Solstice™ CMIP 8.2 Programmer's Guide

SunSoft
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.

Part No.: 802-5281-10
Revision A, April 1996

Contents

Preface xv

1. Introduction to OSI Systems Management 1

Overview 1

Managed Systems 2

Managed Objects 3

Attributes 4

Operations 4

Behavior 5

Notifications 5

Packages and Conditional Packages 5

Classes and Inheritance 5

ISO Registration Tree 6

Object Containment Hierarchy 9

Object Naming 9

Management Information Tree (MIT) 10

Scoping 11
Filtering 12
Linked Replies 13
Chapter Summary 13
2. Software Architecture Overview 15
Overview of the Software Architecture 15
XMP Services 18
Management Communication Services (MCS) 18
User Context 19
Association Context 19
MCS Communication to the User 19
Common Management Information Service (CMIS) 20
Chapter Summary 20
3. Object Management (OM) 21
Overview 21
C Naming Conventions 23
Package 25
Object Attributes 26
Class 26
XOM Representation of ASN.1 27
Public and Private Objects 28
Public Objects 28
Client-Generated Public Object 28
Service-Generated Public Object 30

Private Objects 31
Import/Export of Object Identifiers 34
XOM Function Interface 35
Storage Management 36
Workspaces 37
Chapter Summary 37
4. Systems Management Protocol 39
Overview 39
The Manager and Agent 40
C Naming Convention 42
CMIS Services 44
Function Calls 45
Function Sequencing 49
Implementation Specific Enhancements 50
`mp_negotiate()` Function 50
`mp_wait()` Function 51
`proprietary-Args` Attribute of Session Object 51
AE Titles 52
Chapter Summary 52
5. XMP Development Concepts 55
Overview 55
Initial Declaration 55
Connection Management 56
Responder Versatility 56

Loopback Facility 56
Synchronous and Asynchronous Operations 58
Synchronous 58
Asynchronous 58
Access Control 59
Error Codes 59
Unbind / Shutdown Errors 59
Asynchronous Mode Errors 60
Session Objects 60
Default Session Object Attributes 61
Context Objects 62
Default Context Object Attributes 63
Restrictions 64
Managing Multiple Event Types 64
Packages 65
Common Management Service Package 65
CMIS Management Service Package 65
Chapter Summary 65
6. Addressing 67
Remote Addressing 67
Specialized Session 67
Specialized Context 68
Local Addressing 68
Chapter Summary 70

7. Compiling and Linking Application Programs 71
Library Contents 71
Include File Structure 72
Compile and Link Procedure 72
Example `Makefile` 73
Running the Example Programs 74
Example 1 74
Compiling and Linking 75
Running Example 1 75
Example 2 79
Compiling and Linking 80
Running Example 2 80
Example 3 82
Compiling and Linking 82
Running Example 3 83
A. Enhancements to Draft 7 Preliminary Specification 85
Error Handling 85
Draft 7 Preliminary Specification 86
CAE Specification 86
OM Class Definitions 87
Automated Connection Management 88
Draft 7 Preliminary Specification 88
CAE Specification 88
Session Argument with ACM Disabled 89

Draft 7 Preliminary Specification 89
CAE Specification 89
Synchronous Operation With ACM Disabled. 89
Asynchronous Operation With ACM Disabled 90
B. Compliance Information and Product Limitations 91
Compliance Information 91
Product Limitations 92
General 92
Security 92
Session Object Attributes 93
Maximum Number of Sessions 93
Context Object Attributes 93
Interface Objects. 93
Using Loopback Mode 94
`mp_validate_object()` Function 94
`mp_assoc_rsp()` Function 95
`mp_release_rsp()` Function 95
`mp_get_assoc_info()` Function 95
Glossary 97

Figures

Figure 1-1 Hierarchical Organization of Managers and Agents 2

Figure 1-2 Managed Systems 3

Figure 1-3 ISO Registration Tree 8

Figure 1-4 Containment Hierarchy and Object Naming 10

Figure 1-5 Management Information Tree (MIT) 11

Figure 1-6 Scoping Algorithms. 12

Figure 2-1 Global Architecture Overview. 17

Figure 3-1 Conceptual Model of Object Management (XOM). 22

Figure 4-1 Manager / Agent Interaction. 41

Figure 4-2 CMIS Interaction 42

Figure 4-3 XMP Sequencing State Diagram 48

Figure 5-1 Loopback through `osimcsd` 57

Figure 5-2 Loopback through Transport Layer 57

Figure 6-1 Addressing Scheme. 69

Tables

Table P-1	Typographic Conventions	xvii
Table P-2	Shell Prompts	xviii
Table 3-1	Derivation of C Identifiers	24
Table 3-2	Attributes of `CMIS-Get-Result` Object	26
Table 3-3	Attributes of `CMIS-Get-Result` Object	27
Table 4-1	C Naming Conventions	42
Table 4-2	CMIS Services	44
Table 4-3	XMP Functions	46
Table 5-1	Session Object Attributes	61
Table 5-2	Context Object Attributes	63

Code Samples

Code Example 3-1 `address.h` Segment 29

Code Example 3-2 `extract.c` Segment 30

Code Example 3-3 `objtool.c` Segment 31

Code Example 3-4 `put_desc()` Function Segment. 32

Code Example 3-5 `ENDOBJ` Macro Definition 33

Code Example 3-6 `imports.h` Segment 34

Code Example 3-7 `exports.c` Segment 34

Code Example 3-8 `imports.h` Segment 35

Preface

This manual provides an introduction to the object management and management protocols defined in the X/Open™ document set. Properties of the Common Management Information Service (CMIS) interface are covered. Code segments are provided to facilitate implementation of management applications on top of the standard XMP interface. Information on the compile and linking procedure is also included.

This manual is part of the document set for Solstice™ CMIP SDE. The other documents contained in this set are:

- *Solstice CMIP 8.2 Administrator's Guide*
- *Solstice XOM Programming Reference*
- *Solstice XMP Programming Reference*

Who Should Use This Book

This document is written for programmers with a working knowledge of CMIS principles and concepts.

Reference documentation from X/Open Company Ltd. is included with this release and is referenced frequently throughout this document.

How This Book Is Organized

The *Solstice CMIP Programmer's Guide* is organized as follows:

Chapter 1, "Introduction to OSI Systems Management," introduces the object-oriented terminology used throughout this manual and describes the concepts on which the Solstice CMIP is based.

Chapter 2, "Software Architecture Overview," covers the software architecture of the XMP/XOM structure.

Chapter 3, "Object Management (OM)," covers the object XOM management application programming interface implementation.

Chapter 4, "Systems Management Protocol," explains the facilities used to develop applications using the XMP development environment.

Chapter 5, "XMP Development Concepts," covers the procedures for building XMP applications.

Chapter 6, "Addressing," describes the local and addressing structuring.

Chapter 7, "Compiling and Linking Application Programs," discusses the library structure and `Makefile` process.

Appendix A, "Enhancements to Draft 7 Preliminary Specification," describes the primary enhancements to the XMP interface introduced since Draft 7 of the preliminary specification and describes how to modify applications that conform to Draft 7 so that they can be compiled and linked using Solstice CMIP SDE.

Appendix B, "Compliance Information and Product Limitations," describes the related product compliance information.

"Glossary," contains all of the terms used in object-oriented programming and the XOM/XMP interface API environment.

Related Books

The following reference documents are helpful in understanding OSI principles:

- *Computer Networks (Second Edition)* by Andrew S. Tanenbaum (Prentice-Hall International Editions, 1988)
- *OSI A Model for Computer Communications Standards* by Uyless Black (Prentice-Hall, 1991)
- *Network Management Standards (The OSI, SNMP and CMOL Protocols)* by Uyless Black (McGraw-Hill on Computer Communications, 1992)
- *SNMP, SNMPv2, and CMIP: The Practical Guide to Network-Management Standards* by William Stallings (Addison-Wesley, 1993)

What Typographic Changes Mean

The following table describes the typographic changes 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. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name% 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

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	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#

Introduction to OSI Systems Management 1

This chapter provides an introduction to the Open Systems Interconnection (OSI) systems management model. It introduces the object-oriented terminology used throughout this manual and describes the concepts on which Solstice CMIP is based.

Overview	*page 1*
Managed Systems	*page 2*
Managed Objects	*page 3*
Classes and Inheritance	*page 5*
Object Containment Hierarchy	*page 9*
Chapter Summary	*page 13*

Overview

OSI systems management refers to a set of standards that defines object-oriented network management in the OSI domain. It includes specifications for a management service (CMIS), a management protocol (CMIP), a hierarchical information database (MIT), and the data objects it contains.

OSI systems management is based on the hierarchical exchange of management information between two entities—a *manager* and an *agent*. The *manager* issues management directives and receives status reports; the *agent*

responds to directives and issues status reports. The manager of a lower-level system may simultaneously act as the agent of a higher-level system as shown in Figure 1-1.

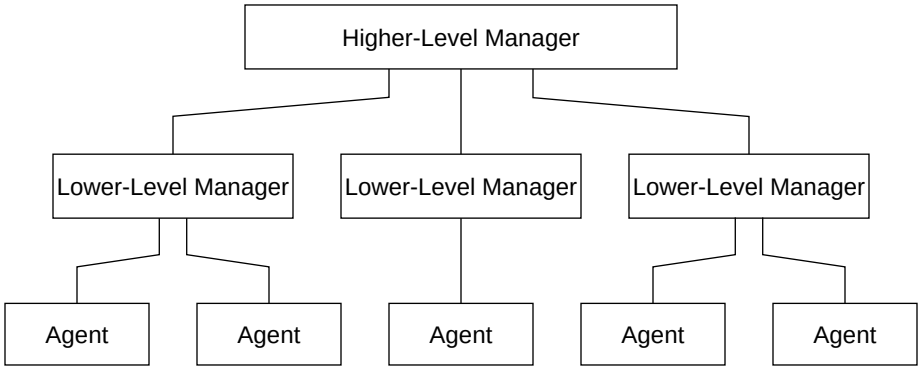

Figure 1-1 Hierarchical Organization of Managers and Agents

Managed Systems

A *managed system* consists of one or more agent processes controlled by a managing system as shown in Figure 1-2 on page 3. Each agent in a managed system is responsible for carrying out management directives to control or return information from system resources. There may be several managed systems in a network.

Communication between managers and a managed system uses CMIP; communication between agents and managed resources is based on a device-dependent protocol. The implementation of a device-dependent protocol is called an *access method*.

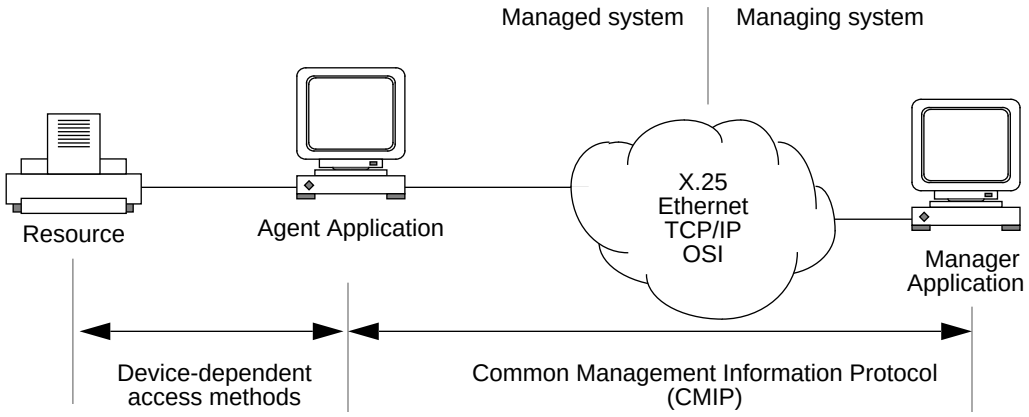

Figure 1-2 Managed Systems

Managed Objects

A *managed object* is a software abstraction of a resource that can be managed across an OSI network. Managed objects can be defined for hardware (such as workstations, servers, printers, switches, multiplexers, and private branch exchanges) or software (such as queuing programs, routing algorithms, and buffer management routines).

A managed object is an *encapsulation* of the data associated with a resource, combined with device-dependent software procedures (called *object methods*), which implement the operations that can be performed on the data. The modular form of a managed object allows new objects to be defined and old objects to be deleted as required.

A managed object is fully defined by:

- Its attributes
- The management operations that can be performed on it
- Its behavior in response to a requested management operation
- The notifications it generates

- The conditional packages it may include
- Its position in the information hierarchy

Attributes

The data elements that are encapsulated in a managed object are called *attributes*. Each attribute corresponds to one of the characteristics of the system resource that the managed object represents. An attribute has a *name*, a *type*, and one or more *values* that reflect the current status of the associated resource.

For example, the managed object defined for a packet switch could have an attribute that describes the operational state of the switch. The attribute named `operational state` could be of type `integer`, and the valid values could be *disabled* (`0`), *enabled* (`1`), *inactive* (`2`), or *busy* (`3`).

Attributes can be read to recover information about the associated resource or modified to alter the current state of the associated resource.

Operations

The following management operations are defined by CMIS service primitives:

- `CREATE` – Creates a new managed object
- `DELETE` – Deletes a managed object
- `GET` – Obtains attribute values from the managed object
- `SET` – Modifies attribute values for a managed object
- `ACTION` – Performs an operation on a managed object
- `EVENT-REPORT` – Sends an event report
- `CANCEL-GET` – Cancels outstanding `GET` request

The operations that can be performed on a managed object must form part of its definition, which may also include the effect that these operations have on related system resources. Note that the current state of a managed object may determine the type of operations that can be performed on it at a given time.

The `DELETE`, `GET`, `SET`, and `ACTION` operations can be performed on multiple objects.

Behavior

A managed object reacts to both internal and external events. Internal events are associated with the managed resource or the object itself—for example, a watchdog timer timing out. External events occur in response to a request issued by a managing process in the form of CMIP messages.

The behavior of a managed object describes its response to these events and the constraints placed upon it. This response is determined by the object methods associated with the object, the current state of the object, the dependencies between values of particular attributes, and the classification of the object within the managed system.

Notifications

A managed object issues status reports, called *notifications*, in response to internal and external events. Notifications may be transmitted to a managing process in the form of CMIP messages or logged internally. The type of notifications issued by a managed object and the conditions under which notifications are issued form part of its definition.

Packages and Conditional Packages

A *package* is an indivisible set of specifications—for example, attributes, actions and notifications. A conditional package is a set of specifications that are optionally all present or all absent in a managed object. The conditions under which a package is present are dependent on the characteristics of the resource being managed.

Classes and Inheritance

Managed objects that exhibit similar characteristics are grouped into *object classes*. Each managed object is therefore a specific *instance* of a class, and its properties are consistent for all other object instances of that class. However, the attributes of different class instances may possess different values.

The concept of object classes is similar to that of a template. An object class is defined once and reused thereafter for all objects of the same class. In addition, new classes can be defined in terms of existing classes. The new class is referred to as a *subclass* of the class from which it is derived and may in turn have subclasses of its own.

A subclass *inherits* some of its characteristics from the class from which it is derived (also called its *superclass*). However, its characteristics can be extended in one or more of the following ways:

- Adding new attributes or modifying the properties of existing attributes
- Adding new operations or modifying the arguments or restrictions associated with existing operations
- Adding new notifications or modifying the arguments or restrictions associated with existing notifications

The OSI systems management model does not allow a subclass to be derived by the deletion of any of the characteristics of the superclass.

The ultimate superclass is the *top object class*, from which all other object classes are derived. This object is specified by the OSI systems management model in ITU-T X.720/ISO-10165-1 *Management Information Model* and contains definitions for the attributes that are common to all object classes.

For example, the *object class* attribute, which is contained in every object instance, is defined in the top object class.

ISO Registration Tree

Object classes are identified by registered object identifiers. The ISO registration tree is a hierarchy of officially recognized object identifiers for object classes, attribute definitions, actions, notifications, and packages. Each object identifier is a sequence of integers that points to its location in the tree.

The part of the ISO registration tree which relates to OSI systems management is shown in Figure 1-3 on page 8. The key nodes are:

- `joint-iso-ccitt`
  Allocated to all identifier values specified by joint agreement between ISO and ITU (formerly CCITT).
- `ms` (management specification)
  Allocated to all identifier values specified by the OSI systems management standards.
- `smo` (systems management overview)
  Allocated to all identifiers specified by ITU-T X.701 / ISO-10040 *System Management Overview*.
- `cmip` (common management information protocol)
  Allocated to all identifiers specified by ITU-T X.711 / ISO-9596-1 *Common Management Information Protocol Specification*.
- `smf` (system management functions)
  Allocated to all identifiers specified by ITU-T X.730 / ISO-10164-1 *Object Management Function*.
- `smi` (structure of management information)
  Allocated to all identifiers specified by ITU-T X.720 / ISO-10165-1 *Management Information Model*.

The object identifier (OID) for the CMIP (version1) protocol is therefore:
`2.9.1.1.3`

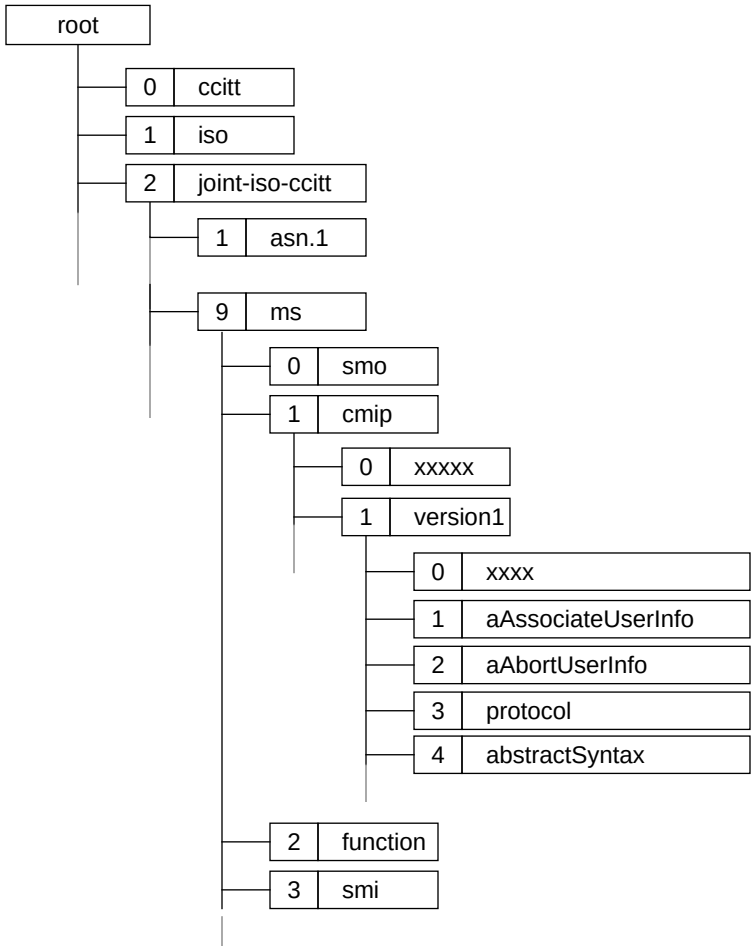

Figure 1-3 ISO Registration Tree

Object Containment Hierarchy

In the OSI systems management model an object can include (or *contain*) another. A containing object may, in turn, be contained in another object. A superior object can contain more than one object, but a contained object can only be contained in one superior object at a time. This last restriction forces a tree structure on the containment hierarchy.

Object Naming

The naming scheme for managed objects is dependent on the containment hierarchy. Each managed object class includes a *naming* attribute. The naming attribute is chosen to ensure its value is unique for each managed object instance amongst objects that are subordinate to the same superior. The naming attribute and its value provide the *relative distinguished name* (RDN) of an object instance. The RDN is expressed in an *attribute value assertion* (AVA) as *namingAttribute* = "*value*". For example, if the naming attribute of an object class is `RouterName` and its value for an instance is `A`, the RDN of the instance is `RouterName="A"`.

The *name binding* is the rule that provides the *distinguished name* (DN) of an object instance. The DN of an object instance represents its unique location in the containment hierarchy. It is a concatenation of the sequence of RDNs from the root of the containment hierarchy to the object. For example, consider a port contained in a card, which is in turn contained in a router. The RDNs of the objects are:

- `RouterName="RouterA"` (for the router)
- `CardName="Card0"` (for the card)
- `PortId="Port2"` (for the port)

The DN of the port is `RouterName="RouterA",CardName="Card0",PortId="Port2"`. This is shown in Figure 1-4 on page 10.

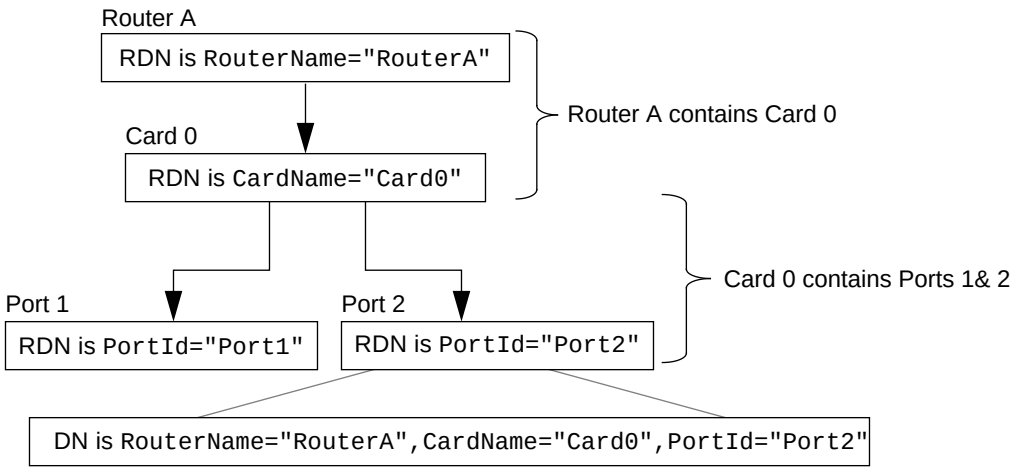

Figure 1-4 Containment Hierarchy and Object Naming

Management Information Tree (MIT)

The management information tree (MIT) is a computational representation of the object containment hierarchy. Such a representation allows an agent application to locate or create data associated with an object identified by its DN. The *system object* contains definitions for the attributes that are common to all objects in the managed system. Each node represents an object entry that is contained in the superior object, and each object entry contains the data associated with a managed resource in the form of attributes, as shown in Figure 1-5 on page 11.

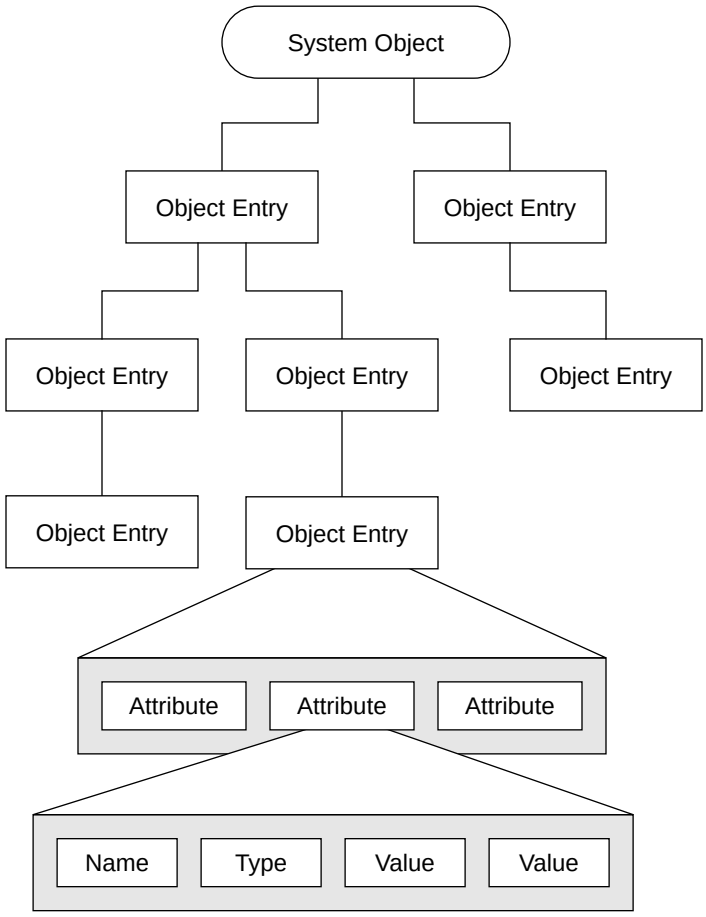

Figure 1-5 Management Information Tree (MIT)

Scoping

The term *scoping* is used to describe the way in which one or more objects in the MIT are selected to be the subject of a management operation. Scoping defines a subtree within the containment hierarchy.

Scoping is defined with reference to the *base managed object*, which is the root object for the scoped subtree. The scope of the subtree can include any one of the following:

- The base managed object only
- The *n*th-level subordinates of the base managed object
- The base managed object and all its subordinates to the *n*th-level
- The base managed object and all its subordinates

These four scoping algorithms are illustrated in Figure 1-6.

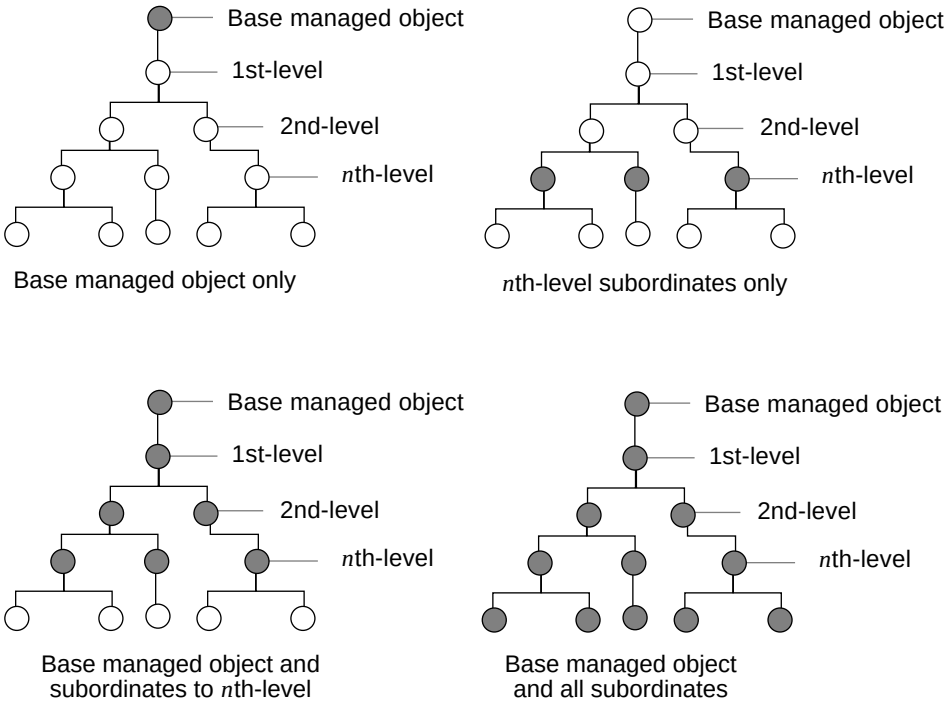

Figure 1-6 Scoping Algorithms

Filtering

Filtering is used to select or reject scoped objects based on the presence, values, or order of specific attributes. The filter is a Boolean expression, which may be a single test or a combination of multiple tests.

Filters are specified in the request received from the managing process and are applied as follows:

1. Scoping is used to select the objects to which the filter is to be applied.
2. The filter is applied to the attributes of each scoped object.
3. A subset of scoped objects is identified.

Linked Replies

When a management operation is applied to multiple objects, one response is returned for each object selected in the request received from the managing process. These responses are called *linked replies* because they refer to the same request.

Chapter Summary

OSI systems management is based on the hierarchical exchange of management information between managers and an agents. Communication between managers and agents uses the common management information protocol (CMIP).

A managed object is a software abstraction of a resource that can be managed across an OSI network. It is an encapsulation of the data associated with a resource, combined with device-dependent software procedures.

Managed objects that exhibit similar characteristics are grouped into object classes. Each managed object is therefore a specific instance of a class, and its properties are consistent for all other object instances of that class.

In the OSI systems management model, managed objects are organized in a containment hierarchy, which has a tree structure. The naming scheme for managed objects is dependent on the containment hierarchy. The distinguished name (DN) of an object instance represents its unique location in the containment hierarchy.

The management information tree (MIT) is a computational representation of the object containment hierarchy. Such a representation allows an agent application to locate or create data associated with an object identified by its DN.

1

Software Architecture Overview 2

This chapter describes the software architecture for the management protocol. Flow diagrams are used to facilitate an understanding of the unique terminology that is part of this development environment.

Overview of the Software Architecture	*page 15*
XMP Services	*page 18*
Management Communication Services (MCS)	*page 18*
Common Management Information Service (CMIS)	*page 20*
Chapter Summary	*page 20*

Overview of the Software Architecture

The software contains the following modules:

- An XMP/XOM library that contains the implementation of the X/Open XMP API.
- The Management Communication Service (MCS) is the entity that provides connectionless access to the CMIS service.
- The Common Management Information Service (CMIS) is the entity that provides the services and protocols specified in ISO-IS 9595/9596.
- The transport provider, which provides access to the *peer-to-peer* communication.

2

As a performance enhancement, the MCS and CMIS entities are merged into a single process called "`osimcsd`". This process resides in user space and provides the communication mechanism between the XMP library and the communication platform. This process is transparent to the XMP developer. The communication mechanism (XMP library to `osimcsd`) is based on InterProcess Communication (IPC). IPC messaging allows processes to send and receive messages, and to queue messages for processing in an arbitrary order. Unlike the file byte-stream model of data flow used for pipes, each IPC message has an explicit length.

Figure 2-1 shows a diagram of three components: user process, `osimcsd` process, and the transport provider. There is a clear division between the user process (XMP library) and the Communication Management Information Service (CMIS) provider.

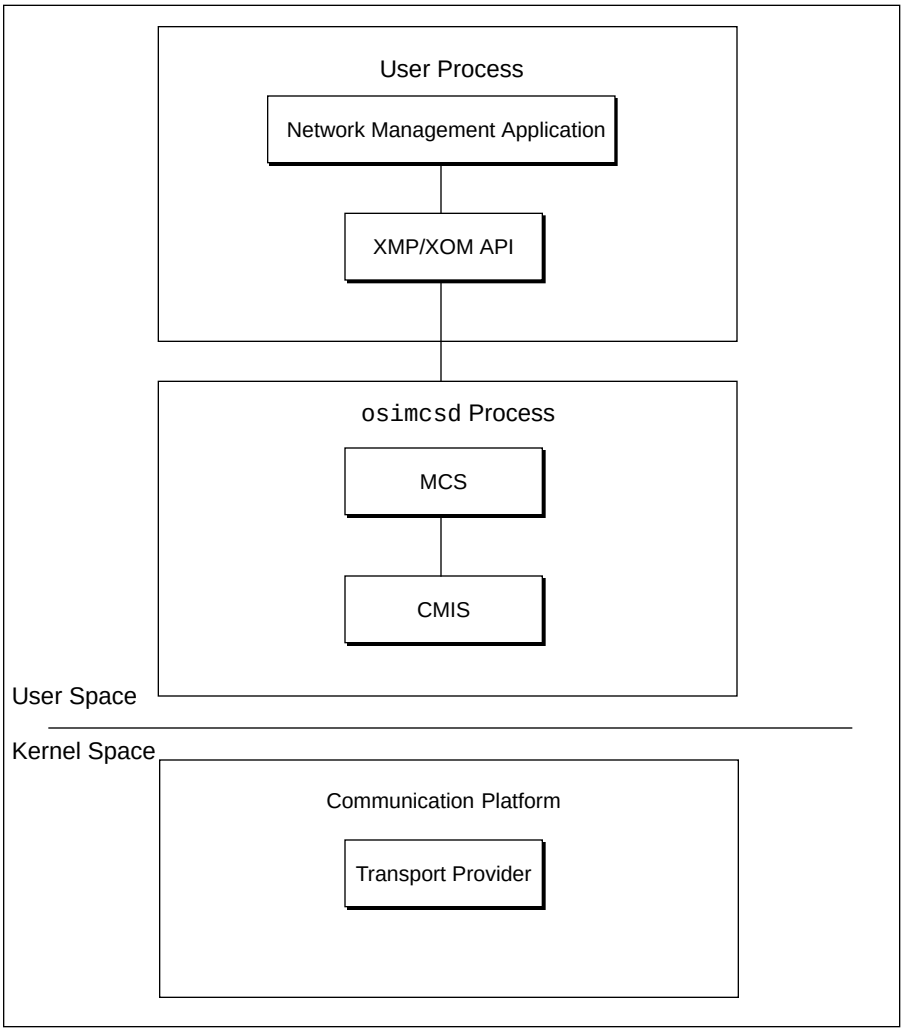

Figure 2-1 Global Architecture Overview

2

XMP Services

The XMP API is designed to provide access to both CMIP and the Simple Network Management Protocol (SNMP).

Note – This release of the management protocol development environment does not provide support for SNMP.

XMP is implemented on top of the MCS entity, as shown in Figure 2-1. Additional programming information on XMP is provided in the following chapters.

Note, you can control association allocation through the XMP interface or allow the MCS to handle associations automatically.

Management Communication Services (MCS)

The MCS offers network management applications an easy way to use the CMIS. As shown in Figure 2-1 on page 17, the MCS module is part of the `osimcsd` process.

The primary function of the MCS is to provide connectionless CMIS service by handling association management. When you use the MCS services, an association does not have to be established prior to issuing an operation. Prior to sending the operation, the MCS will handle the management of an association. If an association is presently open to a remote application, the MCS will use it to issue the requested operation. You can choose to bypass the MCS's association management control and use the XMP interface instead of opening and closing associations.

If no association is open, the MCS will open the association before issuing the requested operation. Once the operation is completed, the MCS will release the association after a specified *inactivity* time. This inactivity time can be configured, see *Solstice CMIP 8.2 Administrator's Guide.*

The MCS entity handles the management associations. This implies that responding to incoming requests does not require the OSI address of the requestor. The invoke identifier associated with the operation allows the XMP library and the `osimcsd` process to correlate indications and responses.

Part of the MCS communication mechanism is *User Context, Association Context,* and *MCS Communication to the User.* These three items are explained in the following subsections.

User Context

The *user context* contains information that is related to a user's registration with the MCS: user capabilities, user role, default address, and default time-out values.

A user context is freed whenever any of the following occurs:

- Registration is cancelled.
- Errors are encountered.
- Administrative commands are received.

Association Context

An *association context* contains all information related to a CMISE association such as, function negotiation units, application context, and remote address. During the initialization process, the association context is established. An association context cannot be shared by several MCS users.

The termination of an association and the release of the association context occurs when the association is:

- Refused
- Abnormally released (an Abort Indication is received from CMISE)
- Released by the MCS, the remote CMISE user or both

MCS Communication to the User

When the MCS wants to communicate with an user, it checks the user status to determine if it is ready. The message is sent immediately to the user, or it is queued for later delivery.

A communication message contains:

- User context data
- Flow control status
- Interaction acceptance
- User busy status

2

Common Management Information Service (CMIS)

CMIS provides services for performing management operations. The services provided by the implementation are compliant with the ISO-IS 9595/9596 version 2 standards. The services are divided into these categories:

- Management operation services
- Management notification services

To communicate, any entity that uses CMIS services also needs to use association services for establishing an application association.

CMIS also provides structuring facilities for:

- Multiple responses to confirmed operations that are linked through a identification parameter
- Management of multiple objects through a selected criteria of scoping and filtering

Chapter Summary

Solstice CMIP provides XMP/XOM library access to the MCS and CMIS. The integration with X/Open's libraries and the application programming interface allow service communication to be merged with the MCS/CMIP (`osimcsd` process). The `osimcsd` communication mechanism uses InterProcess Communication, where messages can be sent, received, and queued. The primary function of the MCS is to provide connectionless CMIS service by handling association management. Solstice CMIP SDE provides the developer a choice of automatic MCS control of associations, or programmatic manipulation through the XMP interface.

Object Management (OM) 3

This chapter provides an overview of Object Management (OM), the implementation of which is described in detail in *Solstice XOM Programming Reference*.

Overview	*page 21*
XOM Representation of ASN.1	*page 27*
Public and Private Objects	*page 28*
Import/Export of Object Identifiers	*page 34*
XOM Function Interface	*page 35*
Storage Management	*page 36*
Workspaces	*page 37*
Chapter Summary	*page 37*

Overview

The OM can create, modify, and delete complex information objects. This environment provides developers with a uniform architecture model of information based upon groups and classes.

Some of the terms used in this document, such as object and attribute, are used in a different way when referring to parts of the management information. Care has been taken to avoid confusion by using distinct names for each such term. Note, there is a distinction between OM classes and managed object classes and between OM attributes and managed object attributes. The OM

class and OM attribute construct the interface, while managed object class and managed object attribute represent the managed information. The usage of the term *attribute* denotes a managed object attribute, while the phrase *OM attribute* denotes the OM construct.

The XOM API is designed to be used by one or more independent developers. This API can be used with these services: X.400 / MHS, X.500 / Directory Service, and X.700 / CMIP. XAPIA provides access to X.400 service. XDS provides access to X.500 service. XMP provides access to X.700 service. As illustrated in Figure 3-1, each developer provides her own code to manipulate information objects.

X.400 Message Handling System allows clients to manipulate message queues, to send messages, and handle query search. *X.500 Directory Service* provides a naming service. It manages names, associated attributes, and provides a hierarchical architecture for naming. *X.700 Management Framework* outlines the OSI network management model. The Common Management Information Protocol (CMIP) is specified in recommendation X.711 / ISO 9596. The Common Management Information Service (CMIS) is specified in recommendation X.710 / ISO 9595.

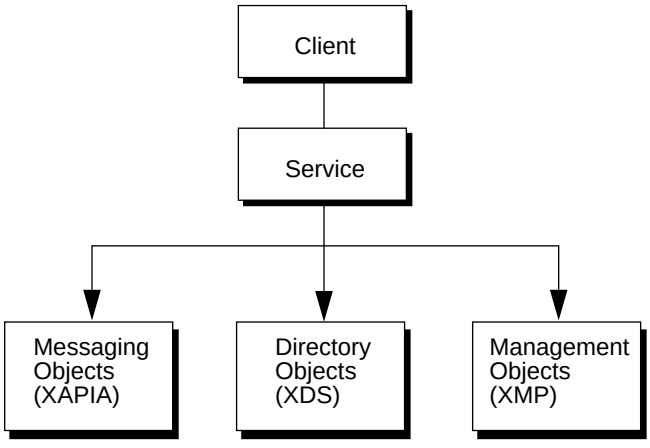

Figure 3-1 Conceptual Model of Object Management (XOM)

Throughout this chapter, the term *interface* denotes the OM API. The term *service* denotes software that implements the interface, and *client* denotes software that uses the interface. The *service interface* denotes the interface realized by the service as a whole, and is used as a synonym for *interface*.

Note – The OM uses a structured view of information; however, it does not incorporate all characteristics of other object-oriented programming environments. For instance, the implementation functions that manipulate objects are separate from the definitions of the object's classes. Also, there is no notion of encapsulation or hiding the information associated with objects, although the interface hides the information representation.

Objects communicate between the client and service by using a sequence of descriptors (C structures). Unlike the objects themselves, the representation of such sequences is part of the OM interface specification.

All the C identifiers are mechanically derived from a generic, language-independent interface as outlined below.

The binding specifies C identifiers for all elements of the interface so that application programs written in C can access the MIB. These interface elements include *function names*, *typedef names*, and *constants*.

All C language names are shown in *italic* typeface. A function is indicated by following parentheses *function()*, and a constant is surrounded by braces *{CONSTANT}*. The names of errors are surrounded by brackets *[ERRORS]*.

The definitions of the C identifiers appear in the `<xom.h>` header file, which contains definitions for the associated OM interface.

C Naming Conventions

The identifier for an element of the C interface is derived from the name of the corresponding element of the generic interface. This depends on the element type, as specified in Table 3-1 on page 24. The generic name is prefixed with the character string in the second column of the table, alphabetic characters are converted to the case in the third column, and an underscore (_) is substituted for each hyphen (-) or space ( ).

The prefixes “omP” and “OMP” are reserved for developers. The prefixes “omX” and “OMX” are reserved for the proprietary extension of the interface. In all other respects, such extension is outside the scope of this document.

Note – Hyphens are translated everywhere to underscores. X/Open Management Protocol functions pass most arguments by reference. The data referenced by these arguments are modelled and manipulated in an object-oriented fashion.

Table 3-1 Derivation of C Identifiers

Element Type	Prefix	Case	Example Usage
Data type	OM_	Lower	OM_sint32
Data value	OM_	Upper	OM_TRUE
Data value (Class)	OM_C_	Upper	OM_C_ENCODING
Data value (Syntax)	OM_S_	Upper	OM_S_SYNTAX
Data value component (Structure member)	*none*	Lower	anything from a structure
Function	om_	Lower	om_put ()
Function argument	*none*	Lower	subject, type, etc.,
Function result	*none*	Lower	success, network-error, etc.,
Macro	OM_	Upper	OM_OID_DESC
Reserved for use by implementors	OMP	any	reserved for implementor
Reserved for use by implementors	omP	any	reserved for implementor
Reserved for proprietary extension	omX	any	
Reserved for proprietary extension	OMX	any	

Package

Related classes are grouped into collections called packages. A package defines the set of OM classes that are grouped together by the specification because they are functionally related. A package is denoted by an ASN.1 object identifier. This number uniquely identifies an object identifier. For example, XMP specifies that a CMIS package be identified by the object identifier:

```
\x2a\x86\x3a\x00\x88\x1a\x01\x02
```

See *SunLink ASN.1 Compiler User's Guide* for additional information.

The closure of a package P is the set of classes that need to be supported in order to be able to create all possible instances of all classes defined in P.

Package closure is formally defined in terms of class closure, which is the set of classes that need to be supported in order to be able to create all possible instances of a particular class.

More specifically, the closure of a class C, *where C is used as a name qualifier,* is a set that consists of:

- The class C itself
- The closures of any subclasses of C defined in the same package as C
- The closures of the classes of all permitted subobjects of instances of C

The closure of a package P is the set of classes made up of the union of the closures of all the classes defined in P.

For purposes of the generic interface, the definition of a package has the following elements:

- The package's name, which denotes the package's object identifier
- The definitions of the one or more classes which make up the package
- The identification of zero or more concrete (an instance is permitted) classes in the package to which the *Create* function applies (in every implementation of the service)
- The identification of zero or more concrete classes in the package to which the *Encode* function applies (in every implementation of the service)
- The explicit identification of the zero or more classes in other packages that appear in the package's closure (as a convenience to the reader)

Object Attributes

An object can have one or more attributes. An attribute definition consists of the name of the attribute, a syntax, and the value of the specified syntax. For example, the XOM definition of `CMIS-Get-Result` in Table 3-2 indicates that this object can contain the XOM attributes: *managed object class, managed object instance, attribute list* and *current time*. This definition indicates that the syntax of the *managed object class* attribute is an XOM object and the syntax of *current time* is generalized time. The *Value Number* specifies the number of times an attribute can appear within an object. If the attribute can appear zero times, then it is an optional attribute.

Table 3-2 Attributes of `CMIS-Get-Result` Object

OM Attribute	Value Syntax	Value Number
managed object class	Object (Object Class)	0 - 1
managed object instance	Object (Object Instance)	0 - 1
current time	String (Generalized Time)	0 - 1
attribute list	Object (Attribute)	0 - more

A definition of an XOM attribute contains the following:

- Name of the attribute (*OM Attribute*)
- Syntax of the attribute (*Value Syntax*)
- Constraint on the length of a value syntax string (*Value Length*)
- Constraint on the number of values (*Value Number*)
- Default value that is used to initialize the attribute (*Initial Value*)

Class

Each object is an instance of its *class*. A class is characterized by the types of the attributes that appear as its instances. A class is denoted by an ASN.1 object identifier. The object identifier that denotes a class is an attribute of every instance of the class. As an aid to the discussion of classes, *C1 and C2* are used as class qualifiers. The types that may appear in an instance of one class, C1, are often a superset of those that may appear in an instance of another class, C2. When this is so, C2 may (but need not) be designated a *subclass* of C1, making C1 a *superclass* of C2. If C1 is a superclass of no other superclass of C2, C1 is called the *immediate superclass* of C2, and C2 an *immediate subclass* of C1.

There are two kinds of classes: *concrete* and *abstract*. Instances of a concrete class are permitted, but instances of an abstract class are forbidden. An abstract class may be defined as a superclass in order to share attributes between classes, or simply to ensure that the class hierarchy is convenient for the interface definition. An XOM class definition consist of:

- Name of the class
- Identification of the superclass (XOM has an inheritance scheme)
- Definitions of the attributes specific to the class
- Whether the class is abstract or concrete

XOM Representation of ASN.1

XOM provides a structural solution for representing complex ASN.1 syntax. For example, in the X.700 CMIP Standards documentation, the `GetResult` ASN.1 syntax is defined as follows:

```
GetResult ::= SEQUENCE {
    managedObjectClass    ObjectClass    OPTIONAL,
    managedObjectInstance ObjectInstance OPTIONAL,
    currentTime           [5]IMPLICIT GeneralizedTime OPTIONAL,
    attributeList         [6]IMPLICIT SET OF attribute OPTIONAL }
```

The XOM object corresponding to this ASN.1 representation is shown in Table 3-3.

Table 3-3 Attributes of `CMIS-Get-Result` Object

OM Attribute	Value Syntax	Value Number
managed object class	Object (Object Class)	0 - 1
managed object instance	Object (Object Instance)	0 - 1
current time	String (Generalized Time)	0 - 1
attribute list	Object (Attribute)	0 - more

Note – All of the code segments given in the following sections were taken from the complete programs located in `/opt/SUNWconn/cmip/examples`.

3

Public and Private Objects

The system management data abstract services consist of many data structures designed to manipulate programming constructs. Public objects are programmer-visible data structures that enable objects to be statically defined. Private objects are private to the service and can only be accessed from programs indirectly using interface functions.

Public Objects

A public object can be generated by a *client* or by a *service*.

- *Client-generated public objects*-correspond to XOM objects that are created by the developer through normal language constructs. The developer is responsible for managing any storage involved in the representation of the object.
- *Service-generated public objects*-correspond to public objects that are generated by the service (XOM/XMP). They are generated from specific calls to the services. The management of the storage is handled by the service. The `om_delete()` function must be used on service-generated public objects to maintain storage consistency.

Client-Generated Public Object

Code Example 3-1 shows a static definition of a client-generated public object. A public object is an array of descriptors. The first descriptor (`OM_descriptor`) structure must provide the class of the object. The class can be defined with the standard XOM macro `OM_OID_DESC`, which is defined in `xom.h`.

The end of the array of the OM descriptors must be indicated with the null descriptor. `OM_NULL_DESCRIPTOR` is the standard macro for the null descriptor.

Keep in mind that the storage management must be done by the application. Note that a client public object is not deleted when the workspace containing it is deleted.

Code Example 3-1 `address.h` Segment

```
/* Declaration of the agent's address */

OM_descriptor agentAddr[]= {

/*  The "OM_OID_DESC" macro is defined by XOM. It allows the
 *  initialization of "OM_CLASS" attribute for public objects.
 *  As a reminder, OM_CLASS attribute of a public object is used
 *  to indicate the public object class.  The "agentAddr"
 *  structure corresponds to a public XOM object whose class
 *  is MP_C_PRESENTATION_ADDRESS.
 */

    OM_OID_DESC(OM_CLASS, MP_C_PRESENTATION_ADDRESS),

/*  An XOM object whose class is MP_C_PRESENTATION_ADDESS must
 *  contain a network address (MP_N_ADDRESSES) attribute. By
 *  default, the example is using RFC1006; therefore, the
 *  network address is not used. The syntax of the network
 *  address (MP_N_ADDRESSES) attribute is defined as an octet
 *  string. Note, the OM_STRING is a macro defined by XOM for
 *  filling a "OM_String" structure.
 */

    {MP_N_ADDRESSES, OM_S_OCTET_STRING, {OM_STRING("")}},

/*  The presentation selector (MP_P_SELECTOR) attribute is set
 *  to "rfc0"
 */

    {MP_P_SELECTOR, OM_S_OCTET_STRING, {OM_STRING("rfc0")}},
    {MP_S_SELECTOR, OM_S_OCTET_STRING, {OM_STRING("Prs")}},
    {MP_T_SELECTOR, OM_S_OCTET_STRING, {OM_STRING(CMIP)}},

/*  The null descriptor indicates the end of the public object. */

    OM_NULL_DESCRIPTOR};
```

Service-Generated Public Object

Service-generated public objects are created by specific calls to the XOM API. Calls to the `om_get()` function return a service-generated public object. Code Example 3-2 segment illustrates the generation of a service public object.

Note that a service public object is not deleted when the workspace containing it is deleted. The `om_delete()` function must be used to destroy service-generated public objects.

Code Example 3-2 `extract.c` Segment

```
/*
 * Extract the details of an attribute
 */
static void
extract_one_attribute(OM_public_object attribute)
{
    OM_return_code    ret;
    OM_public_object  attr_id, global_form;
    OM_exclusions     exclusions;
    OM_type           types[2];
    OM_value_position total;
    sysAttr_t         attr;

exclusions = OM_EXCLUDE_ALL_BUT_THESE_TYPES |
OM_EXCLUDE_SUBOBJECTS;

types[0] = MP_ATTRIBUTE_ID;
types[1] = OM_NO_MORE_TYPES;

ret = om_get(attribute, exclusions, types, OM_FALSE, 0, 0,
               &attr_id, &total);

CHECK_OM_CALL("om_get MP_ATTRIBUTE_ID ", ret);

types[0] = MP_GLOBAL_FORM;
types[1] = OM_NO_MORE_TYPES;

/*  At this stage, attr_id is a generated service public object. */

ret = om_get(attr_id[0].value.object.object, exclusions, types,
               OM_FALSE, 0, 0, &global_form, &total);

CHECK_OM_CALL("om_get MP_GLOBAL_FORM ", ret)
```

Private Objects

Private objects are defined in an implementation-defined manner and cannot be directly accessed by the developer, as is the case for public objects. The contents of a private object can only be accessed through the XOM interface by calling `om_get()`, `om_put()`, `om_read()`, `om_remove()`, or `om_write()`. Private objects can only be created through direct calls to the `om_copy()` or `om_create()` routines of the XOM interface. When creating a private object, you can specify default values that can be used to initialize some of the attributes. Code Example 3-3 illustrates the use of private objects.

Code Example 3-3 `objtool.c` Segment

```
void
addGlobalFormClass(OM_workspace workspace, OM_private_object
obj,
                       OM_type type, char *val)
{
    OM_private_object     class;
    OM_return_code        ret;
    OM_descriptor         desc[2];
    OM_object             source= (OM_object) desc;

/*  Create an instance of type object class used to identify
 *  a managed object class
 */

ret = om__create(MP_C_OBJECT_CLASS, OM_FALSE, workspace,
&class);
CHECK_OM_CALL("om_create MP_C_OBJECT_CLASS", ret);

/*  Add the global form of the attribute into the object class.
 *  This must be a registered object class identifier.
 */

put_desc(&desc[0], MP_GLOBAL_FORM,
OM_S_OBJECT_IDENTIFIER_STRING,
       val, strlen(val));
ENDOBJ(desc[1]);

ret = om_put(class, OM_INSERT_AT_END, source, 0, 0, 0);
CHECK_OM_CALL("om_put MP_C_OBJECT_CLASS, MP_GLOBAL_FORM", ret)
```

```
/*  Add the object class to the supplied object.*/
put_desc(&desc[0], type, OM_S_OBJECT, &class, 0);
ENDOBJ(desc[1]);
ret = om_put(obj, OM_INSERT_AT_END, source, 0, 0, 0);
CHECK_OM_CALL("om_put MP_C_OBJECT_CLASS, MOC", ret)

om_delete(class);
```

To add information to a private object, a list of descriptors that correspond to the public representation of the information must be formatted. In the general example program, there is a function `put_desc()`, that adds descriptors to a private object. See Code Example 3-4 for the definition of `put_desc()`.

Code Example 3-4 `put_desc()` Function Segment

```
void
put_desc(OM_descriptor *desc_ptr, OM_type type, OM_syntax syntax,
               void *value, int len)
{
    desc_ptr->type = type;
    desc_ptr->syntax = syntax;
    switch(syntax) {
    case OM_S_BOOLEAN:
        desc_ptr->value.boolean =  (* (OM_boolean *) value);
        break;
    case OM_S_ENUMERATION:
        desc_ptr->value.enumeration = (* (OM_enumeration *) value);
        break;
    case OM_S_INTEGER:
        desc_ptr->value.integer = *(OM_integer *)value;
        break;
    case OM_S_OBJECT:
        desc_ptr->value.object.object = * (OM_object *) value;
        break;
    case OM_S_BIT_STRING:
    case OM_S_ENCODING:
    case OM_S_GENERAL_STRING:
    case OM_S_GENERALISED_TIME_STRING:
    case OM_S_GRAPHIC_STRING:
```

```
    case OM_S_IA5_STRING:
    case OM_S_NUMERIC_STRING:
    case OM_S_OBJECT_DESCRIPTOR_STRING:
    case OM_S_OBJECT_IDENTIFIER_STRING:
    case OM_S_OCTET_STRING:
    case OM_S_PRINTABLE_STRING:
    case OM_S_TELETEX_STRING:
    case OM_S_UTC_TIME_STRING:
    case OM_S_VIDEOTEX_STRING:
    case OM_S_VISIBLE_STRING:
        desc_ptr->value.string.length =
(OM_element_position)len;
        desc_ptr->value.string.elements = (void *)value;
        break;
    }
}
```

Code example 3-5 shows the ENDOBJ macro definition.

Code Example 3-5 ENDOBJ Macro Definition

```
#define ENDOBJ(attr)                                    \
        attr/**/.type=OM_NO_MORE_TYPES;                 \
        attr/**/.syntax=OM_S_NO_MORE_SYNTAXES;          \
        attr/**/.value.string.length = 0;               \
        attr/**/.value.string.elements = 0;
```

Using a private object may, in some circumstances, introduce an associated overhead. For instance, GetArg is an object used for formatting a get request. One attribute of GetArg, scope, is defined as an XOM object. Therefore, to set the scope of a get request using a private object requires four function calls:

- Create a scope object with om_create()
- Write the scope level in the scope object with om_put()
- Create the GetArg object with om_create()
- Put the scope object into the GetArg object with om_put()

However, using XOM private objects can clean up an application and free you from the burden of memory management. Abnormal memory consumption is very easy to track when using private and service-generated public objects.

3

Import/Export of Object Identifiers

Object identifiers are used to identify a number of things in XOM. For example, a unique object identifier is associated with each class definition.
Code Example 3-6 is from the `imports.h` include file; it shows some object identifier string definitions.

Code Example 3-6 `imports.h` Segment

```
#define OMP_O_MP_C_ACTION_ERROR\
/*2001*/   "\053\014\002\207\161\002\001\002\217\121"
#define OMP_O_MP_C_ACTION_ERROR_INFO\
/*2002*/   "\053\014\002\207\161\002\001\002\217\122"
```

A specific structure is provided by XOM to handle object identifiers. The structure defined in `xom.h` file is `OM_object_identifier`. All of the API functions dealing with *class identifiers* request a `OM_object_identifier` structure. This means that a constant definition of object identifier cannot be directly used. Instead, variables of type `OM_object_identifier` must be used. There is no restriction on the length of object identifiers.

Specific macros are defined by XOM in order to derive a variable of type `OM_object_identifier` from an object identifier string:

- `OM_EXPORT(MP_C_ATTRIBUTE)`—defines the variable `MP_C_ATTRIBUTE`, that is of type `OM_object_identifier`. Note that `OMP_O_MP_C_ATTRIBUTE` (object identifier string) should have been previously defined as a string constant. Code Example 3-7 illustrates an example of this type.

The `OM_EXPORT` macro defines variables and can be used only once for a specific object identifier in an entire application.

In the example program, the export definitions are gathered in a compilation unit called `exports.c`. The compilation unit is then linked with the entire application.

Code Example 3-7 `exports.c` Segment

```
OM_EXPORT(MP_C_ATTRIBUTE)
OM_EXPORT(MP_C_ATTRIBUTE_ID)
OM_EXPORT(MP_C_BASE_MANAGED_OBJECT_ID)
OM_EXPORT(MP_C_CMIS_GET_ARGUMENT)
OM_EXPORT(MP_C_CMIS_GET_RESULT)
```

The OM_IMPORT macro must be used in any compilation unit that refers to an object identifier.

- OM_IMPORT(MP_C_ATTRIBUTE)—imports the external declaration of the variable MP_C_ATTRIBUTE into this file.

In the example program, the import statements are placed in the include file imports.h and are imported into each compilation unit. Refer to Code Example 3-8.

Code Example 3-8 imports.h Segment

```
OM_IMPORT(MP_C_ATTRIBUTE)
OM_IMPORT(MP_C_ATTRIBUTE_ID)
OM_IMPORT(MP_C_BASE_MANAGED_OBJECT_ID)
OM_IMPORT(MP_C_CMIS_GET_ARGUMENT)
OM_IMPORT(MP_C_CMIS_GET_RESULT)
```

XOM Function Interface

The following functions are provided in the XOM interface:

om_copy—create a new private object that is an exact, but independent copy of an existing private object

om_copy_value—place or replace a string in one private object with a copy of a string in another private object. This handles segments of a string.

om_create—creates a new private object that is an instance of a particular class

om_decode—creates a new private object that contains the decoded form of an existing private object. This function is used to decode an encoded object.

om_delete—delete an instance of a private object or public service generated object

om_encode—create a new private object that contains the encoded form of an existing private object. This function is used to encode a decoded object.

om_get—create a public copy of a particular part of a private object with certain characteristics

om_instance—determine whether or not an object is an instance of a particular class or any of its subclasses

`om_put`—insert or replace, in one private object, a copy of the attribute value of another public or private object

`om_read`—read a segment of a string in a private object

`om_remove`—remove values of a particular attribute of a private object

`om_write`—write a segment of a string into a private object

Storage Management

An object occupies storage. The storage occupied by a public object is directly accessible to the client, while the storage occupied by a private object is not.

The storage an object occupies is allocated and released by the client if the object is client-generated, or by the service if the object is service-generated or private.

An object is accessed through an object handle. An object handle is the means by which the client supplies an object to the service as an argument of an interface function, and the service returns an object to the client as the result of an interface function. A public object handle is simply a pointer to the data structure containing the object attributes. A private object handle is a pointer to a data structure whose layout is implementation-specific and is unknown to the client. The client accesses a private object only through the XOM API functions.

The client creates a client-generated public object by using normal programming language constructs. The client is responsible for managing any storage involved.

The service creates service-generated public objects when `om_get()` is called and allocates any necessary storage. The client destroys a service-generated public object and releases the storage by applying the `om_delete()` function.

At any point in time, a private object is either accessible or inaccessible to the client. An object is accessible if the client possesses a valid object handle for it. The object is inaccessible if the client does not possess an object handle, or the handle is invalid. Should the client designate an inaccessible object as an argument, the effect on the service's subsequent behavior is undefined.

The service makes a private object accessible by returning an object handle as the result of a function in this or another (application-specific) interface. The client makes such an object inaccessible by applying the `om_delete()` function to it, or by supplying it as an argument of any other function that, according to the specification, makes the argument inaccessible.

A private object is also destroyed when the workspace containing it is destroyed. A service-generated public object is unaffected by the destruction of the workspace that generated it. A client-generated public object is not associated with a workspace.

The storage occupied by a service-generated public object must not be changed by the client, and the effect of doing so is undefined. This includes all values (strings, subobjects, integers, and so on). However, it is possible to use a value that is a private subobject as an argument to an interface function that modifies the subobject.

Workspaces

The service maintains private objects in workspaces. A workspace is a repository for instances of classes in the closures of one or more packages associated with the workspace. The implementations of the OM interface functions may differ from one workspace to another. A package may be associated with any number of workspaces. The OM package is implicitly associated with every workspace. Other packages may be explicitly associated with a workspace when it is defined with `mp_negotiate`.

The interface includes functions for effectively copying and moving objects from one workspace to another, provided that the object's classes are associated with both. How workspaces are created, made known to the client and destroyed is outside the scope of this document. In all cases, destroying a workspace effectively applies the `om_delete()` function to each private object it contains.

Chapter Summary

The OM API can be used to create, delete, and modify complex information objects. However, there is a difference between OM classes and managed object classes and between OM attributes and managed object attributes. The OM attribute constructs the interface, while managed object class and managed

object attributes provide the MIS access provision. The XOM interface provides twelve functions to manipulate objects: `om_copy`, `om_copy_value`, `om_create`, `om_decode`, `om_delete`, `om_encode`, `om_get`, `om_instance`, `om_put`, `om_read`, `om_remove`, and `om_write`.

Objects communicate with the client and service using descriptors (C structures). A descriptor is a data structure which is used to represent an OM attribute type and a single value. The interface element includes function names, typedef names, and constants.

Packages within the XOM are defined by ASN.1 object identifiers.

An object can contain one or more attributes. For instance, an object of class `CMIS-Get-Result` contains the attributes *managed object class, managed object instance, attribute list,* and *current time*. The XOM class is characterized by the type of attributes that can appear in its instances and as previously stated a class is denoted by the ASN.1 object identifier. There are two types of classes: *concrete* and *abstract*. A concrete class permits an OM instance to occur, while an abstract class forbids an instance to occur.

XOM objects are a way to represent ASN.1 syntaxes. Public and private objects are data structures that provide a service to other programs that wish to manipulate and access other systems management data abstraction services. Public and private objects are further defined into a groups of service-generated public object, service-generated private object, client-generated public object, and client-generated private objects.

Storage management of objects is accomplished through the manipulation of object handles. The storage an object occupies is allocated and released by the client if the object is client-generated or by the service (through the `om_delete()` function) if the object is service-generated, or private.

A workspace is a repository for instances of classes in the closures of one or more packages associated with the workspace. The OM package is implicitly associated with every workspace. Whenever a workspace is destroyed, the `om_delete()` function is applied to each private object.

Systems Management Protocol 4

This chapter provides an overview of the Management Protocols described in detail in *Solstice XMP Programming Reference*.

Overview	*page 39*
CMIS Services	*page 44*
Function Calls	*page 45*
Function Sequencing	*page 49*
Implementation Specific Enhancements	*page 50*
Chapter Summary	*page 52*

Overview

The X/Open Management Protocol (XMP) library is used in conjunction with the X/Open Object Management (XOM) library to provide an object-oriented network management application programming interface. This implementation is designed to offer services that are limited to the Common Management Interface Protocol (CMIP), as defined in the ISO-IS 9595/9596 specification standard.

This interface defines the communication to the Management Information Service (MIS) which is based on the Common Management Information Service (CMIS) and Common Management Information Protocol (CMIP) (ITU X.710 and X.711, ISO 9595 and 9596-1) standards.

The interface uses the generic systems management concepts defined by ISO and supports the model defined in the Structure of Management Information (SMI).

The interface also provides access to the MIS, which is abstracted in terms of notifications and operations on managed objects. It offers service primitives that correspond to the abstract services of the CMIS and the Simple Network Management Protocol (SNMP) of the Internet community.

The Management Information Base (MIB) is a conceptual repository of all management information. The MIB is modelled as a collection of managed objects; programs can access the managed objects through the interface to make queries, updates, or to generate reports.

Management services are modelled as specific managed objects, termed *management support objects*, which provide the services.

Access to MIS is done through the managed object in conjunction with the implementation of the general-purpose XOM as stated in the previous chapter.

All C language names are shown in *italic* typeface. A function is indicated by following parentheses *function()*, and a constant is surrounded by braces *{CONSTANT}*. The names of errors are surrounded by brackets *[ERRORS]*.

- `<xmp.h>` contains common definitions for the access to the Management Communication Service.
- `<xmp_cmis.h>` contains specific definitions that reflect the Abstract Services of the Common Management Information Service along with the ASN.1 productions of the related protocol (CMIP).

The Manager and Agent

A network management system consists of *manager* and *agent* processes. In the simplest form, a network management system really contains nothing more than protocols that convey information about network elements back and forth between various agents in the system and the manager processes. The MIB database is shared between the manager and agent to provide information about the managed network elements.

- *Manager*—directs the operations of the agent
- *Agent*—reports to the manager on the status of managed network elements and receives directions from the manager on actions it is to perform on these elements
- *MIB*—is used by both the manager and agent processes to determine the structure and content of management information

A conceptual overview of the interaction between a manager and agent is shown in Figure 4-1.

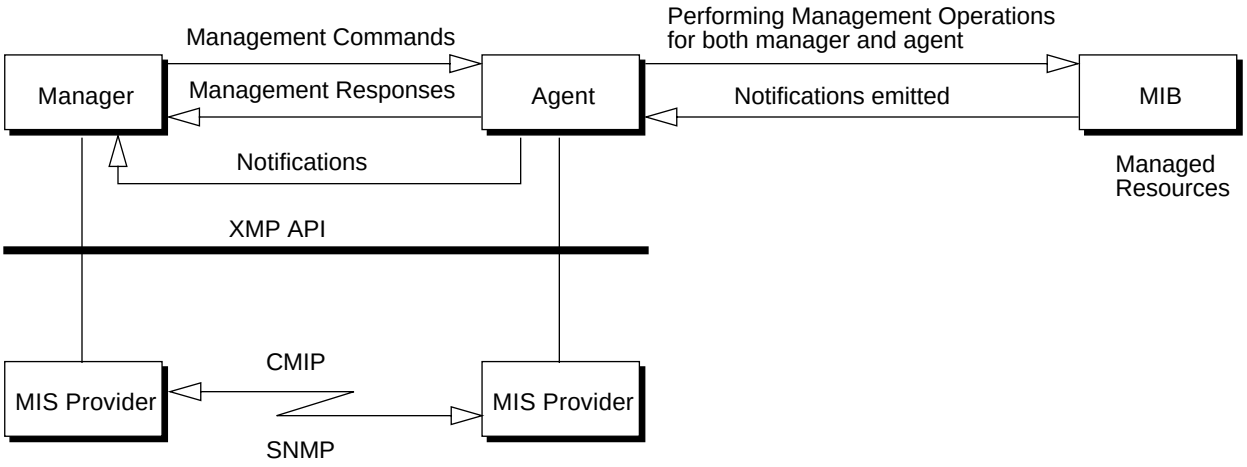

Figure 4-1 Manager / Agent Interaction

The XMP interface provides access to all facilities provided by the MIS Provider. This is “symmetrical” in the sense that it can be used to implement management programs acting in manager or agent roles. It supports:

- A management program acting as a manager accessing managed information from an agent.
- A management program acting as an agent interacting with a manager by receiving operation requests and sending back responses or event reports.

The interface sends *requests* on the invoker side and receives *indications* on the performer side within a management interaction. If the service is confirmed, the performer can send back *responses* that will be received as *confirmations* by the invoker. This communication path is shown in Figure 4-2.

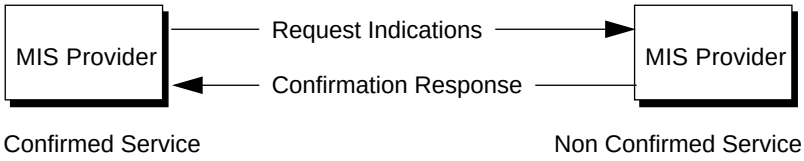

Figure 4-2 CMIS Interaction

C Naming Convention

The interface uses part of the C public namespace for its facilities. All identifiers start with the letters *mp, MP,* or *OMP.* See Table 4-1 for more details of the conventions used. The interface reserves *all* identifiers starting with the letters *mpP* for Private (that is, internal) use by interface implementations. It also reserves *all* identifiers starting with the letters *mpX* or *MPX* for vendor-specific extensions of the interface. You should not use any identifier starting with these letters.

The OSI-Abstract-Data Manipulation API uses similar, though not identical, naming conventions. All its identifiers are prefixed by the letters *OM* or *om*.

Table 4-1 C Naming Conventions

Element Type	**Prefix**	**Example Usage**
Functions	mp_	mp_initialize
Error “problem” values	MP_E_	MP_E_BAD_ADDRESS
Enumeration tags (except errors)	MP_T_	MP_T_NORMAL
OM class names	MP_C_	MP_C_ABORT_ARGUMENT
OM value length limits	MP_VL_	See below paragraphs
OM value number limits	MP_VN_	See below paragraphs
Other constants	MP_	See below paragraphs

Table 4-1 C Naming Conventions *(Continued)*

Element Type	Prefix	Example Usage
Reserved for implementors	mpP	Reserved for developer
Reserved for interface extensions	mpX	
Reserved for interface extensions	MPX	

A complete list of all identifiers used (except those beginning *mpP, mpX, MPX* or *OMP*) is provided in *Solstice XMP Programming Reference*. No implementation of the interface will use any other public identifiers. A *public identifier* is any name except those reserved in section 4.1.2.1 of the ISO C Standard. The *public namespace* is the set of all possible public identifiers.

The C identifiers are derived from the language-independent names used throughout this document by a mechanical process that depends on the kind of name:

- Interface function names are made entirely of lowercase letters and are prefixed by *mp_*. For example, **Get-Req()** would become *mp_get_req()*.
- C function parameters are derived from the argument and result names by making them entirely of lowercase letters. In addition, the names of results have *_return* added as a suffix. Thus the argument **Name** becomes *name*, and the result **Operation-Notification** becomes *operation_notification_return*.
- OM class names are made entirely of uppercase letters and are prefixed by *MP_C_*; therefore, **Get-Result** becomes *MP_C_GET_RESULT*. The symbolic OM class names are strictly those used in the abstract syntax ASN.1 of the CMIP except that names containing multiple words are separated with hyphens.
- Enumeration tags are derived from the name of the corresponding OM type and syntax by prefixing with *MP_*. The case of letters is left unchanged. Thus **Enum(Synchronization)** becomes *MP_Synchronization*.
- Enumeration constants, except errors, are made entirely of uppercase letters and are prefixed by *MP_T*, thus **Atomic** becomes *MP_T_ATOMIC*.
- The name of an OM attribute is local to its OM class. Therefore, the same name of an OM attribute may appear in different OM classes. For example, OM attribute **filter** is defined in both OM classes **Get-Argument** and **Set-Argument.** Independent-language attribute **filter** appears as

MP_FILTER in C-language. The symbolic OM attribute names are strictly those used in the abstract syntax ASN.1 of the CMIP with the exception that names containing multiple words are separated with hyphens.

- Errors are treated as a special case. Constants that are the possible values of the OM attribute **Error-Status** of a subclass of the OM class **Error** are made entirely of uppercase letters and are prefixed by *MP_E_*. Thus **no-such-object-instance** becomes *MP_E_NO_SUCH_OBJECT_INSTANCE*.
- The constants in the *Value Length* and *Value Number* columns of the OM class definition tables are also assigned identifiers. (They have no names in the language-independent specification.) The upper limit in one of these columns is not “1” (one), it is given a name consisting of the OM attribute name, prefixed by *MP_VL_* for value length or *MP_VN_* for value numbers.
- The sequence of octets for each object identifier is also assigned an identifier for internal use by certain OM macros. These identifiers are all upper case and are prefixed by *OMP_O_*.

CMIS Services

The communication of management information occurs between the manager and agent. This is illustrated in Figure 4-1. The OSI service for systems management is CMIS. Table 4-2 defines the CMIS services.

Table 4-2 CMIS Services

CMIS Services	**Type**
M-ACTION	confirmed / non-confirmed
M-CREATE	confirmed
M-CANCEL-GET	confirmed
M-DELETE	confirmed
M-EVENT-REPORT	confirmed / non-confirmed
M-GET	confirmed
M-SET	confirmed / non-confirmed

Function Calls

XMP functions are divided into these service groups:

- **Registration service**

`mp_bind/mp_unbind`	establishes/releases a session between XMP and user

- **CMIS services**

  `mp_create_req` and `mp_create_rsp`

  `mp_delete_req` and `mp_delete_rsp`

  `mp_get_req` and `mp_get_rsp`

  `mp_set_req` and `mp_set_rsp`

  `mp_cancel_get_req` and `mp_cancel_get_rsp`

  `mp_action_req` and `mp_action_rsp`

  `mp_event_report_req` and `mp_event_report_rsp`

- **Asynchronous services**

`mp_receive`	gets indications and confirmations of previous asynchronous operations
`mp_wait`	waits for incoming (indication or confirmation) messages
`mp_abandon`	discards reception of pending asynchronous result

- **XMP specific services**

`mp_error_message`	gives a full description of an error detected by the XMP library
`mp_get_assoc_info`	retrieves negotiated connection values
`mp_get_last_error`	retrieves the secondary return code of the most recent function call, communications or system error.
`mp_validate_object`	analyses an OM-Object and returns Bad-Argument details if necessary
`mp_initialize`	allocates a workspace for the session

`mp_shutdown`	releases the workspace at end of the session
`mp_negotiate`	negotiates the profile of the user

- **Association control services**

`mp_abort_req`	aborts a management session that is either connected or partially connected
`mp_assoc_req`	requests the creation of a management association
`mp_assoc_rsp`	replies to a request to create a management association
`mp_release_req`	requests the release of a management association
`mp_release_rsp`	replies to a request to release a management association

The example XMP programs are located in the directory `/opt/SUNWconn/cmip/examples`. These examples encompass the full set of XMP functionality such as synchronous/asynchronous modes and simple manager/agent code. Figure 4-3 on page 48 illustrates the state machine for the XMP function sequencing rules. Each state is represented in a box. These are individual interactions and are not global to the entire session.

Table 4-3 XMP Functions

Function	Description
`mp_abandon`	Abandons the local results of a pending asynchronously executing operation.
`mp_abort_req`	Aborts a management association
`mp_action_req`	Requests managed objects to perform an action.
`mp_action_rsp`	Replies to a previously requested confirmed action.
`mp_assoc_req`	Requests the creation of a management association.
`mp_assoc_rsp`	Replies to a previously requested operation to create a management association.
`mp_bind`	Opens a management session.
`mp_cancel_get_req`	Cancels the result of a pending get operation.
`mp_cancel_get_rsp`	Replies to a previously requested cancel-get operation.

Table 4-3 XMP Functions *(Continued)*

Function	Description
mp_create_req	Requests a create for a new managed object instance.
mp_create_rsp	Replies to a previously requested create operation.
mp_delete_req	Requests the deletion of a managed object instance.
mp_delete_rsp	Replies to a previously requested delete operation.
mp_event_report_req	Requests to report a notification emitted by an object.
mp_event_report_rsp	Replies to a previously reported notification.
mp_get_assoc_info	Retrieves negotiated connection values.
mp_get_last_error	Retrieves additional error information.
mp_get_req	Requests to retrieve management information.
mp_get_rsp	Replies to a previously requested get operation.
mp_initialize	Performs the necessary initialization of the interface.
mp_negotiate	Performs the negotiation features of the interface
mp_receive	Used to retrieve inbound messages.
mp_release_req	Requests the release of a management association.
mp_release_rsp	Replies to a previously requested operation to release a management association.
mp_set_req	Requests to change attribute values of managed object instances.
mp_set_rsp	Replies to a previously requested set operation.
mp_shutdown	Deletes a workspace and the associated resources.
mp_unbind	Terminates the given management session.
mp_validate_object	Returns an object that indicates the cause of a Bad-Argument return by an XMP function.
mp_wait	Waits for activity on one or more sessions.

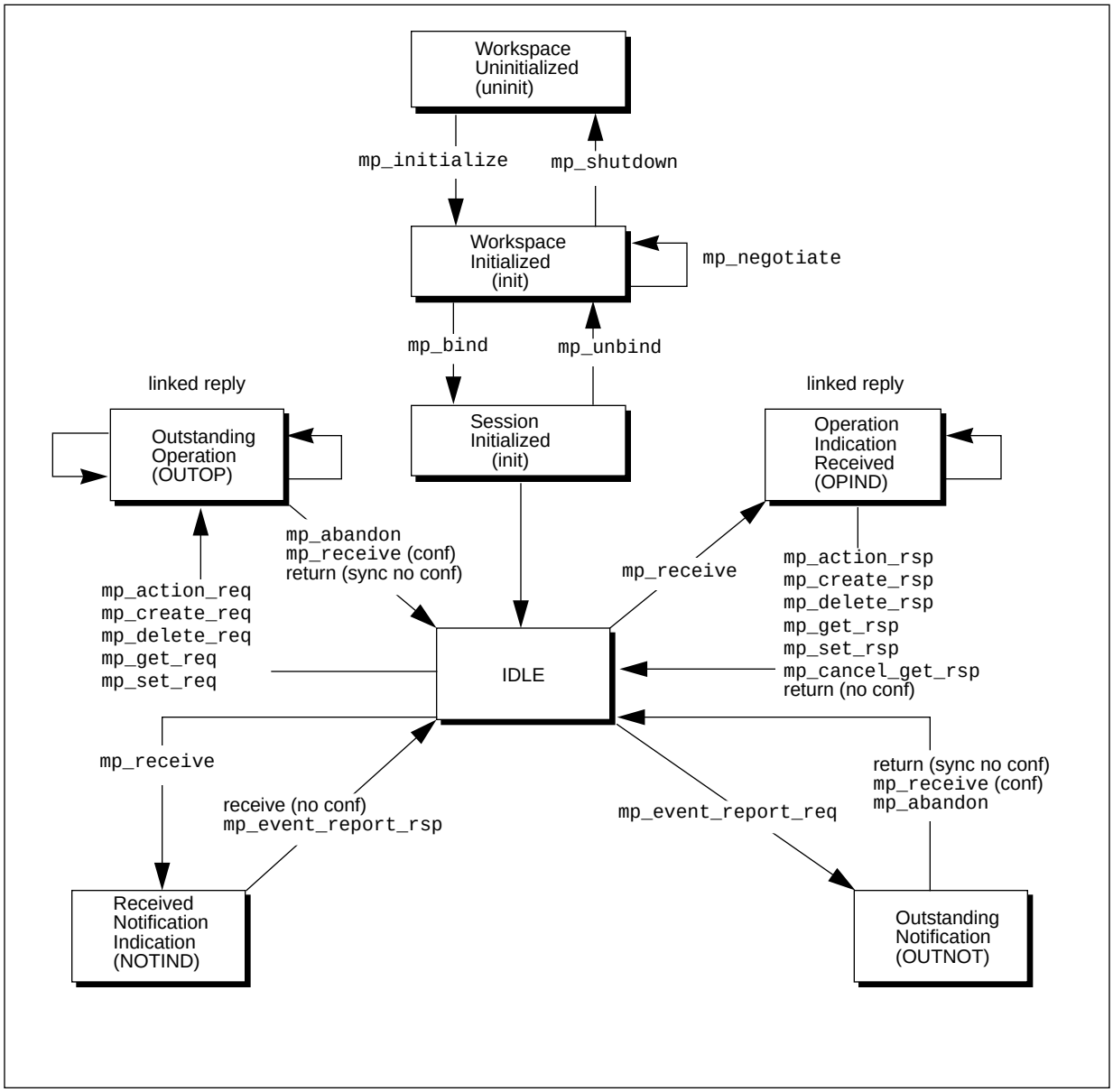

Figure 4-3 XMP Sequencing State Diagram

Function Sequencing

A minimum set of rules are necessary when using the interface services. These rules must be followed by the management programs to ensure that the interface functions are called in the relevant sequence. If these rules are not adhered to by the management program, the XMP API returns a library error.

XMP requires that the following general rules be followed:

- Initialize the workspace with the `mp_initialize()` function.
- Declare the list of Object Management (OM) packages that are supported with the `mp_negotiate()` function (optional).
- Use the `mp_bind()` function for opening sessions.

Note – A session identifies a particular link from the application program to the System Management Services provider (`osimcsd`). The session is passed as the first argument to most interface functions.

- Perform all the management interaction with the CMIS service interface functions.
- Upon the completion of management operations, close all the associated sessions by using the `mp_unbind()` function.
- Discard the workspace with the `mp_shutdown()` function. If you do not close all the sessions before the management application terminates, the XMP API will automatically close and release the remaining resources.

When you call the `mp_bind()` function, you have two alternatives:

1. If you want to bind using the default session, an OSI address must correspond to the default session. This is done with `cmiptool`. See *Solstice CMIP 8.2 Administrator's Guide* for more information.

   Note that only one call to `mp_bind()` using the default session can be performed on a system. If you initiate an additional call to `mp_bind()` with the default session, an error will occur until the default session is closed. Keep in mind, it is **not** possible to have two applications with the same address simultaneously open.

2. If you want to bind to a specific session, an address must be provided. This address must correspond to a valid address; otherwise, the call to `mp_bind()` will fail with an error code “bad address”. You can check the validity of an address with `cmiptrace`. See *Solstice CMIP 8.2 Administrator’s Guide* for more information.

Implementation Specific Enhancements

The following enhancements to the XMP specification are specific to the Solstice implementation of these protocols. Applications that do not take advantage of these enhancements are still supported.

`mp_negotiate()` *Function*

The following implementation-specific features can be negotiated using the `mp_negotiate()` function. The object identifiers for these features are defined in the header file `<cmip/xmp.h>`. These features are disabled by default.

```
#define OMP_O_MP_ONE_FD_PER_SESSION    "\x2a\x86\x3a\x00\x88\x1a\x06\x01\x03"
```

If this implementation-specific feature is enabled, each session has its own file descriptor. If this implementation-specific feature is disabled, all sessions share the same file descriptor. In either case, the file descriptor is set in the *file-Descriptor* attribute of the session object. One use of file descriptors is explained in “Managing Multiple Event Types” on page 64.

```
#define OMP_O_MP_ANY_APP_CONTEXT     "\x2a\x86\x3a\x00\x88\x1a\x06\x01\x04"
```

If this implementation-specific feature is enabled, then any application context can be specified in the *application-Context* attribute of the *acse-Args* attribute object of the session object. For outgoing associations, the application context is sent in the associate request PDU. Only associate request PDUs that contain a valid application context are accepted by incoming associations. By default, only the `ISO`, `NMF`, and `TMN` application contexts can be specified in the session object.

`mp_wait()` *Function*

The `mp_wait()` function has been extended to include an additional value for the *timeout* parameter, which specifies how long (in milliseconds) the function waits for activity on one or more sessions. If this parameter is set to `0`, the `mp_wait()` function waits indefinitely for activity on the session.

With the implementation-specific enhancement, the *timeout* parameter can also be set to `–1`, in which case, the `mp_wait()` function returns immediately if there is no activity on the session. This can be used as a polling mode to check for activity.

`proprietary-Args` *Attribute of Session Object*

The session object can contain the implementation-specific attribute *proprietary-Args*, which is identified by `MP_PROPRIETARY_ARGS` in the header `<cmip/xmp.h>`.

Each *proprietary-Args* object can contain the following attributes, and each attribute can appear zero or one times:

- *inactivity-Timer* (`MP_INACTIVITY_TIMER`)
  If present, this attribute sets the inactivity timer for the session. The value is an unsigned integer, specified in seconds. If this attribute is not present, the system-wide activity timer, which is configured using `cmiptool`, is used.

  Setting the *inactivity-Timer* attribute to zero (0) disables the inactivity timer. In this case, the association remains open until closed explicitly.

- *bind-State* (`MP_BIND_STATE`)
  Shows the current state of the session and can have value `MP_T_UNBOUND` or `MP_T_BOUND`.

- *connect-State* (`MP_CONNECT_STATE`)
  Shows the current state of the session and can have value `MP_T_UNCONNECTED` or `MP_T_PARTLY_CONNECTED` or `MP_T_CONNECTED`.

Normally the attributes *bind-State* and *connect-State* should not be modified as they are used internally by the XMP library. Sessions can be created using `mp_bind()` without defining these attributes.

However, these attributes can optionally be combined to specify that a session will be bound to the next incoming association, and no other. In this case, the user must perform the following steps:

1. **Before calling** `mp_bind()`**, create a session object with the** *bind-State* **and** *connect-State* **attributes defined.**

```
bind-State    =  MP_T_UNBOUND | MP_T_SINGLE_ASSOC
connect-State =  MP_T_UNCONNECTED | MP_T_SINGLE_ASSOC
```

2. **Pass the session object in the call to** `mp_bind()`**.**

Any attempt to set the attributes to an invalid value (for example, `MP_T_BOUND`) will cause the call to `mp_bind()` to fail.

AE Titles

An AE title can be specified in the `requestor-Title` and `responder-Title` attributes of the session object. AE titles are in object identifier form—that is, Form2 AE titles.

The `osimcsd` daemon can multiplex associations based on the AE title. This means that multiple associations can be established with the same presentation selector, provided they have different AE titles.

Chapter Summary

The XMP interface is defined by a series of standards that determines how communication to the MIS is handled. This access is done through notifications and operations on managed objects. Corresponding services for CMIS and SNMP are available in the XMP interface.

The interface to the C programming language is handled through specific identifiers that denote the XMP API. Function names are prefixed with `mp_` in lower case. Arguments also use the lower-case nomenclature for identification purposes. Errors are treated as special case and are communicated in upper-case with the prefix `MP_E`.

Management information between the *manager* and *agent* is done through the CMIS services. This service provides an operations notification communication link using `M-ACTION`, `M-CREATE`, `M-CANCEL-GET`, `M-DELETE`, `M-EVENT-REPORT`, `M-GET`, and `M-SET`. These operations allow access to managed information from an agent and the management program.

The XMP functions are divided into these groups: *registration services, CMIS services, asynchronous services, XMP specific services*, and *association control services*. Each group provides full XMP functionality in workspace operations. Function sequencing rules require that the workspace be initialized, declare the list of object management packages, use `bind()` to open a session, perform management interaction, close all associated sessions with `unbind()`, and discard workspace with `shutdown()`.

4

XMP Development Concepts 5

This chapter covers XMP/XOM development and other fundamental concepts associated with Solstice CMIP. All the necessary components for application development and object management are discussed.

Overview	*page 55*
Synchronous and Asynchronous Operations	*page 58*
Access Control	*page 59*
Error Codes	*page 59*
Session Objects	*page 60*
Context Objects	*page 62*
Managing Multiple Event Types	*page 64*
Packages	*page 65*
Chapter Summary	*page 65*

Overview

Initial Declaration

The environment containing the XOM objects is referred to as the *workspace*. Applications using XOM must create a workspace. This is done through a call to `mp_initialize()`. Refer to the Chapter 3, “Object Management (OM)” for additional details on the workspace creation process.

Once the workspace is initialized, the application can open a session by calling `mp_bind()`.

Connection Management

The Management Communication Service (MCS), which is implemented in the `osimcsd` process, automatically handles the associations between *agents* and *managers*. If the MCS association management is used, the XMP interface does not access the stack when the `mp_bind()` function is invoked. This means that access to the OSI stack is deferred until a management operation or management notification is received.

When an inactivity time-out occurs, associations controlled by the MCS are released without any notification to the XMP user.

Responder Versatility

You can change the responder in each function call using the same session. This facility gives the XMP developer the opportunity to perform any operation or notification with an agent or manager in the same work session.

Loopback Facility

The loopback feature allows communication between two management applications above the MCS without requiring a CMIS association. This saves resources and communication time. The loopback mode is transparent to both the *requestor* and *acceptor.*

A loopback is typically used for a configuration of an agent and a manager in the same system.

Each time the MCS establishes an association, it will first test if the target address is local. If so, a simulation of an association is implemented. Every request sent later to the destination address will be passed directly to the proper application.

MCS loopback occurs at the `osimcsd` level as shown in Figure 5-1 on page 57. This can be configured by `cmiptool`. See *Solstice CMIP 8.2 Administrator's Guide* for more information. Also, loopback can occur at the transport layer as shown in Figure 5-2 on page 57, if MCS loopback is not enabled.

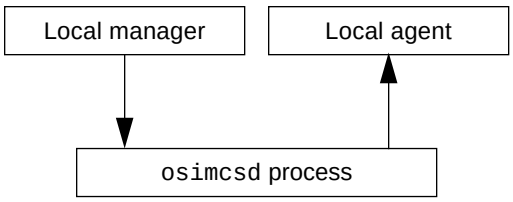

Figure 5-1 Loopback through `osimcsd`

Since underlying protocol layers are not used in MCS loopback mode, the MCS service interactions are routed without being fully checked. This could lead to a different reaction to service violations compared to a regular CMISE-based transfer.

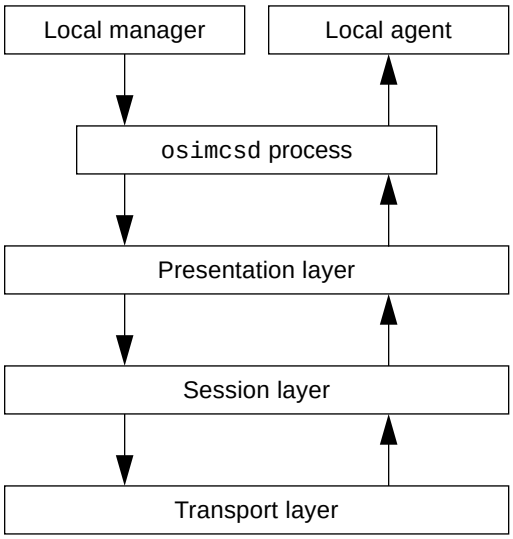

Figure 5-2 Loopback through Transport Layer

5

Synchronous and Asynchronous Operations

The synchronous or asynchronous mode of each requested operation or notification is specified and determined by the value of the *asynchronous* attribute in the context object. The context object is passed by the interface function. If the value of the *asynchronous* attribute is *false*, the operations will be synchronous.

Synchronous

During a synchronous operation or notification, the calling function is blocked until the service is complete. Note that a CMIS operation may have multiple responses. You may specify a time-limit or size-limit after which several responses can be returned. The function can return under the following conditions:

- When all the responses have been received in a specified period
- When the time limit occurs
- When the limit on the number of responses is reached by XMP

Asynchronous

In an asynchronous operation, the XMP function that is called immediately returns an identifier. This identifier can be used later to receive responses or to abandon the operation. This unique identifier is used to match the response to the request.

Results from the call are received with the `mp_receive()` function. This function returns one of the following values:

- `nothing`—if no response or request has been received.
- `incoming`—if an incoming request / notification is received from the peer.
- `outstanding`—if no response has been received.
- `completed`—if the complete response has been received.
- `partial`—in the case of multiple replies where only a subset of the responses have been received.

Access Control

There are no specific access restrictions in the XMP library. Access control is the responsibility of the agent or manager application. External access control, as defined by the XOM class *external*, is allowed.

Error Codes

A typical function call will return a success or a failure code. A successful function call returns the specified appropriate attributes. Any other reply indicates a failure.

Unbind/Shutdown Errors

When a request containing a scoping specification is received by the agent, it may send multiple responses to the manager application. If the agent application calls `mp_shutdown()` or `mp_unbind()` before sending the last response indicating the end of the linked replies, the manager application will receive an error message indicating that the agent has terminated the work session.

The returned error constants for `mp_shutdown` are:

- `MP_NO_WORKSPACE`
- `MP_INSUFFICIENT_RESOURCES`

`mp_unbind` can return a system-error or one of the following library-errors:

- `bad-class`
- `bad-session`
- `miscellaneous`
- `session-terminated`

`mp_unbind` can return the following error codes:

- `MP_NO_WORKSPACE`
- `MP_INVALID_SESSION`
- `MP_INSUFFICIENT_RESOURCES`

Asynchronous Mode Errors

When multiple replies are possible, the following errors can occur when a function is used in asynchronous mode:

Library Error TIME-LIMIT-EXCEEDED—indicates that the request has timed out; however, any replies that have been received will be returned by XMP.

Library Error SIZE-LIMIT-REACHED—indicates that the maximum number of replies specified by the XMP user has been reached. The extra replies will be discarded.

Service Error MULTIPLE-REPLY-ERRORS—indicates that among all the replies there is an error.

Session Objects

A *session* is the link between the XMP API and the application. A session is initialized with the `mp_bind()` function.

`mp_bind()` returns a session object that contains all the parameters for the session. This function is called in one of two ways:

- Without a session object, XMP will use the default session and return the default session parameters in the session object
- With a session object that has parameter values set, XMP will use the parameters to setup the session

Multiple concurrent sessions can be used. The maximum number of concurrent sessions in a workspace is 16. The maximum number of workspaces in a process is 16. This means that a single UNIX process can open up to 256 sessions simultaneously. A complete set of guidelines can be found in *Solstice XMP Programming Reference*.

The rules of a session are defined as follows:

- A session can be restricted to allow interoperation only with a designated management application. If these descriptors are omitted in the session object, the session allows interoperation with any application.

Note – A descriptor is a data structure that is used to represent an object management attribute type and a single value.

- Once initiated, no attribute of the session object can be changed: XOM utilities will refuse all user updates.
- If different from the default session, the session parameter provided to the `mp_bind()` function must be compliant with the XOM definition of the session object class. In particular the file descriptor attribute must be provided. The value should be `MP_NO_VALID_FILE_DESCRIPTOR`.

The definition of the session class, as supported by the implementation, is listed in Table 5-1.

Table 5-1 Session Object Attributes

OM Attribute	Syntax Value	Length Value	Number Value	Initial Value
requester-Address	Object (Address)	—	0 or 1	—
requestor-Title	Object (Title)	—	0 or 1	—
role	Integer	—	0 or 1	see below
file-Descriptor	Integer	—	1	see below
presentation-Layer-Args	Object (Presentation-Layer-Args)	—	0 or 1	—
acse-Args	Object (Asce-Args)	—	0 or 1	—
cmip-Assoc-Args	Object (Cmip-Assoc-Args)	—	0 or 1	—
standard-Externals	Object (Standard-Externals)	—	0 or 1	—

Default Session Object Attributes

The defined structure for the default session object are:

- No requestor address

  This is the default local address that has been configured with `cmiptool`. See *Solstice CMIP 8.2 Administrator's Guide* for more information.

- No requestor title
- Performs both manager roles (managing / monitoring) and agent (performing / reporting)
- File descriptor set to `MP_NO_VALID_FILE_DESCRIPTOR`
- acse-Args specify
  - no responder address
  - no responder title

  - no authentication information
  - application context set to ISO (obj id <2.9.0.0.2>)
  - no user-info
- cmip-Assoc-Args specifies
  - no access control
- A functional unit that describes the maximum profile of a CMIS user:
  - Multiple-object-selection, filter operations, multiple-replies, and cancel-get operations are allowed.
  - no user-info
- Standard-Externals is NULL (no SMASE user data)

Context Objects

The *context* object defines a number of parameters that are common to many XMP function calls. All of these parameters are collected into a particular OM object so that the parameters don't have to be passed one-by-one to the XMP functions. The default OM attribute values can be used instead of building your own context object.

Various administrative details are contained in the context object. These include: synchronous or asynchronous mode, the size limit of the response, confirmed or unconfirmed mode, and priority of the request.

If an attribute is defined in both the session and context object, the context attribute value takes precedence over the session attribute value. This rule applies to the *responder-address, responder-title,* and *access-control* attributes.

If the time limit and size limit attributes are provided with negative values, or if the values exceed the allowed range, the context object is rejected and the request/response is refused with the code `BAD_CONTEXT`.

The context object must contain the attributes *priority, asynchronous,* and *mode.* The mode attribute is only meaningful for the action, set, and event requests. Table 5-2 lists the attributes for context objects.

Table 5-2 Context Object Attributes

Common Arguments				
OM Attribute	**Value Syntax**	**Value Length**	**Value Number**	**Initial Value**
extensions	Object(Extension)	-	0 or more	-
Service Controls				
access-Control	Object(Access-Control)	-	0-1	-
connection-Id	Integer	-	0-1	-
mode	Enum(Mode)	-	1	confirmed
priority	Enum(Priority)	-	1	medium
responder-Address	Object(Address)	-	0-1	-
responder-Title	Object(Title)	-	0-1	-
Local Controls				
asynchronous	Boolean	-	1	false
size-Limit	Integer	-	0-1	-
time-Limit	Integer	-	0-1	-

Default Context Object Attributes

The defined structure for the default context object is:

- No extension is supported.
- No access control value.
- The mode of the request is confirmed.
- The priority of the request is medium.
- No responder address.
- No responder title.
- The synchronous mode is used.
- The size limit is set to the constant `XMP_DEFAULT_SIZE_LIMIT`, which allows a minimum number of linked replies.
- No time limit. The system-wide activity timer, which is configured using `cmiptool`, is used.

Restrictions

The following context attributes are not supported:

- Extension
- Priority

If these attributes are provided, syntax analysis will be performed on them. If the analysis fails, the XMP API will reject the object and return an error `BAD_CONTEXT`.

Managing Multiple Event Types

If your applications need to process several types of events, you can use the `select()` or `poll()` system call followed by the `mp_wait()` command to obtain the correct session to pass to the `mp_receive()` function.

To manage multiple event types:

1. **Use the `poll()` or `select()` system call to obtain file descriptors on which an event has occurred.**
2. **If the active file descriptor corresponds to one or more sessions (that is, a CMIP event) pass the list of sessions to the `mp_wait()` function.**
   The result of this function call is a list of active sessions. You can omit this step if you have implemented one file descriptor per session as described in “mp_negotiate() Function” on page 50.
3. **Pass each active session to the `mp_receive()` function.**
   The *Session* object you specify in each call to `mp_receive()` must be one of the active sessions returned by the call to `mp_wait()` in Step 2.

Packages

Related classes are grouped into a collection called a package. There are two packages of interest in the XMP interface: Common Management Service and CMIS Management Service.

Common Management Service Package

The common management service package includes classes for XMP error management and classes common to SNMP and CMIP. Some of the object management classes defined in this package are not supported in this implementation of XMP:

- Community Name access control subclass
- Entity Name title subclass
- Network-Address address subclass (SNMP)
- CMOT-system-id definition
- SNMP Object Name (SNMP package)
- Name and Relative Name (XDS definition)
- Name-String as Name subclass (XDS definition)

CMIS Management Service Package

The CMIS management service package contains a collection of XOM class definitions that represent CMIS services.

Chapter Summary

The environment which contains the objects is defined as a workspace. Any application using XOM must create a workspace with the function call `mp_initialize()`. Once this has been done, a session must be opened with a `mp_bind()`.

The MCS transparently handles the association service by hiding the communication establishment. Before issuing any CMIS operation or notification, the MCS controls the remote I/O operation. Automatic association control is very helpful to developers; however, programmatic manipulation control is available through the XMP interface.

Loopback through `osimcsd` provides a method of communication between the manager and agent without using the CMIS association. This is usually used in a local system configuration where the underlying OSI layers are not used. However, loopback through the transport layer provides communication to the presentation, session, and transport layers.

Synchronous and asynchronous operations provide an attribute checking procedure that verifies an identifier for appropriate responses. The identifier is a unique value that is handled by `mp_receive()`.

Accessing the XMP library is done with the manager and agent application; however, an external method through the XOM is also supported. Functions return a success or failure code. An error code in the asynchronous mode of operation can specify a library error of: `TIME-LIMIT-EXCEEDED`, `SIZE-LIMIT-REACHED`, and `MULTIPLE-REPLY-ERRORS`.

A *session* is defined as a link between the XMP API and the application. It is initialized with `mp_bind()`. Multiple concurrent session is supported to a maximum number of 16. It can be used as an acting manager, that is, invoker of management operations and performer of management notifications. The same acting principle is true for the agent.

The context object emphasizes the number of parameters common to XMP function calls. Applications can assume that an object of OM context created with default values from its attributes will work with the interface. The constant `MP_DEFAULT_CONTEXT` can be used as an argument to the interface functions instead of creating an OM object with default values.

There are two packages important to Solstice CMIP, CMS and CMIS. The CMS service package provides XMP classes with an error management, while the CMIS service package contains a collection of XOM class definitions.

Addressing 6

This chapter describes situations where an application has to provide local and remote addressing.

Remote Addressing	*page 67*
Local Addressing	*page 68*
Chapter Summary	*page 70*

Remote Addressing

Addressing a remote network management application can be done in two ways:

- Specialized session
- Specialized context

Specialized Session

When you open a session with `mp_bind()`, you can specify a remote management application with the *responder address* and / or *responder title* attribute. These attributes must be provided in the acse-Args of the session object. Any operation requested that does not contain one or more of these attributes is sent by the MCS to the default local OSI address.

Note that this default address is configured with `cmiptool`. For more information, see *Solstice CMIP 8.2 Administrator's Guide*.

Specialized Context

If the responder address and/or responder title is specified in the context object, the underlying MCS will attempt to send the request to the given address. Note, that if these attributes are defined in the context object, they take precedence over the attribute values in the session object. If a connection cannot be established, the XMP function will return the bad address with an error code.

Local Addressing

Each application must have a different Presentation Service Access Point (PSAP) address that consists of:

- Presentation Selector (PSEL)
- Session Selector (SSEL)
- Transport Selector (TSEL)
- Network Service Access Point (NSAP)

Refer to *Solstice CMIP 8.2 Administrator's Guide* for actual configuration information.

Figure 6-1 illustrates the address of two applications in a restricted static configuration. The following addresses are listed in `PSEL/SSEL/TSEL/NSAP` format:

```
AppA/Prs/CMIP/TrsAddr

AppB/Prs/CMIP/TrsAddr
```

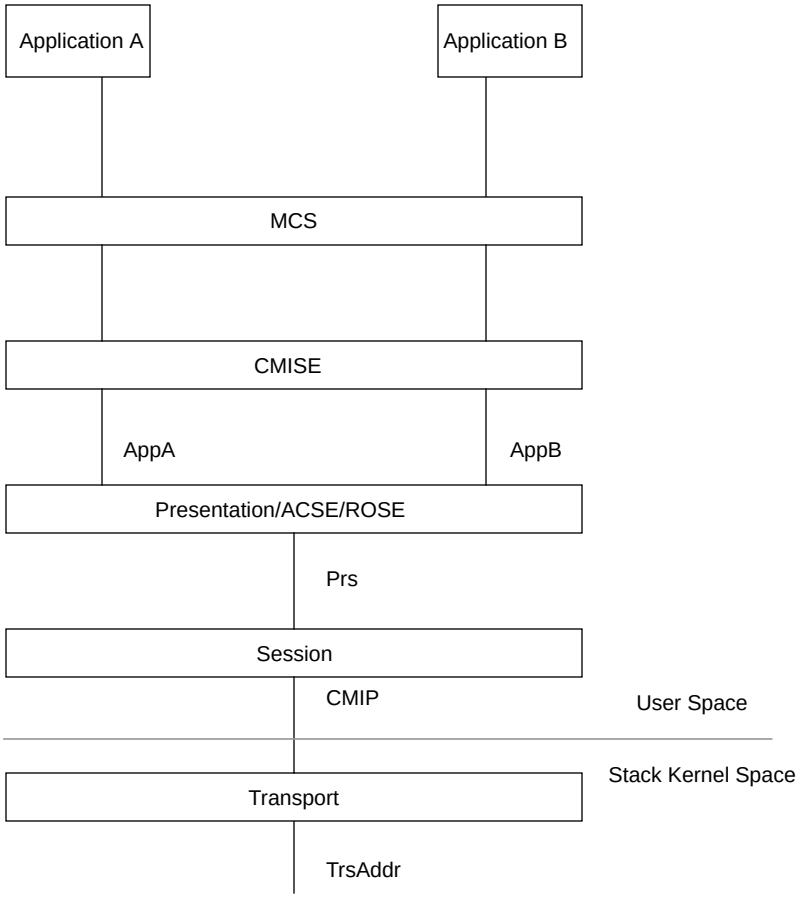

Figure 6-1 Addressing Scheme

6

Chapter Summary

The network locations of management program instances are referenced by *addresses*. Addressing a remote management application can be accomplished in one of two ways: specialized session and specialized context.

A specialized session is opened with a call to `mp_bind()`. The remote managed application can be specified with a responder address, or responder title, or both. One of these attributes must be provided in the session object.

The specialized context is specific to the context object. Attributes that are defined in the context object take precedence over the session object.

Compiling and Linking Application Programs 7

This chapter describes how to compile and link network management applications with the XMP/XOM libraries. It includes instructions for compiling and running the example programs supplied with Solstice CMIP SDE.

Library Contents	*page 71*
Include File Structure	*page 72*
Compile and Link Procedure	*page 72*
Example Makefile	*page 73*
Running the Example Programs	*page 74*

Library Contents

Applications must be linked with the libraries that provide the XMP and XOM functions. These libraries are provided as shared libraries. The library files are contained in the `/opt/SUNWconn/lib` directory. They are:

- `libxmp.so` provides the services defined in the XMP API
- `libxom.so` contains the XOM object management library

7

Include File Structure

The XOM and XMP include files are defined in the specification. The include files are located in `/opt/SUNWconn/include/cmip`.

- `xom.h`

  This header defines the types, symbols, macros for objects, and declarations for the available services.

- `xomi.h`

  This header defines the types, symbols, macros for objects, and declarations for the available services.

- `xmp.h`

  This header declares the interface functions and the structures that pass information between those functions. Constants that are used by the functions and structures are also defined.

- `xmp_cmis.h`

  This header declares the interface functions and the structures that pass information between those functions. Constants that are used by the functions and structures is also defined.

Note – All of these files can be used with ANSI C and C++ compilers.

Compile and Link Procedure

To compile and link the include files for XMP/XOM, you must add the following to your `Makefile`:

```
CFLAGS  += -I/opt/SUNWconn/include
LDFLAGS += -L/opt/SUNWconn/lib   -lxmp -lxom -R/opt/SUNWconn/lib
```

Example Makefile

The following is an example of a Makefile used to compile and link the manager and agent example applications provided with Solstice CMIP SDE.

```
######################
# Global variables
######################
CDEBUGFLAGS= -DSVR4 -O
CC=cc $(CDEBUGFLAGS)

OSIAM=/opt/SUNWconn
INC=-I$(OSIAM)/include
OBJ_UTI= contexts.o exports.o objtools.o system.o
OBJ_MAN= getarg.o extract.o
OBJ_AGT= getres.o
EXE= manager agent
all: includes exec
includes: address.h system.h imports.h macros.h
clean: clean_obj clean_exec
       @echo “Cleaning ‘pwd‘”
veryclean: clean
           @rm -rf .??* core *~ *% *.BAK
exec: $(EXE)
clean_obj:
           @rm -f $(OBJ_UTI) $(OBJ_AGT) $(OBJ_MAN) $(OBJ_XVIEW)
           $(EXE:%=%.o)
clean_exec:
           @rm -f $(EXE)
.c.o: $(INCLUDES)
       $(CC) $(INC) -c $<
manager:
       $$.c $(OBJ_UTI) $(OBJ_MAN)
$(LIB_XMP:%=$(OSIAM)/lib/lib%.a)
       $(CC) $@.c $(INC) -o $@ $(OBJ_UTI) $(OBJ_MAN) \
       -L$(OSIAM)/lib $(LIB_XMP:%=-l%)
agent:
       $$@.c $(OBJ_UTI) $(OBJ_AGT)
$(LIB_XMP:%=$(OSIAM)/lib/lib%.a)
       $(CC) $@.c $(INC) -o $@ $(OBJ_UTI) $(OBJ_AGT) \
       -L$(OSIAM)/lib $(LIB_XMP:%=-l%)
```

7

Running the Example Programs

Example programs are provided with Solstice CMIP SDE. Each one consists of a manager and an agent application that communicate through the `osimcsd` process and the transport provider. You can run these programs without the transport provider, using loopback mode, provided the manager and agent are running on the same machine.

Note – Before compiling and linking any of the examples, copy the contents of the directory containing the example to another location. The file access permissions of the directories under `/opt/SUNWconn` do not allow ordinary users to run `make`.

Example 1

This example returns and displays information about the machine on which the agent application is running. It is located in:

```
/opt/SUNWconn/cmip/examples/xmp
```

The manager application performs the following tasks:

1. **Gets the address of a remote agent to query**
2. **Sends a `get` request to the agent**
3. **Waits for a response**
4. **Displays the message received from the agent**
5. **Shuts down the API and resets**

The agent application performs the following tasks:

1. **Performs an `mp_bind()`**
2. **Waits for an incoming request from a manager**
3. **If the incoming request is a `get` request, returns a `linked-get` response**
4. **Returns to wait state**

Compiling and Linking

The `make` command is used to build both the manager and the agent applications. The `Makefile` is shown in "Example Makefile" on page 73.

To compile and link the example programs:

1. Copy the example files to another directory.

```
prompt% cp /opt/SUNWconn/cmip/examples/xmp/* <dest_dir>
```

2. Change to the destination directory (<*dest_dir*>).

```
prompt% cd <dest_dir>
```

3. Compile and link the agent application.

```
prompt% make cmipagent
```

4. Compile and link the manager application.

```
prompt% make cmipmanager
```

Running Example 1

You must start the agent application before the manager application. If you are not running the transport provider, you must start the agent and the manager on the same machine and enable loopback mode.

By default, the manager and agent run locally in loopback mode over a TCP/IP (RFC 1006) connection.

The agent has a default address: `PSEL=rfc0`, `SSEL=Prs`

The manager has a default address: `PSEL=rfc1`, `SSEL=Prs`

Note that if you set the environment variable `CMIP_ASYNC`, the `cmipmanager` application will send its `Get` request in asynchronous mode.

To start the agent using the default address:

```
prompt% <dest_dir>/cmipagent
Performing mp_initialize()
Performing mp_bind()

Agent : iteration 1
Performing mp_wait()
```

To start the manager using the default address:

```
prompt% <dest_dir>/cmipmanager
```

On the agent side, you will see the following series of events:

```
Performing mp_initialize()
Performing mp_bind()

Agent : iteration 1
Performing mp_wait()
Performing mp_receive()
Performing mp_get_rsp()

Agent : iteration 2
Performing mp_wait()
```

On the manager side, you will see the following series of events:

```
Performing mp_initialize()
Performing mp_bind()
Performing mp_get_req()
Results of mp_get_req()

The remote system is called papyrus
The system was manufactured by Sun_Microsystems
Number of processors configured is 1
Number of processors currently online is 1
CPU info : Processor 0, a 40 MHz sparc CPU + sparc FPU, is online.
The size of physical memory (in Mbytes) is 48
System supports POSIX version 1

Performing mp_unbind()
Performing mp_shutdown()

cmipmanager: done at Fri Jul  8 18:16:43 1994
```

The manager application stops automatically when it has completed its request and received a response. You must stop the agent explicitly by typing `Ctrl-C`.

You can alter the default address for the agent and the manager by entering the following command-line options:

Note – The session selector cannot be changed.

Address of Local Application

–N <*addr*> : local network service access point

–T <*tsel*> : local transport selector

–P <*psel*> : local presentation selector

Address of Remote Application

–n <*addr*> : remote network service access point

–t <*tsel*> : remote transport selector

–p <*psel*> : remote presentation selector

All addresses that are entered in hexadecimal must be preceded by `0x`.

For example:

To set up a connection over TCP/IP (RFC 1006) with loopback mode disabled, you must specify the agent address when you start the manager, even if the agent is running locally.

On the agent side, type:

```
prompt% <dest_dir>/cmipagent
```

On the manager side type:

```
prompt% <dest_dir>/cmipmanager –n 0x<tcp/ip address>
```

The TCP/IP address of the agent application is displayed in the bottom right corner of `cmiptool` running on the machine on which the agent is started. It must be entered exactly as it appears in `cmiptool`, preceded by `0x`.

To set up a connection over CLNP, where the agent listens on PSEL=tp40 and the manager listens on PSEL=tp45.

On the agent side, type:

```
prompt% <dest_dir>/cmipagent –P tp40 –N 0x49<local>01
```

On the manager side, type:

```
prompt% <dest_dir>/cmipmanager –N 0x49<local>01 –P tp45
        –n 0x49<remote>01 –p tp40
```

Example 2

This example returns a linked-get response that shows a number of different object types. The information returned is always the same, regardless of the system on which the example is launched. The example is located in:

```
/opt/SUNWconn/cmip/examples/xmp2
```

The manager application performs the following tasks:

1. **Gets the address of a remote agent to query**
2. **Sends a** `get` **request to the agent**
3. **Waits for a response**
4. **Displays the message received from the agent**
5. **Shuts down the API**

The agent application performs the following tasks:

1. **Performs an** `mp_bind()`
2. **Waits for an incoming request from a manager**
3. **Returns a** `linked-get` **response containing static information**
4. **Shuts down the API**

Compiling and Linking

The `make` command is used to build both the manager and the agent applications.

To compile and link the example programs:

1. Copy the example files to another directory.

```
prompt% cp /opt/SUNWconn/cmip/examples/xmp2/* <dest_dir>
```

2. Change to the destination directory (<*dest_dir*>).

```
prompt% cd <dest_dir>
```

3. Compile and link the agent application.

```
prompt% make ex_agent
```

4. Compile and link the manager application.

```
prompt% make ex_manager
```

Running Example 2

You must start the agent application before the manager application. If you are not running the transport provider, you must start the agent and the manager on the same machine and enable loopback mode.

By default, the manager and agent run in loopback mode over a CLNP (LLC1) connection. Both the agent and the manager take the default XMP address defined using `cmiptool` if no other address is specified; therefore, you must change the address of one of these applications if they are running on the same machine.

To start the agent using the default address:

```
prompt% <dest_dir>/ex_agent
```

On the agent side, you will see the values used to initialize the agent, followed by the list of objects that will be returned in response to a `get` request from the manager.

To start the manager using the default address:

```
prompt% <dest_dir>/ex_manager
```

On the manager side, you will see the values used to initialize the manager, followed by the `get` request sent to the agent and the list of objects returned. This should be identical to the list of objects displayed by the agent.

You can alter the default address for the agent and the manager by entering the following command-line options:

Note – The session selector cannot be changed.

Address of Local Application

`–N` <*addr*> : local network service access point

`–T` <*tsel*> : local transport selector

`–P` <*psel*> : local presentation selector

Address of Remote Application

`–n` <*addr*> : remote network service access point

`–t` <*tsel*> : remote transport selector

`–p` <*psel*> : remote presentation selector

All addresses that are entered in hexadecimal must be preceded by `0x`.

For example:

To set up a connection over CLNP, where the agent listens on `PSEL=tp40` and the manager listens on `PSEL=tp45`.

On the agent side, type:

```
prompt% <dest_dir>/ex_agent –P tp40 –N 0x49<local>01
```

On the manager side, type:

```
prompt% <dest_dir>/ex_manager –N 0x49<local>01 –P tp45
        –n 0x49<remote>01 –p tp40
```

Example 3

This example demonstrates the use of the XMP association management primitives (for example, `mp_assoc_req()`, `mp_assoc_rsp()`). The example is located in:

```
/opt/SUNWconn/cmip/examples/xmp
```

Compiling and Linking

The `make` command is used to build both the manager and the agent applications.

To compile and link the example programs:

1. Copy the example files to another directory.

```
prompt% cp /opt/SUNWconn/cmip/examples/xmp/* <dest_dir>
```

2. Change to the destination directory (<*dest_dir*>).

```
prompt% cd <dest_dir>
```

3. Compile and link the agent and manager applications.

```
prompt% make assoc
```

Running Example 3

You must start the agent application before the manager application. If you are not running the transport provider, you must start the agent and the manager on the same machine and enable loopback mode.

To start the agent using the default address:

```
prompt% <dest_dir>/assoc_agent
```

To start the manager using the default address:

```
prompt% <dest_dir>/assoc_manager
```

7

Enhancements to Draft 7 Preliminary Specification

A

Solstice CMIP SDE 8.2 and SunLink CMIP 8.1 conform to the *X/Open CAE Specification: System Management Protocol (XMP) API*, which supersedes Draft 7 of the *preliminary specification*. A previous release of this product, SunLink CMIP 8.0, conforms to the older revision of the specification. This appendix describes how to modify applications that were developed in accordance with Draft 7 of the *X/Open Management Protocol (XMP) API* so that they can be recompiled and linked using Solstice CMIP SDE.

Error Handling	*page 85*
OM Class Definitions	*page 87*
Automated Connection Management	*page 88*

Error Handling

Interface functions now return integer values instead of OM private objects (with three integer exceptions). This removes the need to overload the return value with either integers or pointers, and allows programmers to make quick, high-level decisions based on integer return codes. The interface has been extended to provide new functions in this area.

A

Draft 7 Preliminary Specification

Every function returns a status value that is either zero (success) or an error.

An error can be either an integer constant or an error object. If it is an error object, it can be of either class `Bad-argument`, or one of the subclasses of class error—`Communications-Error`, `Library-Error`, `CMIS-Service-Error`, or `System-Error`. An error object reports the base error in the *problem* attribute and an additional error value in the *parameter* attribute.

The Receive function (`mp_receive`) can return a `Communications-Error` or `Service-Error` in the *Operation-Notification-Status* parameter.

CAE Specification

Every function returns a status value that is either zero (success) or non-zero (error).

If the error returned is `MP_E_COMMUNICATIONS_PROBLEM`, `MP_E_BROKEN_SESSION`, or `MP_E_SYSTEM`, the Get-Last-Error function (`mp_get_last_error`) can be used to get an additional integer error value in the *parameter* attribute.

If the error is `MP_E_BAD_ARGUMENT`, the Validate-Object function (`mp_validate_object`) can be used to return an object that contains the details of the bad argument. The object returned is of class `Bad-Argument`, which is equivalent to the same class in Draft 7.

Service errors are no longer reported as the return code of the function. They are reported in the *Result* parameter of the `mp_get_req()` function, for example. In this case, the function will return zero (success) and it is the responsibility of the application to check the *Result* parameter to see if it contains a *result* (for synchronous calls), *invoke id* (for asynchronous calls), or an *error* and to react accordingly.

The Receive function (`mp_receive`) can return a `Service-Error` in the *Result-or-Argument* parameter.

OM Class Definitions

Modifications have been made to the OM class definitions in order to bring them into alignment with the relevant standards. In addition, the following changes have been introduced:

There are fewer attributes in the `Session` object, because some attributes have been moved into sub-objects. For example, *Responder-Address* and *Responder-Title* have been moved into the `Acse-Args` sub-object.

The `Access-Control` class has been renamed `External-AC`. This class appears as an attribute in all arguments to requests (`get`, `set`, `create`, `delete`, `action`). For example, `CMIS-Get-Argument` and `CMIS-Set-Argument`.

In `CMIS-Get-Result()`, `CMIS-Set-Result()`, `CMIS-Create-Result()`, and `CMIS-Create-Argument()` the *attribute-List* attribute is now a `Setof-Attribute` object, which contains one more objects.

In `CMIS-Filter()`, the *and* and *or* attributes are now `Setof-CMIS-Filter` objects, which contain one or more objects.

In `CMIS-Get-List-Error()`, the *get-Info-List* attribute is now a `Setof-Get-Info-Status` object, which contains one or more objects.

In `CMIS-Set-List-Error()`, the *set-Info-List* attribute is now a `Setof-Set-Info-Status` object, which contains one or more objects.

In `CMIS-Set-Argument()`, the *modification-List* attribute is now a `Modification-List` object, which contains one or more objects.

In `Object-Instance()`, the `local-DN` attribute is now a `DS-DN` object, not a `DS-RDN` object.

A

Automated Connection Management

The XMP interface now provides support for connection management applications. If automated connection management is disabled, applications are responsible for the establishment and release of associations. The interface has been extended to provide new functions in this area:

- `Abort-req()`
- `Assoc-req()`
- `Assoc-rsp()`
- `Get-Assoc-Info()`
- `Release-req()`
- `Release-rsp()`

Draft 7 Preliminary Specification

The preliminary specification does not allow applications to negotiate the Automatic Connection Management parameter; therefore associations are always managed by the entity that implements the protocol—for example, the `osimcs` daemon in the case of SunLink CMIP 8.0.

CAE Specification

The CAE specification allows applications to negotiate the Automatic Connection Management parameter; therefore automatic connection management can be disabled. In this case, the application is responsible for managing all associations. Automatic Connection Management is enabled by default.

Session Argument with ACM Disabled

If automatic connection management (ACM) is disabled, the session argument to XMP functions after an association is established must be the session object returned in *Assoc-Result*.

Draft 7 Preliminary Specification

Draft 7 of the preliminary specification did not allow ACM to be disabled; therefore the session argument was always the private object returned from `mp_bind()`.

CAE Specification

The CAE specification allows automatic connection management (ACM) to be disabled. If ACM is disabled, the session argument to XMP functions after an association is established must be the session object returned in *Assoc-Result*. This object is returned to the user by `mp_assoc_req()`, `mp_assoc_rsp()`, or through an `MP_ASSOC_CNF` primitive received by `mp_receive()`.

Synchronous Operation With ACM Disabled

To obtain the session argument to XMP functions for synchronous operations with ACM disabled:

1. **Call `mp_bind()` to bind to the session.**
   The result of this function call is a *Bound-Session* object that contains the bound session.

2. **Call `mp_assoc_req()` to request the creation of a management association.**
   The *Session* object you specify in this function call must be the *Bound-Session* object returned by the call to `mp_bind()` in Step 1. The result of this function call is an *Assoc-Result* object that contains the connected session.

3. **Call the appropriate XMP function to execute the required CMIS primitive.**
   The *Session* object you specify in this function call must be the session contained in the *Assoc-Result* object returned by the call to `mp_assoc_req()` in Step 2.

Asynchronous Operation With ACM Disabled

To obtain the session argument to XMP functions for asynchronous operations with ACM disabled:

1. **Call** `mp_bind()` **to bind to the session.**
   The result of this function call is a *Bound-Session* object that contains the bound session.

2. **Call** `mp_assoc_req()` **to request the creation of a management association.**
   The *Session* object you specify in this function call must be the *Bound-Session* object returned by the call to `mp_bind()` in Step 1. The results of this function call are:

   - A partially-connected session
   - An *Invoke-ID* integer that identifies the management operation

3. **Call** `mp_wait()` **to wait for the availability of management messages from the bound session.**
   The *Bound_session_list* you specify in this function call must include the *Bound-Session* object returned by the call to `mp_bind()` in Step 1.

4. **Call** `mp_receive()` **to retrieve the completed result of the asynchronously executed management operation.**
   The *Session* object you specify in this function call must be the *Bound-Session* object returned by the call to `mp_bind()` in Step 1. Among the results of this function call are:

   - An *Assoc-Result* object that contains a fully-connected session.
   - An *Invoke-ID* integer that identifies the management operation. It is the same as the *Invoke-ID* integer returned by the call to `mp_assoc_req()` in Step 2

5. **Call the appropriate XMP function to execute the required CMIS primitive.**
   The *Session* object you specify in this function call must be the *Assoc-Result* object returned by the call to `mp_receive()` in Step 4.

Compliance Information and Product Limitations B

This appendix lists the standards and specifications with which Solstice CMIP 8.2 complies. It also details the specific limitations to this implementation of these specifications.

Compliance Information	*page 91*
Product Limitations	*page 92*

Compliance Information

The CMIP protocol implementation conforms to the following specifications:

- ISO-IS-9595 *Version 2 Common Management Information Service (CMIS)* (CCITT X.710)
- ISO-IS-9596 *Version 2 Common Management Information Protocol (CMIP)* (CCITT X.711)
- ISO 10040 *Systems Management Overview* (CCITT X.701)
- ISO 10165-1 *Structure of Management Information (SMI)* (CCITT X.720)
- ISO 10165-4 *Guidelines for the Definition of Managed Objects (GDMO)* (CCITT X.722)

B

The application programming interface to Solstice CMIP conforms to the following X/Open CAE Specifications:

- *Systems Management—Management Protocols (XMP) API* dated March 1994.
- *OSI-Abstract-Data Manipulation (XOM) API* dated November 1991.

Product Limitations

Note the following specific limitations to the Solstice implementation of CMIP and XMP:

General

Solstice CMIP supports the Common Management Service (CMS) package and the CMIS Management Service package only. The workspace cannot be extended to support the SNMP package.

Packages used inside XMP return features that are supported by the OM package negotiation definition. Support for the following features is *not* provided in this product release:

- SNMP Object Class Definition
- SNMP Communication Stack
- DMI Object Class Definition
- Encoding/Decoding of CMIS and SNMP OM Objects
- `Get_next_req` service
- Local Automatic Name Resolution
- Use of XDS Features

Security

XMP does not provide a security mechanism. This is application dependent; the access control structure can be used to implement security for the manager or agent application.

Session Object Attributes

A session represents the link between a user and the XMP API. The session object contains a set of parameters for the link between the XMP API and the given user.

Maximum Number of Sessions

XMP supports multiple concurrent sessions with different users. The maximum number of concurrent sessions in a workspace is 16. The maximum number of workspaces in a process is 16. This means that a single UNIX process can open up to 256 sessions simultaneously. This limit is an internal parameter of the XMP implementation.

XMP does not use the Requestor Address session attribute. If this attribute is provided to the `mp_bind()` function, it is syntactically checked by the XMP API; however, the information it carries is not used by the XMP API. When the XMP API detects an erroneous attribute syntax, an error object is returned to the XMP user with the code `BAD_SESSION`.

Context Object Attributes

The XMP API does not use the following context attributes:

- Extension
- Priority

If these attributes are provided, syntactic analysis will be performed on them. If the analysis fails, the XMP API will reject the object and return the error `BAD_CONTEXT`.

Interface Objects

The following XOM classes are not supported by the XMP API:

- `Community Name` as Access Control subclass
- `Entity Name` as Title subclass
- `Kerberos-Ticket` as Access Control subclass
- `User-Password` as Access Control subclass

- `Network-Address` as Address subclass (SNMP)
- `CMOT-system-id` definition
- `SNMP Object Name` (SNMP package)
- `Name` and `Relative Name` (XDS definition)
- `Name-String` as Name subclass (XDS definition)

Using Loopback Mode

When loopback mode is enabled (the default condition), communication that is looped back—that is, communication between a manager and agent on the same machine—does not support the following Sun-specific features:

- The optional `OMP_O_MP_ANY_APP_CONTEXT` feature to `mp_negotiate()`.
- The `MP_T_SINGLE_ASSOC` flag passed in the session object to `mp_bind()`, which specifies that the next incoming association should be accepted.

`mp_validate_object()` *Function*

In the Solstice implementation of XMP, the `mp_validate_object()` function cannot be used to validate an OM object in advance of an XMP function call; however, it can be used to return information following an XMP function call—for example, `mp_get_req()`—that fails with a *Bad-Argument* return code.

The *Bad-Argument* object returned by the `mp_validate_object()`function cannot be deleted using `om_delete()` because it is service generated, and any attempt to do so will result in an error. This object will be deleted by the next function call.

Note that you should not need to validate an OM object in advance of a function call, because the SunLink implementation of XMP prevents the creation of syntactically incorrect objects. The `om_create()`, `om_put()`, and `om_copy()` functions all return an error code and display a description of the problem if the private object that would be created as a result of the call is malformed.

`mp_assoc_rsp()` *Function*

The *response* parameter passed to the `mp_assoc_rsp()` function must be a private object. This allows a *session* object to be inserted and returned to the user. If a public object is passed as the *response* parameter, the function call will fail with the message: `Library Error: Not Supported`

`mp_release_rsp()` *Function*

The `mp_release_rsp()` can only be used to acknowledge a release; therefore it is not possible to refuse an association release.

`mp_get_assoc_info()` *Function*

The `mp_get_assoc_info()` function defined by the *X/Open CAE Specification: System Management Protocol (XMP) API*, is not fully implemented in Solstice CMIP. Only the responder-address and responder-title of the *ACSE_args* object are returned.

If automatic connection management is not used, all information that could be returned by the `mp_get_assoc_info()` function is available in the session returned in the *Assoc-Result* object. The *Assoc-Result* object is returned to the user by `mp_assoc_req()`, `mp_assoc_rsp()`, or through an `MP_ASSOC_CNF` primitive received by `mp_receive()`.

B

Glossary

Abstract class

A class that has no instances, it can be written with the expectation that its subclasses will add to its structure and behavior, usually by completing the implementation of its (typically) incomplete methods.

active object

An object that encompasses its own thread of control.

actor

An object that can both operate upon other objects but is never operated upon by other objects. An agent is usually created to do some work on behalf of an actor or another agent.

Association Context

The *association context* contains all information related to the CMISE association such as, *function negotiation units*, *application context*, and *remote address*.

attribute

Object management uses the term *attribute* to denote a managed object construct.

base class

This is the most generalized class in a class structure. Most applications have many such base classes. Some languages define a primitive base class, which serves as the ultimate superclass of all classes.

behavior

Behavior dictates how an object acts and reacts, in terms of its state changes and message passing.

blocking object

This is a passive object whose semantics are guaranteed in the presence of multiple threads of control.

class

A set of objects that share a common structure and a common behavior. The terms *class* and *type* are usually (but not always) interchangeable; a class is a slightly different concept than type, in that it emphasizes the importance of hierarchies of classes.

class category

A collection of classes, some of which are visible to other class categories, and others of which are hidden.

class diagram

This is part of the object-oriented design notation, used to show the existence of classes and their relationships in the logical design of a system. A class diagram may represent all or part of the class structure of a system.

class structure

The “kind of” hierarchy of a system; a graph whose vertices represent classes and whose arcs represent relationships among these classes. The class structure of a system is represented by a set of class diagrams.

class variable

A placeholder for part of the state of a class. Collectively, the class variables of a class constitute its structure. A class variable is shared by all instances of the same class.

client

An object that uses the resources of another, either by operating upon it or by referencing its state.

Common Management Information Protocol (CMIP)

CMIP provides services that allow two OSI management service users to set up actions to be performed on managed objects, to change attributes of the objects, and to report the status of the managed objects.

Common Management Information Service (CMIS)

CMIS is the entity that provides the services and protocols specified in ISO-IS 9595/9596.

connectionless service/protocol

A service/protocol in which individual data units, messages, or packets are transferred one after the other without any relation to one another, without prior connection setup or subsequent take-down; usually, with a high probability (but no guarantee) of delivery, no sequencing, flow, or error control This is similar to unregistered letters in a postal system. OSI recognizes the concept of a connectionless service/protocol at layers 2 through 7, though it is not actually defined at each layer.

concurrency

The property that distinguishes an active object from one that is not active. Concurrency is one of the fundamental elements of the object model.

concurrent objects

An active object whose semantics are guaranteed in the presence of multiple threads of control.

constructor

An operation that creates an object and/or initializes its state.

container class

A class whose instances are collections of other objects. Container classes may denote homogeneous collections (all of the objects in the collection are of the same class) or heterogeneous collections (each of the objects in the collection may be of a different class, although all must share a common superclass). Container classes are most often defined as generic or parameterized classes, with some parameter designating the class of the contained objects.

context

The context defines the characteristics of the management interaction that are specific to a particular management operation/notification.

context objects

This is an XOM object that contains context data which defines the characteristics of management interaction.

descriptor

A descriptor is a defined data structure that is used to represent an object management attribute type and a single value. The structure has three components: *type*, *syntax*, and *value*.

dynamic binding

Binding denotes the association of a name (such as a variable declaration) with a class; dynamic binding is a binding in which the name/class association is not made until the object designated by the name is created (at execution time).

encapsulation

The process of hiding all of the details of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods. The terms *encapsulation* and *information hiding* are usually interchangeable. Encapsulation is one of the fundamental elements of the object model.

entity

An OSI term referring to a component that implements a protocol at some level. Entities are defined in all seven layers. They can be reached through the SAPs of the immediately inferior layer and offer SAPs to entities of the immediately superior layer.

field

A repository for part of the state of an object; collectively, the fields of an object constitute its structure. The terms *field*, *instance variable*, *member object* and *slot* are interchangeable.

function

In the context of a requirements analysis, a single, outwardly visible and testable behavior.

hierarchy

A ranking or ordering of abstractions. The two most common hierarchies in a complex system include its class structure (the kind of hierarchy) and its object structure (the part of hierarchy); hierarchies may also be found in the module and process architectures of a complex system. Hierarchy is one of the fundamental elements of the object model.

information hiding

The process of hiding all the details of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods.

inheritance

A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes. Inheritance defines a kind of hierarchy among classes in which a subclass inherits from one or more superclasses; a subclass typically augments or redefines the existing structure and behavior of its superclasses.

InterProcess Communication (IPC)

IPC messaging allows processes to send and receive messages, and to queue messages for processing in an arbitrary order. Unlike the file byte-stream model of data flow used for pipes, each IPC message has an explicit length.

instance

An instance has state, behavior, and identity. The structure and behavior of similar instances are defined in their common class. The term *instance* and *object* are interchangeable.

instance variable

A repository for part of the state of an object. Collectively, the instance variables of an object constitute its structure. The terms *fields, instance variable, member object,* and *slot* are interchangeable.

interface

The outside view of a class, object, or module, which emphasizes its abstraction while hiding its structure and the secrets of its behavior.

levels of abstraction

The relative ranking of abstractions in a class structure, object structure, module architecture, or process architecture. In terms of its part of hierarchy, a given abstraction is at a higher high level of abstraction than others if it builds upon the others; in terms of their kind of hierarchy, high-level abstractions are generalized, and low-level abstractions are specialized.

Management Communication Service (MCS)

The MCS is the entity that provides connectionless CMIS service by handling association management.

Management Information Base (MIB)

The MIB is a conceptual repository of all management information, which is a modelled collection of managed objects that programs can access through the interface in order to make queries, updates, or reports.

member function

An operation upon object, defined as part of the declaration of a class; all member functions are operations, but not all operations are member functions. The terms member function and methods are usually interchangeable. In some languages, a member function stands alone and may be refined, but serves as part of the implementation of a generic function or virtual function, both of which may be redefined in a subclass.

member object

A repository for part of the state of an object; collectively, the member objects of an object constitute its structure. The terms *field, instance variable, member object,* and *slot* are interchangeable.

message

An operation that one object performs upon another. The terms *message, methods,* and *operation* are usually interchangeable.

method

An operation upon an object, defined as part of the declaration of a class; all methods are operations, but not all operations are methods. The terms message, method, and operation are usually interchangeable. In some languages, a method stands alone and may be redefined in a subclass; in other languages, a method may not be redefined, but serves as part of the implementation of a generic function or a virtual function, both of which may be redefined in a subclass.

object

An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class. The terms *instance* and *object* are interchangeable.

object management (OM)

In X/Open's definition of managed objects, an object can be created, modified, and deleted with this mechanism.

object management attribute

An object management attribute is an arbitrary category where a specific value is placed.

object management class

An object management class is a category of managed object, it determines the object management attributes that may be present in the managed object, and details of constraints.

object model

The collection of principles that form the foundation of object-oriented design; a software engineering paradigm emphasizing the principles of abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistence.

object structure

The “part of” hierarchy of a system; a set of graphs whose vertices represent objects and whose arcs represent relationships among those objects. An object diagram may represent all or part of the object structure of a system.

object-oriented programming

A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united through inheritance relationships. In such programs, classes are generally viewed as static, whereas objects typically have more dynamic nature, which is encouraged by the existence of dynamic binding and polymorphism.

osimcsd

This is a process that resides in the user space of Solstice CMIP. It provides the communication mechanism between the XMP library and transport provider.

package

A package is a set of object management classes that are grouped together because they are functionally related, for example, SNMP service package.

package closure

A package closure is the set of classes that need to be supported in order to create all possible instances classes defined in the package. Hence, an object management class may be defined to have an object management attribute whose value is a managed object of an object management class that is defined in some other package, but within the same package-closure.

peer-to-peer

A relationship in which both parties are equal in stature (control and capability are evenly distributed).

polymorphism

A concept in type theory, according to which a name (such as a variable declaration) may devote objects to many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways.

private

A declaration that forms part of the interface of a class, object, or module; what is declared as private is not visible to any other classes, objects, or modules.

private objects

Private objects are held in data structures that are private to the service and can only be accessed from programs indirectly using interface functions. They are of particular use for structures that are infrequently manipulated by programs being passed by reference to the service. This allows private objects to be manipulated efficiently. An example, objects in the XMP session object.

protected

A declaration that forms part of the interface of a class, object, or module, but that is not visible to any other classes, objects, or modules except those that represent subclasses.

protocol

The ways in which an object may act and react, constituting the entire static and dynamic outside view of the object; the protocol of an object defines the envelope of the object's allowable behavior.

public

A declaration that forms part of the interface of a class, object, or module, and that is visible to all other classes, objects, and modules that have visibility to it.

public object

Public objects are represented by data structures that are manipulated directly using programming language constructs. Use of public objects simplifies programming by direct access and enabling objects to be statically defined, where appropriate. Programs can efficiently access public objects.

Remote Procedure Call (RPC)

A set of network protocols that allow a node to call procedures that are being executed on a remote machine.

Request (RQ)

A message unit that signals initiation of a particular action or protocol.

Response (RSP)

A message unit that acknowledges receipt of a request; a response consists of a response header, a response unit, or both.

scoping

This term describes the selection of a set of the managed objects in an MIT to which a filter is to be applied. For example, OSI network management permits the definition of levels of managed objects. The base object is the higher-level identifier in the identification tree.

sequential object

A passive object whose semantics are guaranteed only in the presence of a single thread control.

Service-Access-Point (SAP)

The point at which services are provided by an *n*-entity to an n+1 entity, in a layered communication system.

session

A session identifies if the agent or manager will be sent a management operation / notification. This management request contains **Bind-Arguments** such as, the name of the requestor.

session object

This is an XOM object that contains session data on the agent and manager.

slot

A repository for part of the state of an object; collectively, the slots of an object constitute its structure. The terms *field*, *instance variable*, *member object*, and *slot* are interchangeable.

state

One of the possible conditions in which an object may exist, characterized by definite qualities that are distinct from other quantities; at any given point in time, the state of an object encompasses all of the (usually static) properties of the object and current (usually dynamic) values of each of these properties.

state space

An enumeration of all the possible states of an object. The state space of a software object encompasses an indefinite yet finite number of possible (although not always desirable nor expected) states.

static binding

Binding denotes the association of a name (such as a variable declaration) with a class; static binding is a binding in which the name / class association is made when the name is declared (at compile time) but before the creation of the object that the name designates.

strongly typed

A characteristic of a programming language, according to which all expressions are guaranteed to be type-consistent.

structure

The concrete representation of the state of an object. An object does not share its state with any other object, although all objects of the same class do share the same representation of their state

subclass

A class that inherits from one or more classes, which are called its immediate superclass.

superclass

The class from which another class inherits, the inheriting class is referred to as a subclass.

thread of control

A single process, the start of a thread of control is not the root from which independent dynamic action within a system occurs; a given system may have many simultaneous threads of control, some of which may dynamically come into existence and then cease to exist. Systems executing across multiple CPUs allow for true concurrent threads of control, whereas systems running on single CPU can only achieve the illusion of concurrent threads of control.

type

The definition of the domain of allowable values that an object may possess and the set of operations that may be performed upon the object. The terms class and type are usually (but not always) interchangeable; a type is a slightly different concept than a class, in that it emphasizes the importance of enforcing the type of the object.

typing

The enforcement of the class of an object, which prevents objects of different types from being interchanged or, at the most, allows them to be interchanged only in very restricted ways. Typing is one of the fundamental elements of the object model.

User Context

The user context contains information that is related to a user's registration such as, user capabilities, user role, default address, and default time-out values.

workspace

A workspace is allocated storage that contains one or more package-closures, together with an implementation of the systems management data abstraction services. It supports all the OM classes of OM objects in the package-closure.

Index

A

abstraction, 3, 13
access method, 2
ACM (automated connection management), 88, 89
agent, 1, 13
algorithms, 12
arguments, 6
association context, 19
association management, 18
asynchronous operation, 58
attribute
  defined, 4
  of managed object, 3
  of managed resource, 10
attribute for default session object, 61
attribute value assertion (AVA), 9
attributes for context objects, 62
automated connection management (ACM), 88, 89
AVA (attribute value assertion), 9

B

bad address, 50
base managed object, 11
behavior, 3, 5
block diagram
  Addressing Schema, 69
  Global Architecture Overview, 17
  Loopback to osimcs, 57
  Loopback to Transport Layer, 57
Boolean expression, 12

C

CFLAGS, 72
characteristics, 4
classification, 5
CMIP (common management information protocol), 1
  messages, 5
CMIS (common management information service), 1, 15, 20
  service primitives, 4
common management information protocol (CMIP), 1
  messages, 5
common management information service (CMIS), 1, 15, 20
  service primitives, 4
common management service package, 65
compile, 72
concurrent sessions, 60
conditional package, 4, 5

connection management, 56
constraints, 5
containment hierarchy, 9, 10, 13
context object, 62
    default, 63
current status, 4
current time, 26

D

default context object, 63
default session object, 61
definition of the session class, 61
dependencies, 5
device-dependent protocol, 2
directives, 2
distinguished name (DN), 9

E

encapsulation, 3, 13
entry, in management information tree (MIT), 10
error
    MULTIPLE-REPLY-ERRORS, 60
    SIZE-LIMIT-REACHED, 60
    TIME-LIMIT-EXCEEDED, 60
error codes, 59
external events, 5

F

filter, 12

H

hierarchy, object containment, 9, 10, 13
higher-level system, 2

I

inactivity time, 18
include files, 72
information hierarchy, 4
inheritance, managed object class, 6
instance, 5, 13
Inter Process Communication (IPC), 16
internal events, 5
International Telecommunications Union (ITU-T), 7
IPC (Inter Process Communication), 16
ISO registration tree, 6
ITU-T (International Telecommunications Union), 7
ITU-T X.701/ISO-10040 System Management Overview, 7
ITU-T X.711/ISO-9596-1 Common Management Information Protocol Specification, 7
ITU-T X.720/ISO-10165-1 Management Information Model, 6, 7
ITU-T X.730/ISO-10164-1 Object Management Function, 7

L

`LDFLAGS`, 72
`libxmp.so`, 71
`libxom.so`, 71
limitation
    context object attributes, 93
    interface objects, 93
    security, 92
    session object attributes, 93
linked replies, 13
loopback, 56
lower-level system, 2

M

managed
    object, 3, 13
    resource, 10
    system, 2, 10
management
    directives, 1
    operation, 3, 13

protocol, 1
service, 1
management communication service (MCS), 15, 18, 56
management information tree (MIT), 10, 13
nodes, 10
management service, common, 65
manager, 1, 13
managing
process, 5, 13
MCS (management communication service), 15, 18, 56
messages, CMIP, 5
MIT (management information tree), 10
`mp_abandon()` function, 45
`mp_abort_req()` function, 46
`mp_assoc_req()` function, 46
`mp_assoc_rsp()` function, 46
`mp_bind()` function, 45
rules for, 49
`mp_error_message()` function, 45
`mp_get_assoc_info()` function, 45
`mp_get_last_error()` function, 45
`mp_initialize()` function, 45, 55
`mp_negotiate()` function, 46
`mp_receive()` function, 45
`mp_release_req()` function, 46
`mp_release_rsp()` function, 46
`mp_shutdown()` function, 46
`mp_unbind()` function, 45
`mp_validate_object()` function, 45
`mp_wait()` function, 45

N

name binding, 9
name, of attribute, 4
naming attribute, 9
nodes, management information tree (MIT), 10
notification, 3, 5

O

object
class, 5, 13
containment hierarchy, 9, 10, 13
definition, 3
entries, 10
instance, 5, 6, 13
method, 3, 5
naming, 9, 13
system, 10
top class, 6
object identifier (OID), 6, 7
Open Systems Interconnection (OSI) systems management model, 1
operations, 4, 13
OSI (Open Systems Interconnection) systems management model, 1
`osimcsd` process, 16

P

package, 5
conditional, 4, 5
protocol
device-dependent, 2
management, 1

R

registration tree, 6
relative distinguished name (RDN), 9
responder address attribute, 67
restrictions, 6
rules for `mp_bind()`, 49

S

scoped objects, 13
scoping, 11
search algorithms, 12
service primitives, 4
session, 60
session class definition, 61

session objects, 60
shared libraries, 71
software abstraction, 3, 13
Specialized Session, 67
subclass, 6
subtree, 11
superclass, 6
superior object, 10
synchronous operation, 58
system object, 10
systems management model, 1

T

top object class, 6
type, of attribute, 4

U

user context, 19

V

values, of attribute, 4

W

workspace, 55

X

XMP
    functions, 45
    services, 18