: attributeList is administrateState = locked.

```
oammsg_debug:   ***********************************************
```

C.1.1.8 *M-DELETE Request*

The M-Get operation is complete (The PMI Image::boot() is complete). Before the PMI client application exits, the PMI call platform.disconnect()results in a M-DELETE request to the MIS to delete the corresponding emApplicationInstance object.

```
#
#
# PLAT_DISCONNECT
#
# oammsg_debug: ************************************************
```

```
oammsg_debug:   received by OAM: delete request
oammsg_debug:   message type = delete request
oammsg_debug:   id = 8
oammsg_debug:   source =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   dest =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   mode = CONFIRMED
oammsg_debug:   app_context = Du: no data unit allocated
```

The MIS receives an M-Delete indication, and a response is expected. X.711 defines the M-Delete Request/Indication as follows:

```
DeleteArgument ::= SEQUENCE {
        COMPONENTS OF   BaseManagedObjectId,
        accessControl   [5] AccessControl OPTIONAL,
       synchronization [6] IMPLICIT CMISSync DEFAULT bestEffort,
        scope           [7] Scope DEFAULT baseObject,
        filter          CMISFilter DEFAULT and : {}
}
```

and where BaseManagedObjectId is defined as:

```
BaseManagedObjectId ::= SEQUENCE {
        baseManagedObjectClass          ObjectClass,
        baseManagedObjectInstance       ObjectInstance
}
```

```
oammsg_debug:   oc =
oammsg_debug:    Tag      Length        Value
oammsg_debug:    C0       c            1.3.6.1.4.1.42.2.2.2.1.3.2
```

The above is the baseManagedObjectClass. As before, this is in globalForm. The value decodes once again to emApplicationInstance.

```
oammsg_debug:   oi =
oammsg_debug: Tag         Length           Value
oammsg_debug: C4          28
oammsg_debug: Tag         Length           Value
oammsg_debug: SET         11
oammsg_debug: Tag         Length           Value
oammsg_debug: SEQ         f
oammsg_debug: Tag         Length           Value
oammsg_debug: OID         5                 2.9.3.5.7.11
oammsg_debug: Tag         Length           Value
oammsg_debug: GRPH        6                 "EM-MIS"
oammsg_debug: Tag         Length           Value
oammsg_debug: SET         13
oammsg_debug: Tag         Length           Value
oammsg_debug: SEQ         11
oammsg_debug: Tag         Length           Value
oammsg_debug: OID         c          1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: INT         1       =         5U
```

Once again, we have baseManagedObjectInstance using the localDistinguishedName CHOICE. Once against this is

`subsystemId="EM-MIS"/emApplicationID=5`

Because we are using localDistinguishedName form this name is relative to

`/systemId=name:"dynamics"`

and we are specifiying that the Object to be deleted will be:

`/systemId=name:"dynamics"/subsystemId="EM-MIS"/emApplicationID=5`

This is the object instance that was created when the Platform::connect() was performed.

```
oammsg_debug:   scope = BASE_OBJECT
oammsg_debug:   filter = NULL
oammsg_debug:   access = NULL
oammsg_debug:   sync = BEST_EFFORT
oammsg_debug:   ***********************************************
```

Scope is this object only. No filter is present. No access is present. The synchronization is Best Effort.

```
oammsg_debug:   *****************************************
oammsg_debug:   sent by OAM: delete response
oammsg_debug:   message type = delete response
oammsg_debug:   id = 8
oammsg_debug:   source =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   dest =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   remote =
oammsg_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oammsg_debug:   linked = FALSE
```

The M-Delete is received by the MIS. It deletes the emApplicationInstance object that represented this PMI application and it issues an M-Delete Response. There is only a single response so linked = FALSE. From X.711 the structure of the response is:

```
DeleteResult ::= SEQUENCE {
        managedObjectClass      ObjectClass OPTIONAL,
        managedObjectInstance   ObjectInstance OPTIONAL,
      currentTime          [5] IMPLICIT GeneralizedTime OPTIONAL
}
```

```
oammsg_debug:   oc =
oammsg_debug:    Tag       Length       Value
oammsg_debug:   C0       c          1.3.6.1.4.1.42.2.2.2.1.3.2
```

managedObjectClass = emApplicationInstance

```
oammsg_debug:   oi =
oammsg_debug: Tag          Length        Value
oammsg_debug: C2           3d
oammsg_debug: Tag          Length        Value
oammsg_debug: SET          13
oammsg_debug: Tag          Length        Value
oammsg_debug: SEQ          11
oammsg_debug: Tag          Length        Value
oammsg_debug: OID          5             2.9.3.2.7.4
oammsg_debug: Tag          Length        Value
oammsg_debug: GRPH         8             “dynamics”
oammsg_debug: Tag          Length        Value
oammsg_debug: SET          11
oammsg_debug: Tag          Length        Value
oammsg_debug: SEQ          f
oammsg_debug: Tag          Length        Value
oammsg_debug: OID          5             2.9.3.5.7.11
oammsg_debug: Tag          Length        Value
oammsg_debug: GRPH         6             “EM-MIS”
oammsg_debug: Tag          Length        Value
oammsg_debug: SET          13
oammsg_debug: Tag          Length        Value
oammsg_debug: SEQ          11
oammsg_debug: Tag          Length        Value
oammsg_debug: OID          c          1.3.6.1.4.1.42.2.2.2.1.7.1
oammsg_debug: Tag          Length        Value
oammsg_debug: INT          1             0x05
```

managedObjectInstance uses the distinguishedName CHOICE

`= /systemId=name:”dynamics”/subsystemId=”EM-MIS’/emApplicationID=5`

```
oammsg_debug:   curr_time = NULL
oammsg_debug:   ****************************************
```

currentTime is not present (NULL)

```
oamnot_debug:   M-Event-Report Message =
oamnot_debug:   message type = event_report request
oamnot_debug:   id = 14986600
oamnot_debug:   source =
oamnot_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oamnot_debug:   dest =
oamnot_debug:     aclass = DEF, atag = 1
  aval = Du: no data unit allocated
oamnot_debug:   remote =
oamnot_debug:     aclass = DEF, atag = 0
  aval = Du: no data unit allocated
oamnot_debug:   mode = UNCONFIRMED
oamnot_debug:   app_context = Du: no data unit allocated
```

The MIS generates an Event Report and does not expect a response. X.711 defines the structure as follows:

```
EventReportArgument ::= SEQUENCE {
        managedObjectClass      ObjectClass,
        managedObjectInstance   ObjectInstance,
      eventTime            [5] IMPLICIT GeneralizedTime OPTIONAL,
        eventType               EventTypeId,
      eventInfo             [8] ANY DEFINED BY eventType OPTIONAL
}
```

```
oamnot_debug:   oc =
oamnot_debug:    Tag    Length       Value
oamnot_debug:    C0     c             1.3.6.1.4.1.42.2.2.2.1.3.2
```

managedObjectClass uses the globalForm CHOICE = emApplicationInstance

```
oamnot_debug:   oi =
camnot_debug: Tag          Length        Value
camnot_debug: C2           3d
camnot_debug: Tag          Length        Value
camnot_debug: SET          13
camnot_debug: Tag          Length        Value
camnot_debug: SEQ          11
camnot_debug: Tag          Length        Value
camnot_debug: OID          5             2.9.3.2.7.4
camnot_debug: Tag          Length        Value
camnot_debug: GRPH         8             "dynamics"
camnot_debug: Tag          Length        Value
camnot_debug: SET          11
camnot_debug: Tag          Length        Value
camnot_debug: SEQ          f
camnot_debug: Tag          Length        Value
camnot_debug: OID          5             2.9.3.5.7.11
camnot_debug: Tag          Length        Value
camnot_debug: GRPH         6             "EM-MIS"
camnot_debug: Tag          Length        Value
camnot_debug: SET          11
camnot_debug: Tag          Length        Value
camnot_debug: SEQ          11
camnot_debug: Tag          Length        Value
camnot_debug: OID          c         1.3.6.1.4.1.42.2.2.2.1.7.1
camnot_debug: Tag          Length        Value
camnot_debug: INT          1             0x05
```

managedObjectInstance uses the distinguishedName CHOICE

`= /systemId=name:"dynamics"/subsystemId="EM-MIS'/emApplicationID=5`

```
oamnot_debug:   event_type =
oamnot_debug:    Tag        Length      Value
oamnot_debug:    C6         5            0x59 03 02 0a 07
```

eventType = 0x59 03 02 0a 07 = 2.9.3.2.10.7 = objectDeletion

```
oamnot_debug:   event_time =
oamnot_debug:    Tag  Length        Value
oamnot_debug:    C5   e              “19960513135249”
oamnot_debug:   event_info = NULL
```

eventTime = 13th May 1996 12:52:49. The PMI program is now complete.

Topology Database Architecture D

D.1 Introduction

The purpose of the topology database is to store topological information about the managed networked environments. Topological information is in the form of objects which represent a topological node, view, viewnodes, and type. The topology database consists of a topoTypeDB, a topoNodeDB, and a topoViewDB and is used by Enterprise Manager applications to manage the user’s networks.

The topoTypeDB object class contains the general relationship or rules between objects (which represent a topoType). In other words, topoTypeDB contains a list of object types.

The topoNodeDB object class contains a flat layout of the objects in the managed networked environment, that is, lists of all nodes and their attributes. The topoViewDB object class contains views of the objects in the managed networked environment, that is, a list of all views.

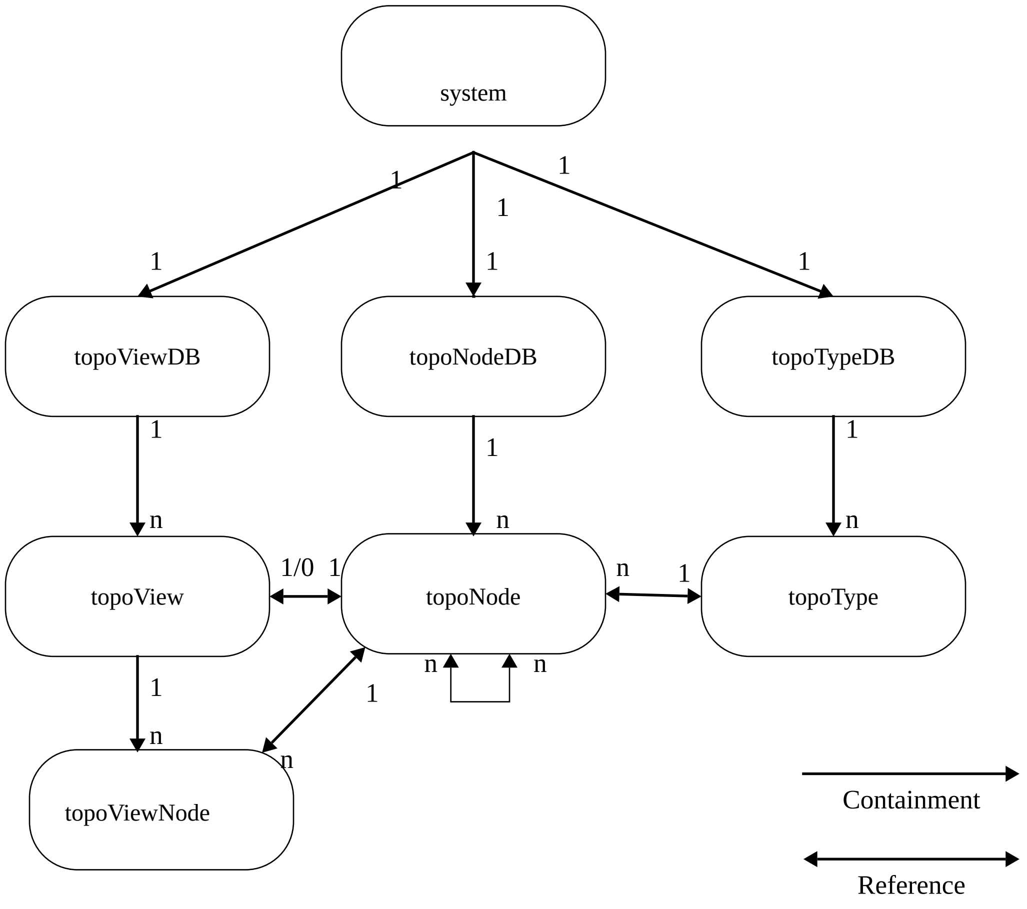


Figure D-1 Object Relationship Diagram

D.2 Object Relationship Diagram Description

D.2.1 Containment

- TopoViewDB, topoNodeDB, and topoTypeDB are contained within system.
- A topoView is contained within topoViewDB.
- A topoViewNode is contained within topoView.
- A topoNode is contained within topoNodeDB.
- A topoType is contained within topoTypeDB.

D.2.2 Reference

- A topoView must reference a topoNode.
- Only one topoView can reference a single topoNode.
- A topoViewNode must reference a topoNode.
- One or more topoViewNodes can reference a single topoNode.
- A topoNode can reference other topoNodes as topoNodeParents or topoNodeChildren.
- A topoNode must reference one topoType.
- A topoType can be referenced by multiple topoNodes.

D.3 topoTypeDB

TopoTypeDB is a managed object class that acts as a container for all topoType objects. The topoTypeDB object class is named under the system object and only one instance of a topoTypeDB object class can be created under a system.

D.3.1 topoType

TopoType is an object class that is named under the topoTypeDB object class.

D.4 topoNodeDB

TopoNodeDB is a managed object class that acts as the container for all topoNode objects. This object class lists all nodes and their attributes. The topoNodeDB object class is named under the system object and only one instance of the topoNodeDB object class can be created under a system.

D.4.1 topoNode

TopoNode is an object class that is named under the topoNodeDB object class.

The topoNode object class has the following features:

- TopoNode can be positioned in multiple view.

  This attribute is allowed since the behavior of topoTypeLegalChildren is checked for all parents specified by the attribute. The ASN1 syntax of topoNodeParents is a set of topoNodeId.

- Special secondary index queries can be done with actions.
- TopoNode objects can be renamed.

  The topoNodeName attribute is unique across all topoNodes under the same topoNodeDB. The reason that topoNodeName is not used as the naming attribute is to allow renaming of topoNode objects.

Note – If a topoNode object is renamed, it's new name cannot be the same as the name of an existing node.

- Data integrity between topoNode, topoView, and topoViewNode is maintained.

  Data integrity is maintained by the behavior of the class. Once a new topoNode is created, if the type of the topoNode can contain other topoNodes, the MIS will create a topoView object associated to the topoNode. If a topoNode is deleted, all topoView and topoViewNode objects associated with the topoNode are automatically removed by the MIS.

- TopoViewNode objects are automatically updated.

  When a new parent is added to the topoNodeParents attribute, the MIS creates a topoViewNode object associating to the topoNode under a topoView object which associates to the new parent. When an old parent is removed from topoNodeParents, the MIS deletes the topoViewNode object associating to the topoNode from the topoView object which associates to the old parent.

Note – If you want to move or place a topoNode in a different or another view, be sure to change the topoNode's topoNodeParents attribute.

The topoNodeChildren attribute is a reverse relationship attribute of the topoNodeParents attribute. It specifies all the topoNode children that are contained by this topoNode.

- Propagation severity of topoNode objects can be tracked and propagation can be controlled.

  The tracking of the propagated severity is done with the topoNodePropogateSeverity attribute. This attribute is the maximum value of the topoNodeSeverity of the topoNode and the topoNodePropagatedSeverity of all its children.

  To control the propagation locally, the topoNode's topoNodePropagateUp attribute is used. To turn the propagation off for the entire topology database, the topoNodeDB's topoStatePropagation attribute is used. By default, the global propagation flag is set on.

  A topoNode propagates its current severity to its parents only if its topoNodePropagateUp flag is on *and* topoNodeDB's topoStatePropagation flag is on.

D.5 topoViewDB

TopoViewDB is a managed object class that acts as the container for all topoView objects. TopoViewDB lists all views; each of these views are called a topoView. The topoViewDB object class is named under the system object and only one instance of the topoViewDB object class can be created under a system.

A view is a graphical representation of a set of related managed objects. For example, in a network that contains multiple subnetworks, the network might be one view that contains the subnetwork views and each subnetwork within it might constitute separate views.

D.5.1 topoView

TopoView is an object class that lists views and is named under the topoViewDB object class. Each topoView object is called a topoViewNode.

Each instance of the topoView class represents a view in em_viewer to display objects that are in the view and store attributes that are related to the view. TopoView objects show relationships and hierarchy between objects.

Instances of the class are named under topoViewDB, but create/delete operations are not supported by the name binding. All topoView objects are created/deleted as side effects of creating/deleting topoNode objects and changing the topoNodeParents attribute of a topoNode. The MIS is responsible for maintaining a one-to-one relationship between a topoView object and a container type topoNode object (for example, a topoNode can contain other topoNodes).

D.5.2 topoViewNode

The topoViewNode objects represent topoNode objects in different views. Each topoViewNode object is associated with a topoNode object. There is a many-to-one relationship between topoViewNode and topoNode objects. If the information is available in the topoViewNode's associated topoNode object, the information is not duplicated in the topoViewNode object. The exception is that the topoNodeId attribute is used as the naming attribute to create topoViewNode objects.

Since instances of the class are named under topoView, create/delete operations are not supported by the name binding. All topoViewNode objects are created/deleted as side effects of creating/deleting topoNodes and adding/removing parents to/from the topoNodeParents attribute of topoNode objects. The MIS is responsible for maintaining the referential integrity between topoViewNode and topoNode objects.

D.6 Topology Types

The following descriptions of topology types are general descriptions.

Table D-1 Topology Types

Topology Type	**Description**
Container	A generic view representation.
Universe	A generic view, generally used at the top level.
Internet	Any combination of IP networks.
Subnetwork	Containers specific to the Internet.
Host	An IP device on a network.

Table D-1 Topology Types

Topology Type	**Description**
Device	A generic representation of a network element.
Link	A physical connection between two network elements.
Router, Bridge, Hub	Multiple interface devices capable of transferring packets between networks.

Caution – Users are not allowed to create their own base topoTypes.

Index