Interworking Between CORBA and TMN Systems Specification

New Edition: August 2000
Version 1.0

PATENT

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

NOTICE

The information contained in this document is subject to change without notice. The material in this document details an Object Management Group specification in accordance with the license and notices set forth on this page. This document does not represent a commitment to implement any portion of this specification in any company's products.

ISSUE REPORTING

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the issue reporting form at *http://www.omg.org/library/issuerpt.htm*.

Contents

Contents

Contents

Preface

About the Object Management Group

The Object Management Group, Inc. (OMG) is an international organization supported by over 800 members, including information system vendors, software developers and users. Founded in 1989, the OMG promotes the theory and practice of object-oriented technology in software development. The organization's charter includes the establishment of industry guidelines and object management specifications to provide a common framework for application development. Primary goals are the reusability, portability, and interoperability of object-based software in distributed, heterogeneous environments. Conformance to these specifications will make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems.

OMG's objectives are to foster the growth of object technology and influence its direction by establishing the Object Management Architecture (OMA). The OMA provides the conceptual infrastructure upon which all OMG specifications are based.

What is CORBA?

The Common Object Request Broker Architecture (CORBA), is the Object Management Group's answer to the need for interoperability among the rapidly proliferating number of hardware and software products available today. Simply stated, CORBA allows applications to communicate with one another no matter where they are located or who has designed them. CORBA 1.1 was introduced in 1991 by Object Management Group (OMG) and defined the Interface Definition Language (IDL) and the Application Programming Interfaces (API) that enable client/server object interaction within a specific implementation of an Object Request Broker (ORB). CORBA 2.0, adopted in December of 1994, defines true interoperability by specifying how ORBs from different vendors can interoperate.

Associated OMG Documents

The CORBA documentation is organized as follows:

- *Object Management Architecture Guide* defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.
- *CORBA: Common Object Request Broker Architecture and Specification* contains the architecture and specifications for the Object Request Broker.
- *CORBAservices: Common Object Services Specification* contains specifications for OMG's Object Services.

The OMG collects information for each specification by issuing Requests for Information, Requests for Proposals, and Requests for Comment and, with its membership, evaluating the responses. Specifications are adopted as standards only when representatives of the OMG membership accept them as such by vote. (The policies and procedures of the OMG are described in detail in the *Object Management Architecture Guide*.)

OMG formal documents are available from our web site in PostScript and PDF format. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

OMG Headquarters
250 First Avenue, Suite 201
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
pubs@omg.org
http://www.omg.org

Acknowledgments

The following companies submitted parts of this specification:

- Alcatel Alshtom Recherche
- DSET Corporation
- Expersoft Corporation
- Hewlett-Packard Company
- Highlander Communications, L.C.
- Inprise Corporation
- International Business Machines Corp.
- IONA Technologies, Plc
- ISR Global Telecom, Inc.
- Lucent Technologies, Inc.

- Nortel Technology
- Sun Microsystems
- Telefónica Investigación y Desarrollo S.A. Unipersonal
- TCSI Corporation

Specification Description 1

Contents

This chapter contains the following sections.

1.1 JIDM Overview

Note – JIDM (Joint Inter-Domain Management)

Ideally, all the CORBA Facilities and interfaces required to support interworking with different management environments would be defined in a generic way (i.e., independent of the Systems Management Reference Model being considered - OSI Systems Management Reference Model, SNMP Reference Model, etc). However, there are aspects related to each specific model that cannot be abstracted away. For example, the naming schema used to name managed objects will depend on the specific model being considered (e.g., a specific naming schema has been defined for OSI Systems Management). Also, interfaces used to operate on collections of managed objects will depend on the model being considered since expressions used to designate such

collections will vary depending on the model (e.g., use of scoping and filtering expressions to designate subset of members of a managed object domain has been specifically defined for OSI Systems Management).

The adopted approach consists of defining a basic set of CORBA Facilities, referred to as JIDM Facilities, that will work for every Systems Management Reference Model. JIDM Facilities can be extended or put together with additional complementary facilities to build up the set of CORBA Facilities that will be finally used to implement each of the specific Systems Management Reference Models. Thus, for example, when defining OSI Systems Management Facilities, specific OSI Management Facilities will be defined (facilities that allow translation between OSI names and **CosNaming::Names**, handling operations with scoping and filtering, etc.) in addition to those defined within JIDM Facilities (facilities defined to get references to single managed objects given their names, etc).

1.2 Definitions and Design Principles

1.2.1 Reference Model

To enable interworking between management systems based on different technologies, it is necessary to be able to map between the relevant object models and to build on this to provide mechanisms to handle protocol and behavior conversions on the domain boundaries.

In order to be able to interwork between a particular pair of management reference models, there are two aspects that need to be defined:

- A translation scheme between the different object models of both management reference models, referred to as *Specification Translation*
- A dynamic conversion mechanism between the protocols and behaviors used in both domains, referred to as *Interaction Translation*

This allows objects in one domain to be represented in the other domain and the interactions can be governed by the domain of choice rather than by the domain in which the target object is implemented. Besides, this should be done without either party being aware of the conversion.

This document presents a set of facilities to provide interoperability between CORBA and alternative telecommunication management models, specifically OSI management and Internet management. As described above, two aspects need to be defined: Specification Translation and Interaction Translation.

1.2.2 Specification Translation

The translation scheme is not part of this document, it has already been adopted and published by other standardization organisms, namely NMF and The OpenGroup, with the following reference “[XoJIDM] Inter Domain Management: Specification Translation” mentioned in Appendix A.

Inter-domain Management: Specification Translation

X/Open Document Number: P509

ISBN: 1-85912-150-0

This specification fully supports the above mentioned JIDM Specification Translation specification, amended with the current list of errata and corrigenda to the document, as expressed in the *"JIDM Specification Translation Issues List"* (available from The OpenGroup and NMF web sites, and also from the OMG as document number telecom/98-05-05). The justification for these changes is also available through the aforementioned amending documentation.

1.2.3 Interaction Translation

This document presents a set of CORBA facilities required to support interworking with different management environments, globally referred to as *"JIDM Interaction Translation."*

There are three levels of interfaces being defined:

1. *Generic interfaces, management model independent* - these facilities provide a generic framework to access a managed domain, independently of the management reference model being used. These generic facilities are referred to as *JIDM Facilities*, and are presented in Chapter 3.

2. *Generic interfaces, management model dependent* - two management reference models are considered, OSI Management and Internet Management (SNMP).
   - *OSI Management Facilities*, presented in Chapter 4, provide a CORBA view of the OSI Management reference model, as described in the relevant ITU-T and ISO documents (see, among others, ["[X720] Management Information Model."] and ["[M3010] Principles for a Telecommunications Management Network." mentioned in Appendix A]). This set of facilities extends the generic JIDM Facilities to support all CMIS interactions in CORBA, and to support OSI specific concepts such us scoping, filtering and multiple replies both in pure CORBA environments and in interworking environments (gateways).
   - *SNMP Management Facilities*, presented in Chapter 5, provide a CORBA view of the Internet Management reference model. These sets of facilities also extend the generic JIDM Facilities to support all SNMP interactions in CORBA, and to support Internet specific concepts.

3. *Specific interfaces, information model, and management model dependent* - these interfaces provide functionality that is specific to a given information model, that conforms to a certain management reference model; these interfaces reuse and extend the generic CORBA facilities of the corresponding management reference model (OSI Management or SNMP Management facilities) in an information model-specific way. In case the specific information model is specified in a foreign specification language (GDMO/ASN.1 for OSI management, SNMP SMI for Internet management), the equivalent CORBA IDL model may be automatically generated by following the translation algorithms defined in "Inter-domain Management: Specification Translation" (see reference to "[XoJIDM] Inter Domain

Management: Specification Translation, mentioned in Appendix A). Note that it is possible to specify an information model directly using CORBA IDL, and yet reuse the OSI management or SNMP management facilities.

It is beyond the scope of this specification to specify any information model specific interfaces. However, the mechanisms to specify such interfaces, as well as a generic set of algorithms to translate existing information models, are specified.

There is a dependency among these three types of interfaces, as shown in Figure 1-1.

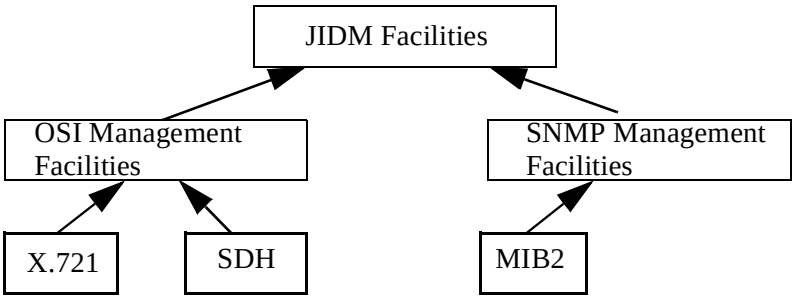

Figure 1-1 JIDM Facilities

1.3 Basic Concepts

Throughout this document, a number of well-known concepts are used and maybe even overused. However, there are certain concepts where the intent when using a certain word is very specific. This section tries to clarify the special meanings attributed to certain words/concepts within this document.

A distributed management system is composed of two kinds of entities: manager entities and managed entities.

"Manager entities" are those that have responsibility for one or more management activities, by issuing management operations and receiving notifications. They are the components exploiting the behavior provided by implementations of a given information model.

"Managed entities" are those that have responsibility for certain underlying resource(s). They perform management operations issued by manager entities on the underlying resources, and emit notifications whenever some specific circumstances occur. They are the components implementing the behavior of a given information model.

In object oriented systems, these abstract entities are materialized in the form of specific objects. Therefore the terms *"manager object"* and *"managed object"* can be considered synonymous of the above in object oriented systems.

Manager objects (entities) are said to act in the "*manager role,*" while managed objects (entities) are said to act in the "*agent role.*"

These objects (entities) are grouped into "*domains*" according to some specific grouping criteria. Domains are considered the unit of accessibility, therefore being the independently addressable components within a distributed system; each domain (both manager and managed) may have any number of objects within it.

Managed domains are sometimes referred to as "*agents*" and "*managed object domains*," while manager domains are sometimes referred to as "*manager applications*" or simply "*managers*."

Domains are identified by using "*titles*." Each domain may have an arbitrary number of titles associated with it, but a title uniquely identifies one domain.

Whenever a manager or an agent needs to interact with an agent or manager (respectively), it must first "*gain access*" to the other domain. This access is always granted through a specific"*port*" to the domain. Each port is uniquely identified by one of the titles associated to the domain being accessed.

Specifically, two types of ports are identified:

1. When access to a domain is required to be able to create and/or invoke operations on managed objects within the domain, the port is called "*domain port*."
2. When access to a domain is required to be able to forward events to manager objects within the domain, the port is called "*event port*."

When a manager (agent) gains access to a managed object domain (manager domain), it is said that a "*session*" has been established. That session may be "*released*," meaning that no further exchange of information may happen, because access has been "*revoked*."

Any number of sessions may exist between a manager and an agent at any given time.

1.4 Problem Statement

1.4.1 Invoking Operations on Managed Objects

CORBA Facilities need to be defined that allow CORBA manager objects to connect to Managed Object Domains given their titles. Additional CORBA Facilities need to be defined to allow a CORBA manager object (that is connected to some given domain of managed objects) to:

- Create a new member of the domain (a new managed object) and assign a name to it.
- Obtain a reference to a member of the domain (an existing managed object) given its name.
- Operate on collections of those members of the domain which meet some criteria (e.g., descendants of some managed object which pass some specific filter, etc.).

A complete solution requires explaining how these CORBA Facilities will interact with Naming and LifeCycle Object Services at CORBA Managed Object Domains. These questions are represented in Figure 1-2 on page 1-6.

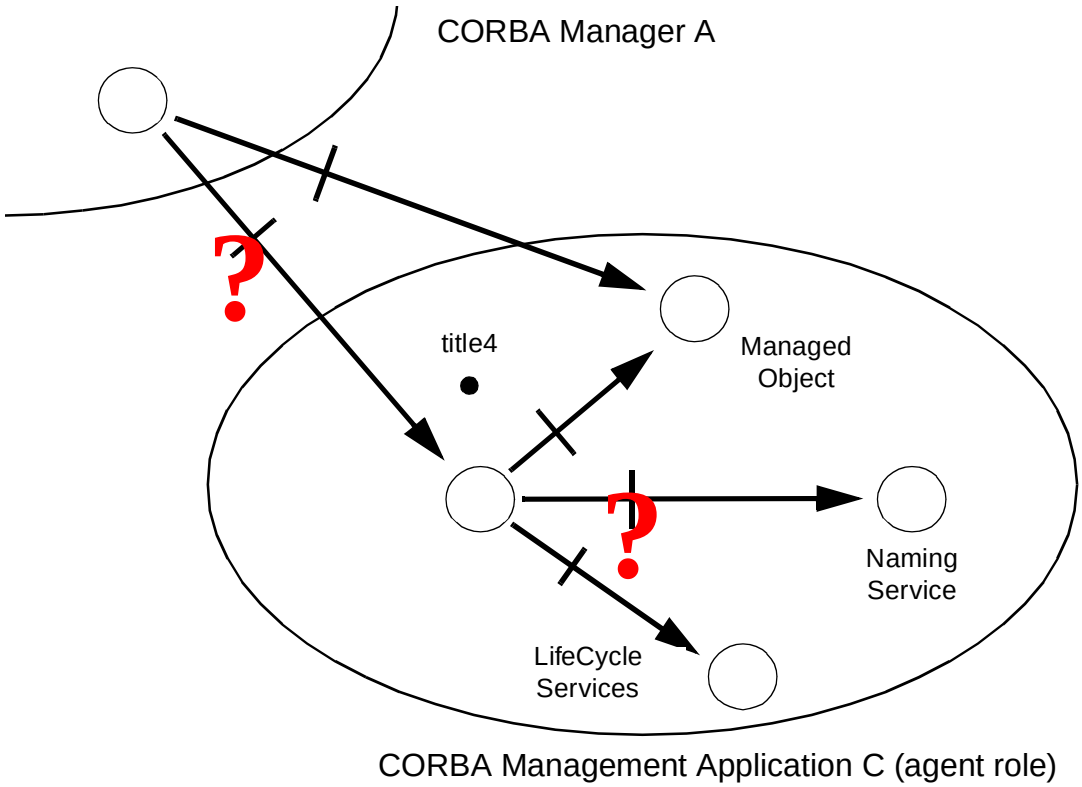

Figure 1-2 Invocation of management operations

A fundamental requirement for definition of such CORBA Facilities is that the specific management protocol (CMIP, SNMP, CORBA IIOP, etc.) being used to get access to a Managed Object Domain and operate upon managed objects located there, must be totally transparent to CORBA manager objects and CORBA managed objects.

1.4.2 Event Reporting

A CORBA Manager will have at least one title associated with it. This title permits it to be identified as a destination for event reporting. CORBA Facilities need to be defined that allow:

- Event reports emitted by CORBA managed objects to be reported to specific CORBA Managers that have been designated by their titles.
- CORBA Manager objects to be notified about events reported from remote Managed Object Domains.

A fundamental requirement for definition of such CORBA Facilities is that the specific management protocol (CMIP, SNMP, CORBA IIOP, etc.) being used to report an event must be totally transparent to CORBA manager objects and CORBA managed objects.

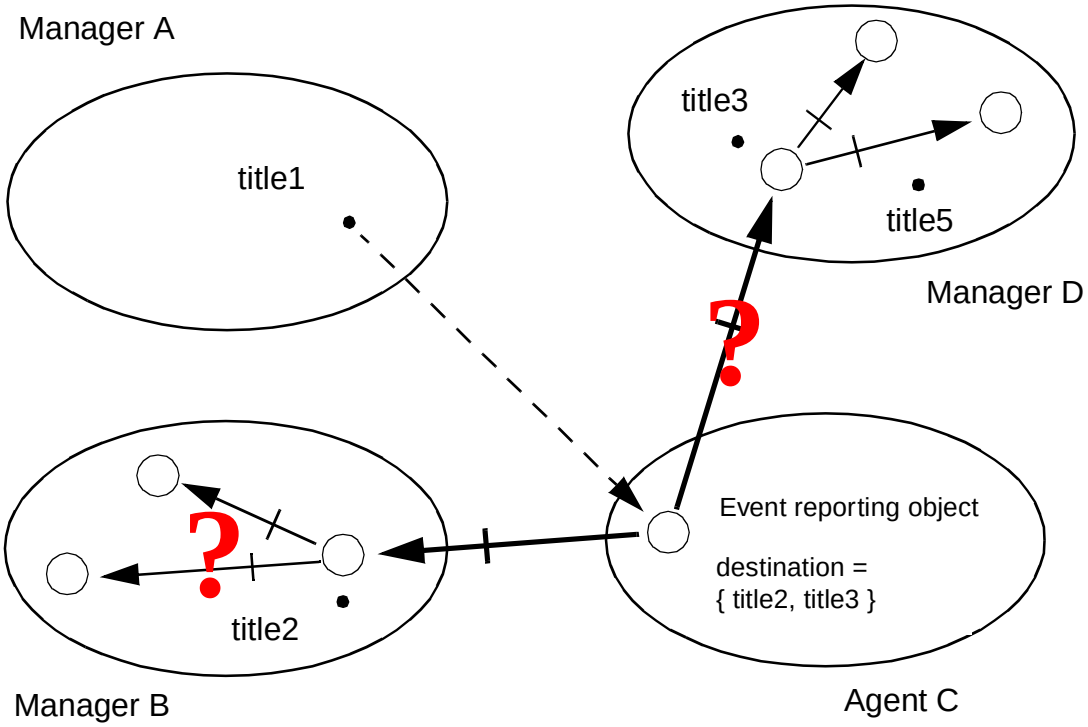

Figure 1-3 Event reporting

When solving the problem of event reporting, the following scenarios must be considered:

- One manager application must be able to change the list of destinations (titles) to which event reports emitted by managed objects in a managed object domain are being reported. In the OSI environment, this will be accomplished by means of changing the destination attribute value of an EFD object that is a member of the managed object domain being considered.
- One manager application may spontaneously start receiving event reports from a remote managed object domain due to a decision taken by a third party (another manager application) who has changed the list of destinations for event reports associated to the managed object domain.

1.5 General Design Principles

1.5.1 Key Design Principles

CORBA/TMN interworking is provided through a common framework (JIDM), which provides interfaces and facilities common to OSI systems management and Internet management. This common framework is then specialized to provide additional interfaces and facilities that are specific to each systems management reference model.

The proposal maximizes the commonality of services (e.g., creation of objects, invocation of operations, event reporting, and distribution) used for interworking scenarios and for pure CORBA environment scenarios.

Also, some specific guiding principles have been consistently applied when trying to resolve the issues encountered:

- *Completeness* - The aim is to provide as complete a set of services as possible, covering all possible cases and specific functionality, regardless of the frequency.
- *Simplicity* - There are certain scenarios and services that are more common than others. Given the completeness principle, all cases should be covered. However, the most common cases should be covered with the simplest approach, at the expense of potentially complicating the less common situations. This is also known as the 80-20 rule.
- *Familiarity* - The design must use concepts and patterns that are familiar to the CORBA programmer. In this way, managed objects must be plain CORBA objects that implement a certain interface and on which the operations exported by the supported interface may be invoked. Also, events and notifications sent from managed objects are plain CORBA events.
- *Transparency* - It should be transparent to the applications the fact that a gateway is being used or not. That is, an application should not be aware, or do anything differently, in case it interacts with another application that uses the same technology, or it does with an application that uses a different technology, using a gateway as an intermediary.
- *Reuse of OMG specifications and services* - Rather than inventing new approaches to do the same thing, already existing OMG specifications have been reused whenever possible.
- *Freedom of implementation* - This document does not impose any particular implementation policy, and does not constrain implementations in any way unless it is absolutely necessary. Although the discussions to arrive to any specific design solution always take into account the feasibility of implementations, the document tries not to provide any implementation bias.

1.5.2 Alignment with CORBA Design Principles

The design of CORBA/TMN interworking facilities:

- Uses and builds on CORBA concepts:
  - Separation of interface and implementation
  - Object references are typed by interfaces
  - Clients depend on interfaces, not implementations
  - Use of multiple inheritance of interfaces
  - Use of subtyping to extend, evolve and specialize functionality
  - Finding a service is orthogonal to using it
  - Factories, factory finders and use of federation of services or traders
- Assumes good ORB and Object Services implementations:

- Specifically, it is assumed that CORBA-compliant ORB implementations are being built that support efficient local and remote access to a very high number of objects and have performance characteristics that place no major barriers to the pervasive use of distributed objects for virtually all service and application elements.

- Allows Local and Remote Implementations:
  - In general the services are structured as CORBA objects with OMG IDL interfaces that can be accessed locally or remotely and which can have local library or remote server styles of implementations. This allows considerable flexibility with regards to the location of participating objects.
- Enforces interface style consistency:
  - Use of exceptions and return codes
  - Use of explicit operations
  - Use of interface inheritance

1.5.3 *Alignment with OSI Systems Management and Internet Management Design Principles*

Management of a communications environment is an information processing application. Because the environment being managed is distributed, the individual components of the management activities are themselves distributed.

Management applications perform the management activities in a distributed manner, by establishing associations between systems management application entities.

The interactions which take place between systems management application entities are abstracted in terms of management operations and notifications issued by one entity to the other; these are communicated using systems management services and protocols.

Management activities are effected through the manipulation of managed objects. For the purposes of systems management, management applications are categorized as MIS-Users. Each interaction takes place between two MIS-Users, one taking the manager role, the other the agent role.

An MIS-User taking the role of an agent is that part of a distributed application that manages the managed objects within its local system environment. An agent performs management operations on managed objects as a consequence of management operations communicated from a manager. An agent may also forward notifications emitted by managed objects to a manager.

An MIS-User taking the role of a manager is that part of a distributed application which has responsibility for one or more management activities, by issuing management operations and receiving notifications.

JIDM CORBA Facilities 2

Contents

This chapter contains the following sections.

2.1 The JIDM Module

The Joint Inter-Domain Management (JIDM) module comprises a collection of interfaces that together define a basic set of services for developing Systems Management Applications based on CORBA. Following the JIDM reference model these interfaces may be used between Manager applications and JIDM Frameworks, or between JIDM Frameworks and Agent applications.

From the Manager application perspective, the following interfaces are used:

- The **ProxyAgent** interface
- The **ProxyAgentController** interface
- The **ProxyAgentFinder** interface
- The **EventPort** interface
- The **EventPortFactory** interface

From the Agent application perspective, the following additional interfaces are used:

- The **DomainPort** interface

- The **DomainPortFactory** interface
- The **EventPortFinder** interface

This section describes these interfaces and their operations in detail.

```
#ifndef _JIDM_IDL_
#define _JIDM_IDL_

#include <CosNaming.idl>
#include <CosLifeCycle.idl>
#include <CosEventChannelAdmin.idl>

#pragma prefix “jidm.org”

module JIDM
{
    typedef CosNaming::Name Key;
    typedef CosLifeCycle::Criteria Criteria;

    exception InvalidKey {};
    exception InvalidCriteria {};
    exception CannotMeetCriteria { Criteria reason; };
    exception CannotAccess {};
    exception AlreadyExists {};

    interface ProxyAgent {
        enum DestructionMode {gracefully, non_gracefully};
        readonly attribute Criteria access_criteria;

        CosLifeCycle::FactoryFinder get_domain_factory_finder ();
        CosNaming::NamingContext get_domain_naming_context ();
        Criteria destroy (in DestructionMode mode, in Criteria the_criteria)
            raises (InvalidCriteria, CannotMeetCriteria);
    };

    interface ProxyAgentController {
        Criteria destruction_is_allowed (in Criteria the_criteria)
            raises (InvalidCriteria,CannotMeetCriteria);

        void destroyed (in Criteria the_criteria);
    };

    interface ProxyAgentFinder {
        ProxyAgent access_domain (in Key k, in Criteria the_criteria)
            raises (InvalidKey, CannotAccess, InvalidCriteria, CannotMeetCriteria);
    };

    interface DomainPort {
        readonly attribute Criteria associated_criteria;
            void destroy ();
    };

    interface DomainPortFactory {
        DomainPort create_domain_port (in Key k, in Criteria creation_criteria)
            raises (InvalidKey, InvalidCriteria, CannotMeetCriteria);
```

```
    };

    interface EventPort {
        readonly attribute CosEventChannelAdmin::SupplierAdmin supplier_admin;
        readonly attribute Criteria associated_criteria;
        void destroy ();
    };

    interface EventPortFactory {
        EventPort
            create_event_port (in Key k, in Criteria creation_criteria,
            in CosEventChannelAdmin::SupplierAdmin the_supplier_admin)
                raises (InvalidKey, InvalidCriteria, CannotMeetCriteria, AlreadyExists);
    };

    interface EventPortFinder {
        CosEventChannelAdmin::SupplierAdmin
            find_event_port (in Key k, in Criteria the_criteria)
                raises (InvalidKey, InvalidCriteria, CannotMeetCriteria, NoEventPort);

    };
};

#endif /* _JIDM_IDL_ */
```

2.1.1 JIDM Managed Objects

The JIDM module does not define an interface for generic JIDM objects, as it would be an empty interface, because there are no truly generic, common operations that could be attributed to all kinds of managed objects.

However, every management environment must define some kind of managed object, that is the entity being managed in the respective environment. For example, in OSI systems management, there is the concept of a managed object directly; in SNMP management, there is no concept of a managed object, but a concept of entries within the SNMP MIB that is equivalent to a managed object.

The definition of these managed objects, in all management environments, must support the design principles as outlined in the RFP. In particular, in support of the transparency principle, the following semantics are required of any managed object interface:

- If a CORBA object reference is used to request a managed object to perform an operation and the managed object does not exist, an OBJECT_NOT_EXIST exception should result.
- If a CORBA object reference is used to request a managed object to perform an operation and the request causes an OBJECT_NOT_EXIST exception, the managed object was actually deleted. (This property helps users avoid unnecessary failures while attempting to recreate an object that already exists.)
- If a CORBA object reference is used to request a managed object to perform an operation and the managed object exists, the operation shall not result in an OBJECT_NOT_EXIST exception.

2.1.2 The JIDM::ProxyAgent Interface

Managers that require creating and/or invoking operations on managed objects that are members of a domain must first gain access to that domain. When a manager gains access to a managed object domain, a **JIDM::ProxyAgent** object (an object that exports the **JIDM::ProxyAgent** interface) is created. Several **JIDM::ProxyAgents** may co-exist, giving parallel access to the same managed object domain.

```
interface ProxyAgent {
    enum DestructionMode {gracefully, non_gracefully};
    readonly attribute Criteria access_criteria;
```

```
interface ProxyAgent {
    enum DestructionMode {gracefully, non_gracefully};
    readonly attribute Criteria access_criteria;

    CosLifeCycle::FactoryFinder get_domain_factory_finder ();
    CosNaming::NamingContext get_domain_naming_context ();

    Criteria destroy (in DestructionMode mode, in Criteria the_criteria)
        raises (InvalidCriteria, CannotMeetCriteria);
};
```

```
interface ProxyAgent {
    enum DestructionMode {gracefully, non_gracefully};
    readonly attribute Criteria access_criteria;

    CosLifeCycle::FactoryFinder get_domain_factory_finder ();
    CosNaming::NamingContext get_domain_naming_context ();

    Criteria destroy (in DestructionMode mode, in Criteria the_criteria)
        raises (InvalidCriteria, CannotMeetCriteria);
};
```

Invoking operations exposed by the **JIDM::ProxyAgent** object, CORBA manager objects are able to obtain references to an initial:

- **CosLifeCycle::FactoryFinder** object in the managed object domain being accessed.
- **CosNaming::NamingContext** object in the managed object domain being accessed.

Invoking the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object, CORBA manager objects may find factories that enable creation of new members of the managed object domain.

Invoking the **resolve** operation exposed by the initial **CosNaming::NamingContext** object, CORBA manager objects may obtain CORBA object references to existing members of the managed object domain.

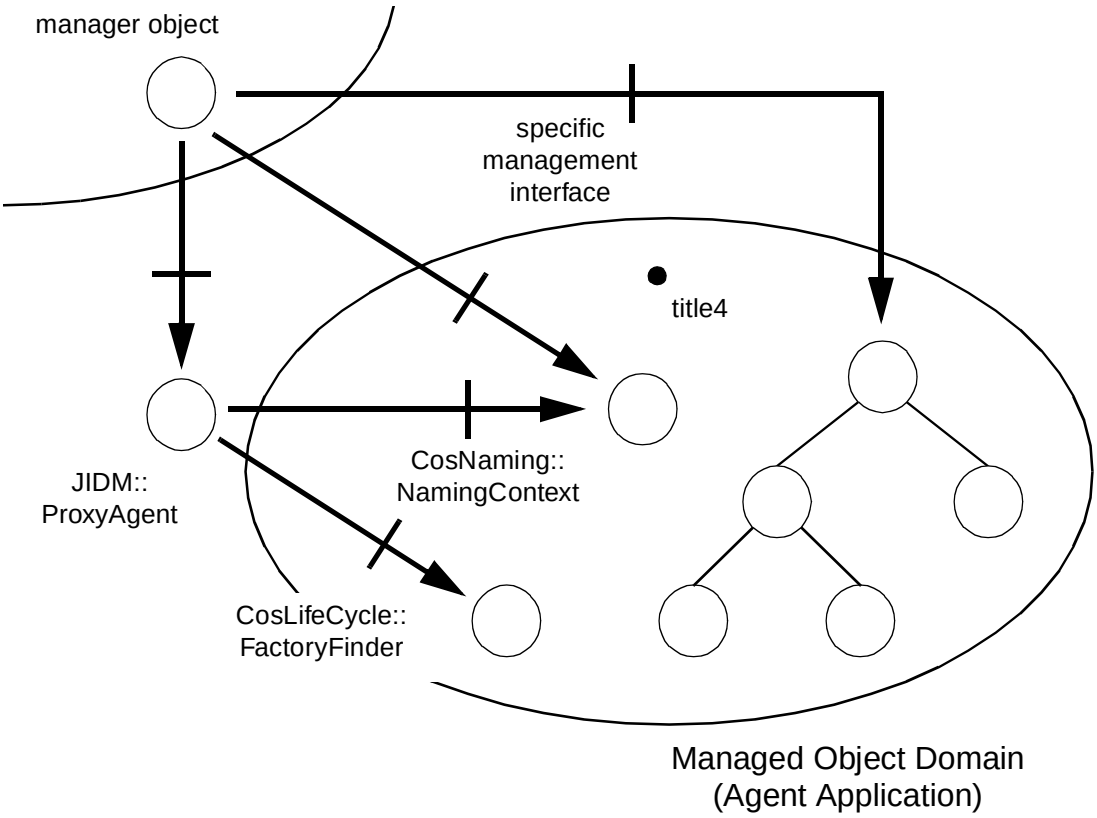

Figure 2-1 JIDM::ProxyAgents in a CORBA Environment

2.1.2.1 The get_domain_factory_finder Operation

To create a managed object, CORBA manager objects first need to find a reference to a suitable factory. They do so by means of invoking the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object in the managed object domain where the new managed object is going to be created.

The **get_domain_factory_finder** operation obtains a reference to this initial **CosLifeCycle::FactoryFinder** object in the domain being accessed through a given **JIDM::ProxyAgent** object.

```
module CosLifeCycle {
....
    typedef CosNaming::Name Key;
    typedef Object Factory;
    typedef sequence <factory> Factories;

    exception NoFactory {
        Key search_key;
    };

    interface FactoryFinder {
```

```
        Factories find_factories (in Key factory_key)
            raises (NoFactory);
    };
    ....
};
```

As shown above, the **find_factories** operation supported by **CosLifeCycle::FactoryFinders** returns a sequence of factories, which matches some given key. The space of keys is established by convention in particular environments as explicitly declared in CORBA (see the *Common Object Request Broker: Architecture and Specification, Interface Repository* chapter).

Conventions adopted for JIDM facilities (i.e., common to every Systems Management Reference Model) are described in Table 2-1. Additional conventions may exist for each specific Systems Management Reference Model being considered.

Table 2-1 : JIDM Conventions for Factory Finder Keys

id field	kind field	meaning
fully scoped name of object interface	“object interface”	Find factories that create objects supporting the named interface.
fully scoped name of factory interface	“factory interface”	Find factories supporting the named factory interface.

Several alternatives can be followed when assigning values to the keys that will be passed to the **find_factories** operation.

Only the name of the object interface is specified

Here, it is implicitly assumed that there is a factory interface associated to the managed object interface. CORBA managers know the name and operations associated with this factory in advance so they can properly narrow and use the reference returned by the **find_factories** operation.

Only the name of the object factory interface is specified

Here, references returned by the **find_factories** operation can be narrowed to the IDL interface whose name has been specified. The CORBA manager object who invoked the operation knows the signature and semantics of operations supported by the designated object factory interface. This option will be the one used to obtain references to generic factories exporting the **CosLifeCycle::GenericFactory** interface or any of the generic factory interfaces defined in SYSMANfacilities (see Appendix A, “References”).

Both the name of the object and factory interfaces are specified

This will be useful in environments where there are more than just one factory object interface associated with each managed object interface. Thus, the CORBA manager object specifies

- the interface that will be exported by the object to be created, and
- the actual interface of the factory that it wants to use for creating the new object (among the possible types of factories that can create objects exporting such interface).

In any case, the result of passing several key values should be interpreted as the logical 'and' of the conditions associated to each of the keys.

In respect to creation of managed objects, it is worth noticing that JIDM facilities provide a generic framework that requires it to be specialized for each specific Systems Management Reference Model. Such specialization implies precise definition of:

- The whole space of keys that are valid for finding factories.
- The space for keys and criteria that can be passed as arguments to the **create_object** operation exposed by **CosLifeCycle::GenericFactory** objects.
- Other types of interfaces that are better suited to the specific Systems Management Reference Model that is being considered (for example, the **OSIMgmt::ManagedObjectFactory** interface or interfaces generated from name-binding GDMO templates in the OSI Systems Management Reference Model).

2.1.2.2 *The get_domain_naming_context operation*

A CORBA manager object may obtain CORBA object references to members of a managed object domain as a result of invoking the **resolve** operation exposed by the initial **CosNaming::NamingContext** object in the domain. The **resolve** operation may also be used to obtain references to **CosNaming::NamingContext** objects other than the initial **CosNaming::NamingContext** object.

The **get_domain_naming_context** operation obtains a reference from the initial **CosNaming::NamingContext** object in the domain being accessed through a given **JIDM::ProxyAgent** object.

The design imparts no semantics or interpretation of the names themselves. Because the structure of names must be common to all Systems Management Reference Models, the structure of names defined in the standard CosNaming Service specification has been adopted. The actual semantics or interpretation of names is up to the specific Systems Management Reference Model being used. CORBA facilities defined for a given reference model will typically include definition of library interfaces enabling construction of names in CORBA Naming Service form (**CosNaming::Names**).

Table 2-2 describes the exceptions raised by the **resolve** operation. Note that this description complies with the description given for the standard CosNaming service in the *Naming Service* specification.

Table 2-2 Exceptions Raised by the Resolve Operation

Exception Raised	Description
NotFound	Indicates the name does not identify a binding (there is no object reference bound to the name passed as argument).
CannotProceed	Indicates that implementation of the resolve operation has given up for some reason. The client, however, may be able to continue the operation using the returned name and reference, which points to a **CosNaming::NamingContext**.
InvalidName	Indicates the name is invalid (a name of length 0 is invalid; additional restrictions apply depending on the specific management support environment).

Only the **resolve** operation is guaranteed to be available in any management environment.

CORBA manager objects may not have access to the rest of operations (**bind**, **unbind**, etc.) exposed by the initial **CosNaming::NamingContext** object or any of its subordinate **CosNaming::NamingContext** objects. If a CORBA manager object invokes an operation it cannot access, a NO_PERMISSION exception is raised.

2.1.2.3 The access_criteria attribute

Any **JIDM::ProxyAgent** object exposes the **access_criteria** attribute, which checks the terms and conditions under which access through the **JIDM::ProxyAgent** object was accepted.

This attribute is represented as a **Criteria**, and its contents depend on the specific System Management Reference model being used.

2.1.2.4 The destroy operation

Any **JIDM::ProxyAgent** object exposes the **destroy** operation, which destroys the object.

Destroying a **JIDM::ProxyAgent** object means closing the session established with the associated managed object domain. If the **JIDM::ProxyAgent** object was running in a JIDM gateway server, destruction of the object implies disposing resources used to maintain the associated connection (closing an XMP descriptor, for example).

Destruction of a **JIDM::ProxyAgent** object can take place in one of the following modes:

- **gracefully**, meaning that resources associated to the session are going to be disposed of in a graceful manner. If this is not possible, a CannotDestroy exception is raised.

- **non_gracefully**, meaning that resources associated with the session are going to be disposed of in an abrupt manner. No user exception is expected in this case.

Graceful destruction of **JIDM::ProxyAgent** objects should always be requested in the first place. If graceful destruction is not possible, a client may request non-graceful destruction to destroy the object.

Invokers of the **destroy** operation may pass a criteria that will be analyzed to determine whether the destruction request can be accepted or not (see Section 2.1.3, "The JIDM::ProxyAgentController Interface," on page 2-9 for more details).

- If any of the destruction criteria are not understood, the InvalidCriteria exception is raised.
- If the destruction request is not accepted, the CannotMeetCriteria exception is raised. The criteria describing reasons for the rejection is provided with the exception.
- If destruction is accepted, the **destroy** operation returns a **Criteria** value that typically describes the terms and conditions under which destruction has been accepted. This criteria is initialized with values provided by controller objects in the manager and managed object domains (see Section 2.1.3, "The JIDM::ProxyAgentController Interface," on page 2-9 for more details).

Once a **JIDM::ProxyAgent** object is destroyed, references to it are no longer valid. Therefore, invoking an operation on a **JIDM::ProxyAgent** object that has been destroyed causes the standard OBJECT_NOT_EXIST exception to be raised. In addition, invoking operations using references to managed objects, factories, factory finders, and naming contexts that were obtained through the destroyed **JIDM::ProxyAgent** object causes the standard INV_OBJREF exception to be raised.

2.1.3 The JIDM::ProxyAgentController Interface

Destruction of **JIDM::ProxyAgent** objects in a distributed environment requires definition of mechanisms that validate whether destruction is allowed or not. This specification introduces validation through **JIDM::ProxyAgentController** objects.

```
interface ProxyAgentController {
    Criteria destruction_is_allowed (in Criteria the_criteria)
                raises (InvalidCriteria, CannotMeetCriteria);
    void destroyed (in Criteria the_criteria);
};
```

A **JIDM::ProxyAgentController** object may be associated with a **JIDM::ProxyAgent** object, by passing its reference in the criteria parameter when invoking the **access_domain** operation of the **JIDM::ProxyAgentFinder**. In this case, the **JIDM::ProxyAgentController** plays the role "associated at the manager side." Following the exact matching rule of key and criteria, a maximum of one **JIDM::ProxyAgentController** object may be associated with a given **JIDM::ProxyAgent** object at the manager side.

One or more **JIDM::ProxyAgentController** objects may be associated with a **JIDM:ProxyAgent**, at the managed object domain. This can be done if a **JIDM::DomainPort** object is initialized with a list of **JIDM::ProxyAgentController** objects (see Section 2.1.5, “The JIDM::DomainPort Interface,” on page 2-13).

A **destroy** operation invoked in a **JIDM::ProxyAgent** object will invoke operation(s), depending on the destruction mode, in all **JIDM::ProxyAgentController** objects associated with such **JIDM::ProxyAgent**s.

2.1.3.1 The destruction_is_allowed Operation

The **destruction_is_allowed** operation is invoked to validate whether a graceful destruction of a **JIDM::ProxyAgent** object may occur. Therefore, this operation is only invoked by the **ProxyAgent** when **destroy** has been called, with a **DestructionMode** of ”**gracefully**.” If there are more than one **JIDM::ProxyAgentController** objects associated to a **JIDM::ProxyAgent**, they should all be consulted. Destruction of the **JIDM::ProxyAgent** object is only permitted if all **JIDM::ProxyAgentController** objects accept destruction of the object.

The criteria passed as argument to the **destroy** operation is passed to the **JIDM::ProxyAgentController** objects as a parameter of the **destruction_is_allowed** operation.

- If destruction is allowed, the **destruction_is_allowed** call returns a Criteria potentially specifying the terms and conditions under which destruction is allowed. If the Criteria is empty, destruction is allowed unconditionally.
- If destruction is prohibited, the CannotMeetCriteria exception should be raised. This exception carries the reason why permission to destroy was not granted.

  When an exception is raised by any **JIDM::ProxyAgentController** object in response to this call, this exception is automatically propagated as the result of the **destroy** call that triggered this process.
- If all **JIDM::ProxyAgentController** objects associated with the **JIDM::ProxyAgent** being destroyed allow the destruction, then the **JIDM::ProxyAgent** object is effectively destroyed. The Criteria returned by the **destroy** call is the result of combining all Criteria returned by all involved **JIDM::ProxyAgentController** objects.

  When combining several criteria into one, the shared components are copied (once) into the combined criteria, and those that are present in one but not another are also copied into the combined criteria. That is, the combination is a “union” of criteria components. The actual number and type of values in the criteria will typically depend on the reference model being considered.

2.1.3.2 The destroyed operation

The **JIDM::ProxyAgentController destroyed** operation is invoked after the **JIDM::ProxyAgent** has been effectively destroyed (at this point, the **JIDM::ProxyAgent** object no longer exists). This may occur as a result of a successful graceful destruction interaction, as described above, or as a result of a non-graceful destruction request.

This call carries the Criteria passed to the **destroy** call in case of ungraceful destruction, or the combined results of the corresponding **destruction_is_allowed** calls, as described above.

If there is more than one **JIDM::ProxyAgentController** object associated with a **JIDM::ProxyAgent**, all will be notified.

2.1.4 The JIDM::ProxyAgentFinder Interface

Objects exporting the **JIDM::ProxyAgentFinder** interface provides an access to managed object domains. CORBA managers that require access to a managed object domain invoke the **access_domain** operation exposed by a **JIDM::ProxyAgentFinder** object.

```
interface ProxyAgentFinder {
    ProxyAgent access_domain (in Key k, in Criteria the_criteria)
        raises (InvalidKey, CannotAccess, InvalidCriteria, CannotMeetCriteria);
};
```

A **JIDM::ProxyAgent** object represents a session established with a managed object domain. Each session is unequivocally characterized by:

- a key that typically identifies the specific Systems Management Environment being considered (e.g., OSI environment, SNMP environment). This key enables adequately interpreting the criteria value passed as second argument to the operation, and
- a criteria value that contains, among other things, the title associated with the domain being accessed (an AE-title in OSI environments, an IP-address, or hostname in SNMP environments).

This means that invoking the **access_domain** operation with two different <key , criteria> pairs will create two different **JIDM::ProxyAgent** objects.

2.1.4.1 The access_domain operation

Two scenarios may occur when invoking the **access_domain** operation:

1. No **JIDM::ProxyAgent** object exists with the same key and criteria values passed in the invocation. In this case, a new **JIDM::ProxyAgent** object is created with the key and criteria values associated with it. Finally, a reference to the new **JIDM::ProxyAgent** object is returned.

2. A **JIDM::ProxyAgent** object exists with the same key and criteria values passed in the invocation. In this case, a reference to the already existing **JIDM::ProxyAgent** object is returned.

It is worth noticing that there can be multiple sessions established with a managed object domain. This means that any managed object may be accessed through multiple **JIDM::ProxyAgents**.

Conventions adopted for keys in JIDM are described in Table 2-3. Standard key values include “OSI Management” and “Internet Management” denoting the OSI and SNMP-based Systems Management Environments.

Table 2-3 JIDM Conventions for Proxy Agent Finding Keys

id field	kind field	meaning
“OSI Management” “Internet Management”	“XSM environment”	Find proxy agents for the specific Systems Management Environment.

The criteria passed as second argument to the **access_domain** operation will contain information needed to set up the requested session. Only the title assigned to the domain being accessed has been identified as required for all Systems Management Reference Models. Wildcard titles are supported in some specific management environments, enabling designation of the whole space (domain) of managed objects.

If the manager requires exercising control upon destruction of the **ProxyAgent**, a reference to a **JIDM::ProxyAgentController** object must also be specified in the Criteria (see section Section 2.1.3, “The JIDM::ProxyAgentController Interface,” on page 2-9).

Table 2-4 summarizes the name and meaning of criteria that may be passed as input argument to the **access_domain** operation exposed by **JIDM::ProxyAgent** objects.

Table 2-4 JIDM Conventions for Proxy Agent Finding Criteria

criterion name	type	meaning
“domain title”	domain specific type	Title associated to the managed object domain for which access is required.
“controller object”	JIDM::ProxyAgentController	reference associated to a JIDM::ProxyAgentController object in the manager (OPTIONAL).

The actual number and type of values in the criteria will depend typically on the reference model being considered.

A reference to a root **JIDM::ProxyAgentFinder** object at the manager may be obtained by invoking the **resolve_initial_references** operation exposed through the standard **CORBA::ORB** interface. The standard **CORBA::ObjectId** assigned to the root **JIDM::ProxyAgentFinder** object would be "**JIDM::ProxyAgentFinder**."

Following is a fragment of code in a CORBA Manager program:

```
CORBA::ORB_ptr my_orb;
Object_ptr obj;
JIDM::ProxyAgentFinder_ptr agent_finder;
JIDM::ProxyAgent_ptr agent;
JIDM::Key a_key;
JIDM::Criteria a_criteria;

// A reference to a local JIDM::ProxyAgentFinder object is
// obtained using standard ORB initialization services:

my_orb = ORB_init (argv, my_ORB_id);
obj = my_orb->resolve_initial_references ("JIDM::ProxyAgentFinder");
agent_finder = ProxyAgentFinder::_narrow (obj);
.....

// After assigning proper values to key and criteria arguments ...
a_key = ...;
a_criteria = ...;

// an association to the managed object domain can be established:
agent = agent_finder -> access_domain (a_key, a_criteria);
```

Other possibilities include (but are not limited to) registration of the root **JIDM::ProxyAgentFinder** object in the local initial **CosNaming::NamingContext** or a local Trader.

2.1.5 The JIDM::DomainPort Interface

Managers that require creating and/or invoking operations on managed objects that are members of a managed object domain must first gain access to that domain. Access to a managed object domain is controlled by **JIDM::DomainPort** objects. Each **JIDM::DomainPort** object has a title associated with it.

```
interface DomainPort {
    readonly attribute Criteria associated_criteria;
            void destroy ();
};
```

All **JIDM::DomainPort** objects associated with a managed object domain (i.e., associated with titles used to refer to that managed object domain) hold references to initial **CosNaming::NamingContext** and **CosLifeCycle::FactoryFinder** objects in the domain.

In pure CORBA environments, when a manager object invokes the operation **access_domain** on a **ProxyAgentFinder** object, the request will finally arrive at the **JIDM::DomainPort** object associated with the title passed as one of the parameters in the criteria.

Two scenarios may occur.

1. A **JIDM::ProxyAgent** object exists that exactly matches the key and criteria values passed as arguments to the **access_domain** operation. In this case, the **JIDM::DomainPort** object finds a reference to the **JIDM::ProxyAgent** object and returns it to the invoker.

2. There is no **JIDM::ProxyAgent** object matching the key and criteria values passed as arguments to the **access_domain** operation. In this case, the **JIDM::DomainPort** object creates a new **JIDM::ProxyAgent** object and returns the reference to the invoker. References to the initial **CosNaming::NamingContext** and **CosLifeCycle::FactoryFinder** objects in the domain will be passed to the new **JIDM::ProxyAgent** object by the **JIDM::DomainPort** object.

During its lifetime, a **JIDM::DomainPort** object will keep the references to all **JIDM::ProxyAgent** objects it creates. This is necessary to resolve the first scenario.

2.1.6 The JIDM::DomainPortFactory Interface

The **JIDM::DomainPort** objects can be created dynamically by means of invoking the **create_domain_port** operation exposed by **JIDM::DomainPortFactory** objects.

```
interface DomainPortFactory {
    DomainPort create_domain_port (in Key k, in Criteria creation_criteria)
        raises (InvalidKey, InvalidCriteria, CannotMeetCriteria);
};
```

The key passed as first parameter to this call follows the same conventions that were specified in Table 2-3 on page 2-12.

The criteria passed as second argument to the **create_domain_port** operation will contain information needed to create the **JIDM::DomainPort** object. Only the title assigned to the domain being accessed has been identified as required for all Systems Management Reference Models. Other criteria values may be considered as default values for all the **JIDM::ProxyAgent** objects that are created by a domain port.

If the managed domain requires exercising control upon destruction of the **JIDM::ProxyAgent** objects, a reference to one or more **JIDM::ProxyAgentController** objects must also be specified in the Criteria.

Table 2-5 JIDM Conventions for create_domain_port Criteria

criterion name	meaning
“domain title”	Title associated to the managed object domain for which access is required.

criterion name	meaning
“controller object”	Reference associated to JIDM::ProxyAgentController object(s) in the managed domain. (OPTIONAL).

2.1.7 The JIDM::EventPort Interface

Managed objects that require forwarding events to managers must first gain access to the manager. Access to a manager is gained through **JIDM::EventPort** objects. Each **JIDM::EventPort** object has a title associated with it.

```
interface EventPort {
    readonly attribute CosEventChannelAdmin::SupplierAdmin supplier_admin;
    readonly attribute Criteria associated_criteria;
        void destroy ();
};
```

A **JIDM::EventPort** object models the port through which events are going to be received by a manager application. This port is created according to a certain criteria which, among other things, contain the title that identifies the manager domain (see Section 2.1.8, “The JIDM::EventPortFactory Interface,” on page 2-15). Any agent/managed object that wants to send events to a manager must set up a connection with the **CosEventChannelAdmin::SupplierAdmin** object associated with the **JIDM::EventPort** object with the appropriate title.

Therefore, a **JIDM::EventPort** provides and handles access to a specific **CosEventChannelAdmin::SupplierAdmin** object in a channel (the **SupplierAdmin** object associated with the **EventPort**). The same **CosEventChannelAdmin::SupplierAdmin** object may be associated with different **JIDM::EventPorts** and accessed through several **JIDM::EventPorts**.

Also note that any number of agents/managed objects may establish sessions with the same **CosEventChannelAdmin::SupplierAdmin** associated with a given **JIDM::EventPort**.

The **JIDM::EventPorts** may be destroyed by invoking the **destroy** operation they expose.

2.1.8 The JIDM::EventPortFactory Interface

The **JIDM::EventPort** objects can be created dynamically by invoking the **create_event_port** operation exposed by **JIDM::EventPortFactory** objects.

```
interface EventPortFactory {
    EventPort create_event_port (in Key k, in Criteria creation_criteria,
        in CosEventChannelAdmin::SupplierAdmin the_supplier_admin)
            raises (InvalidKey, InvalidCriteria, CannotMeetCriteria, AlreadyExists);
};
```

Any manager wanting to receive events should obtain an object reference to a **CosEventChannelAdmin::SupplierAdmin** of the channel that is to receive the events, and create a **JIDM::EventPort** object with the desired criteria. The **JIDM::EventPort** object receives

- a key, identifying the Systems Management Reference model to be used (with the same values specified in Table 2-3 on page 2-12),
- a criteria containing at least the title of the manager, and
- a reference to the **CosEventChannelAdmin::SupplierAdmin** object.

Only the title of the manager application has been identified as required by all System Management Reference models. The criteria may contain other fields associated with the specific management environment being used.

Table 2-6 JIDM Conventions for create_event_port Criteria

criterion name	meaning
"domain title"	Title associated to the manager domain that wants to receive events.

Note – The manager may create a new channel (or admin) or use an object reference to an existing one.

A manager needs to create an **EventPort** just once (independent of the agents that are or will be interested in receiving events). This enables a manager to receive event reports from one or more agents through a specific title.

In addition, several manager objects could share the same **SupplierAdmin** or **EventChannel** (**NotificationChannel**) object, and connect themselves as consumers interested in receiving just some kind of events.

If the creation is successful, a reference to the newly created **JIDM::EventPort** object is returned.

In case of problems, the appropriate exception is raised:

- InvalidKey in case the Key is not recognized.
- InvalidCriteria if any of the components of the Criteria is not understood.
- CannotMeetCriteria if the conditions for creating the **EventPort** cannot be met.
- AlreadyExists in case there is already an existing **EventPort** registered with matching Key/Criteria.

2.1.9 The JIDM::EventPortFinder Interface

CORBA Managed objects that are members of a CORBA-based managed object domain can obtain references to **JIDM::EventPort** objects by invoking operations exposed by a **JIDM::EventPortFinder** object..

```
interface EventPortFinder {
    exception NoEventPort {};

CosEventChannelAdmin::SupplierAdmin
    find_event_port (in Key k, in Criteria the_criteria)
        raises (InvalidKey, InvalidCriteria, CannotMeetCriteria, NoEventPort);
};
```

Connection to the **CosEventChannelAdmin::SupplierAdmin** object associated with a **JIDM::EventPort** is gained by invoking the operation **find_event_port** exposed by a **JIDM::EventPortFinder** object in the agent. As a result of that invocation, a reference to the **CosEventChannelAdmin::SupplierAdmin** object that matches the corresponding Key and Criteria is returned. Managed objects may establish specific connections to this **CosEventChannelAdmin::SupplierAdmin** object by using operations exposed by the **SupplierAdmin** interface.

It is worth noticing that the **JIDM::EventPortFinder** object returns a reference to the **CosEventChannelAdmin::SupplierAdmin** objects associated with the **JIDM::EventPort** object, and not a reference to the **JIDM::EventPort** itself.

Essentially, invoking the **find_event_port** operation exposed by a **JIDM::EventPortFinder** object implies that the following steps are followed:

1. The **JIDM::EventPortFinder** object finds a reference to the **JIDM::EventPort** object associated with a key and criteria.
2. The **JIDM::EventPortFinder** finds the **CosEventChannelAdmin::SupplierAdmin** object associated with this **JIDM::EventPort**, and a reference to this object is returned to the CORBA managed object that invoked the **find_event_port** operation.

In case the request fails, the appropriate exception is raised:

- InvalidKey in case the Key is not recognized.
- InvalidCriteria if any of the components of the Criteria are not understood.
- CannotMeetCriteria if the conditions for finding the **EventPort** cannot be met.
- NoEventPort in case there is no **EventPort** registered with the appropriate Key/Criteria.

Different strategies to resolve how CORBA managed objects finally report events can be implemented, including but not limited to:

1. CORBA managed objects directly register themselves as **PushSuppliers** or **PullSuppliers** through the **SupplierAdmin** associated to the **JIDM::EventPort**.

2. CORBA managed objects register themselves as **PushSuppliers** or **PullSuppliers** in a single object that acts as some kind of **EventChannel** (or **NotificationChannel**), which in turn is registered as a **PushSupplier** or **PullSupplier** as described above. This is particularly useful when there is more than one **JIDM::EventPort** object and the CORBA managed objects do not need to be aware of the specific port to which events must be sent.

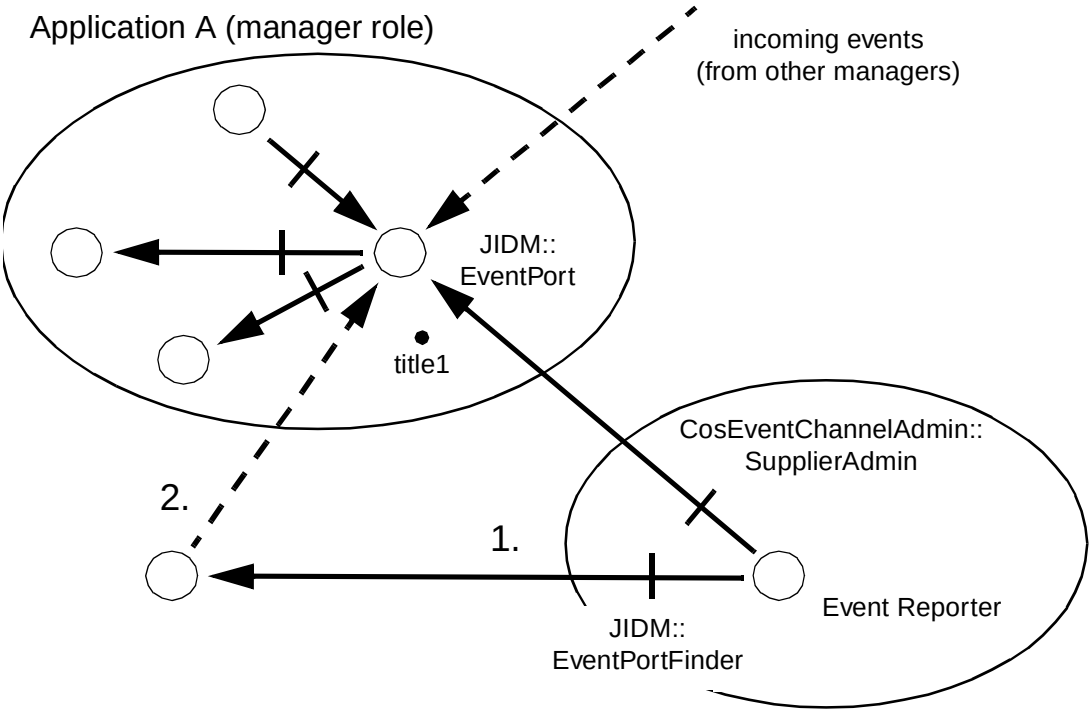

Figure 2-2 Finding References to JIDM::EventPort Objects

2.2 Programming Model

This section is provided as information only, and does not represent a normative part of the specification. Different scenarios are described where the use of this specification will be clarified. This should be considered as a high-level tutorial on some potential uses of the JIDM model. Also, some potential implementation options are discussed.

2.2.1 Programming Semantics

CORBA manager programs create and invoke operations on managed objects in the same way they create and invoke operations on ordinary CORBA objects located in the same CORBA domain. Analogously, they receive events supplied by managed objects as if they were ordinary CORBA objects supplying events to an event channel located in the CORBA domain. Whether this actually happens or not is transparent to the CORBA manager program.

This concept of transparency is specifically supported by the fulfillment of the semantic rules presented in Section 2.1.1, “JIDM Managed Objects,” on page 2-3.

2.2.2 Creating Managed Objects

Creating a managed object implies performing the following actions:

1. Obtain a reference to a **JIDM::ProxyAgent** object that enables access to the domain where the managed object is going to be created.
2. Obtain a reference to the initial **CosLifeCycle::FactoryFinder** in the domain.
3. Invoke the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object to find a factory for the new managed object.
4. Select a factory from the several factory objects that may meet the keys for finding factories passed to the **find_factories** operation.
5. Invoke an appropriate operation, exposed by the selected factory, to create the managed object.

The CORBA manager will narrow references returned by the **find_factories** operation to get visibility of the specific interface exported by each factory. These scenarios are possible:

- There is a specific factory interface associated with each managed object interface.
- Factories export a well-know generic interface like the **CosLifeCycle::GenericFactory** interface.

CORBA Managers should know which of these two scenarios is implied when the name of factory object interface is not passed as key to the **find_factories** operation..

```
module CosLifeCycle {
...
    typedef struct NVP {
        CosNaming::Istring name;
        any value;
    } NameValuePair;

    typedef sequence <NameValuePair> Criteria;
    typedef CosNaming::Name Key;

    interface GenericFactory {
        boolean supports (in Key k);
        Object create_object (in Key k, in Criteria the_criteria)
                    raises (NoFactory, InvalidCriteria,
                            CannotMeetCriteria);
    };
};
```

Typically, a name for the managed object is passed as argument to the **create** operation. This name would unequivocally identify the managed object within the domain where it will be created. The identifier of the principal interface exported by the managed object should also be passed to the **create** operation in case it was not passed as a key to the **find_factories** operation.

As already mentioned in Section 2.1.2, “The JIDM::ProxyAgent Interface,” on page 2-4, valid key values for finding factories depend on the specific Systems Management Reference Model being considered. However, the overall philosophy is common to all models.

The following example shows the code used to create a managed printer. This fragment of code would be the same for all reference models you consider (e.g., it would be the same for OSI management or SNMP-based management). Only the way in which the reference to the **JIDM::ProxyAgent** object is obtained, the keys used to find factories, or semantics of arguments passed to the **create_object** operation may vary.

```
JIDM::ProxyAgent_ptr agent;
CORBA::Object_ptr obj;
Printing::ManagedPrinter_ptr my_printer;
JIDM::Key finding_key (1);
CosLifeCycle::FactoryFinder_ptr ff;
CosLifeCycle::Factories mo_factories;
CosLifeCycle::GenericFactory_ptr my_factory;
.....

// a reference to a JIDM::ProxyAgent is obtained as a result of
// establishing a session with the managed object domain where
// the printer is going to be created:

agent = ...;
.....

// a reference to the initial CosLifeCycle::FactoryFinder object
// is obtained:

ff = agent -> get_domain_factory_finder ();

// a key to find a factory for the managed object is constructed:

finding_key [0].id = “CosLifeCycle::GenericFactory”;
finding_key [0].kind = “factory interface”;
.....

// factories for the managed object are found through the initial
// factory finder:

mo_factories = ff -> find_factories (finding_key);

// a reference is selected and narrowed to the expected interface:

my_factory = CosLifeCycle::GenericFactory::_narrow (mo_factories [0])

// a managed printer is created and a reference to it is returned
```

```
// which will typically be narrowed to a specific interface

obj = my_factory -> create_object (...);
my_printer = Printing::ManagedPrinter::_narrow (obj);
```

As shown in the example, CORBA managers will create managed objects (a managed printer in the example) in the same way they create CORBA objects in the local ORB domain. That is to say, by means of using standard **CosLifeCycle** services. Whether this actually happens or not depends on how the corresponding managed object domain is being accessed:

- through one or several interconnected ORBs (i.e., directly through CORBA), or
- through a JIDM gateway (a CORBA/CMIP gateway, for example).

Figure 2-3 illustrates how CORBA manager objects will create a new managed object in a pure CORBA environment.

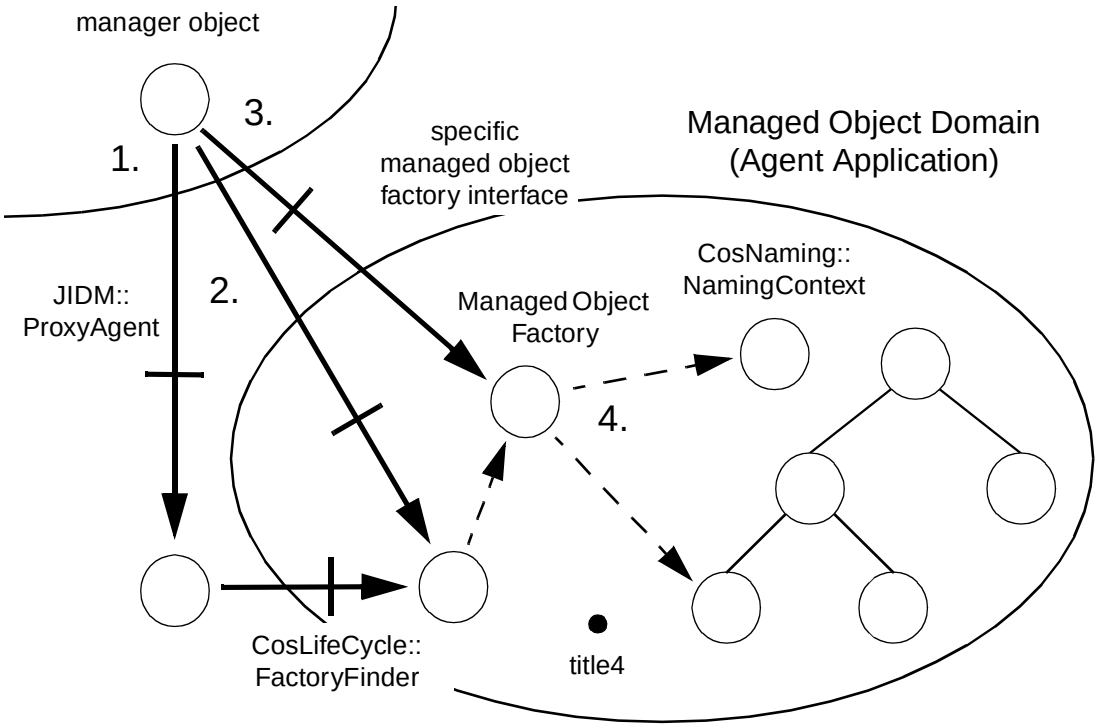

Figure 2-3 Invoking operations on a managed object directly through CORBA

The **JIDM::ProxyAgent** created as a result of establishing a session with a CORBA managed object domain would typically hold references to the root **CosNaming::NamingContext** object and **CosLifeCycle::FactoryFinder** object in the domain. These steps are followed:

1. The CORBA manager invokes the **get_domain_factory_finder** operation exposed by the **JIDM::ProxyAgent** object. As a result, a reference to the initial **CosLifeCycle::FactoryFinder** in the domain being accessed is returned.

2. The CORBA manager object invokes the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object. As a result, a reference to a managed object factory is obtained and returned to the CORBA manager object that requested it.

3. The CORBA manager object invokes a suitable operation on the managed object factory using the CORBA object reference previously obtained. Typically, the CORBA manager will narrow this reference to a well-known managed object factory interface (the **CosLifeCycle::GenericFactory** interface, for example).

4. The managed object factory creates the CORBA managed object and performs any other required action (such as registering a reference to the managed object with a name in the local **CosNaming::NamingContext** and/or notifying that a new managed object has been created to other objects at the managed object domain). This kind of side-effect actions may vary depending on the Systems Management Reference Model being considered.

5. Finally, if everything is all right, the managed object factory returns a reference to the CORBA manager object; otherwise, it returns an exception.

2.2.3 Invoking Operations on Managed Objects

Invoking an operation on a managed object implies performing the following actions:

1. Obtain a reference to a **JIDM::ProxyAgent** object that enables access to some domain of which the managed object is a member.

2. Obtain a reference to the initial **CosNaming::NamingContext** in the domain, by means of invoking the **get_domain_naming_context** operation exposed by the **JIDM::ProxyAgent** object.

3. Construct the name that unequivocally identifies the managed object within the domain.

4. Invoke the **resolve** operation exposed by the initial **CosNaming::NamingContext** object in the domain, thus obtaining a CORBA object reference pointing to the managed object.

5. Invoke the operation on the managed object.

Of course, steps 1 through 4 are only strictly required the first time a managed object is accessed. Actually, the CORBA manager object that obtains a reference to a managed object can register the reference in some local object service so that other CORBA manager objects can find the reference and do not need to interact with **JIDM::ProxyAgent** or **CosNaming::NamingContext** objects.

Once a reference is obtained for a managed object, it is valid as long as the managed object exists and the associated managed object domain is accessible. A valid object reference can be used as many times as required. Two alternatives exist for invoking an operation on a managed object once a CORBA object reference is obtained for the object:

- Use the Dynamic Invocation Interface (DII); or

- Use IDL stubs generated from definition, in OMG IDL, of interfaces exported by the managed object.

In the first case, the CORBA object reference obtained as a result of resolving the name of the managed object can be used directly. In the second case, the CORBA object reference must be narrowed to a specific interface exported by the managed object.

The following example shows the code used to invoke the **reset** operation exposed by a managed printer (i.e., objects exporting the **Printing::ManagedPrinter** interface) using IDL stubs generated in C++.

```
JIDM::ProxyAgent_ptr agent;
CosNaming::Name printer_name;
.....

        // a reference to a JIDM::ProxyAgent is obtained as a result of
        // establishing a session with the managed object domain where
        // the printer is located:

agent = ...;
.....

// a reference to the initial CosNaming::NamingContext object
// is obtained:

CosNaming::NamingContext_ptr ctx = agent -> get_domain_naming_context ();
.....

        // the name of the printer is constructed:

printer_name = ...;

        // a reference to the managed printer is obtained and
        // narrowed to the ManagedPrinter interface:

CORBA::Object_ptr obj = ctx -> resolve (printer_name);
Printing::ManagedPrinter_ptr my_printer =
                        Printing::ManagedPrinter::_narrow (obj);

        // Finally, the reset operation is invoked on the managed object:

my_printer -> reset ();
```

It must be pointed out that this fragment of code would be the same for all Systems Management Reference Models that can be considered (e.g., it would be the same for OSI management or SNMP-based management). Only the way in which the reference to the **JIDM::ProxyAgent** object is obtained or the way in which the name of the managed object is constructed may vary. Thus, CORBA managers will have the illusion that managed objects (a managed printer in the example) are implemented as CORBA objects directly accessible via CORBA.

Whether this actually happens or not depends on how the corresponding managed object domain is being accessed:

- through one or several interconnected ORBs (i.e., directly through CORBA), or
- through a JIDM gateway (a CORBA/CMIP gateway, for example).

Figure 2-4 illustrates how CORBA manager objects invoke operations on a managed object in a pure CORBA environment.

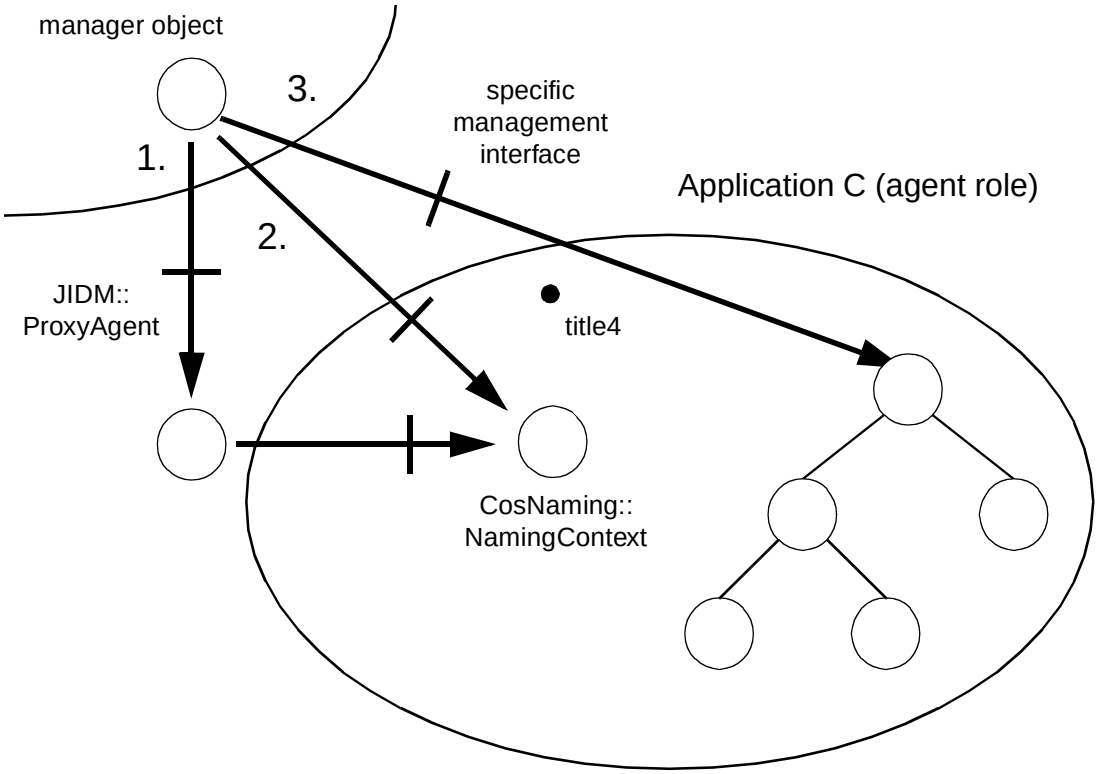

Figure 2-4 Invoking Operations on a Managed Object Directly through CORBA

As previously explained, the **JIDM::ProxyAgent** created as a result of establishing a session with a CORBA managed object domain would typically hold references to the initial **CosNaming::NamingContext** object and **CosLifeCycle::FactoryFinder** objects in the domain. Thus, the following steps will be followed:

1. The CORBA manager object invokes the **get_domain_naming_context** operation exposed by the **JIDM::ProxyAgent** object, in order to obtain a reference to the initial **CosNaming::NamingContext** object.

2. The CORBA manager object invokes the **resolve** operation exposed by the initial **CosNaming::NamingContext** object, passing the name of the managed object upon which it wants to operate. As a result, a CORBA object reference to the managed object is obtained and returned to the CORBA manager object that requested it.

3. The CORBA manager object invokes an operation on the managed object using the CORBA object reference previously obtained. IDL stubs or the standard DII can be used for doing this. If IDL stubs are used, the CORBA manager object must narrow the reference to a specific OMG IDL interface.

2.2.4 Reception of Events at CORBA Managers

Different strategies to resolve how CORBA manager objects finally consume events can be implemented, including but not limited to:

- CORBA manager objects responsible for performing management functions directly register themselves as **PushConsumers** or **PullConsumers** at every local **JIDM::EventPort**.
- CORBA manager objects responsible for performing management functions register themselves as **PushConsumers** or **PullConsumers** in a single **EventChannel**. The event channel registers itself as a **PushConsumer** or **PullConsumer** in every local **JIDM::EventPort**. This is particularly useful when there is more than one **JIDM::EventPort** object and the CORBA manager objects do not need to distinguish through which specific port events were received.

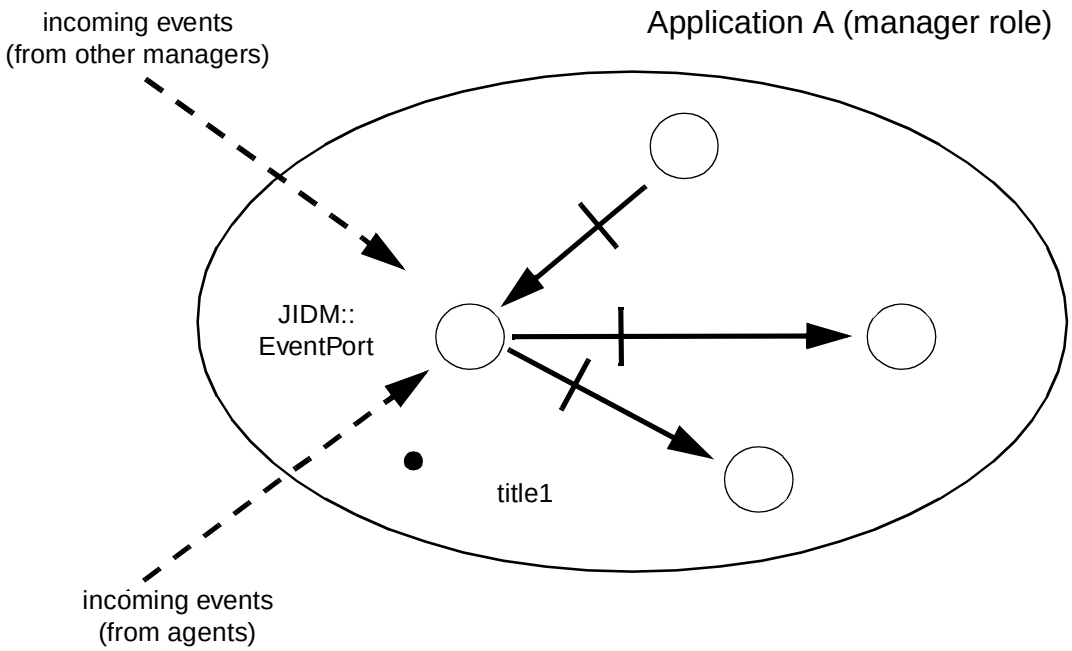

Figure 2-5 Event Reception at CORBA Managers

It is worth noticing that several advantages are derived from defining **JIDM::EventPort** objects as having **CosEventChannelAdmin::SupplierAdmin** objects, some of which are:

- Any **EventChannel** implementation supplied by any software provider can be used to receive events from an agent.

- Managers wanting to filter events can register a **NotificationChannel** instead of an **EventChannel** to filter events. A manager who does not want to filter events can use a simple **EventChannel**. This can be achieved transparently for gateway/managed domains.
- The same **EventChannel** can be shared by several managers and a manager can reuse the **EventChannel** whenever he wants, independently of it being used by other managers.
- Given the new **NotificationChannel** structure, multiple Admins per channel are possible (even frequent). By registering the Admin rather than the Channel itself, the user has better configuration possibilities.
- Ability to globally filter on reception.
- Easier grouping capabilities into a single channel, but keeping separate Admins per title (with potentially different filters).
- Potentially support delegation facilities (an agent/manager application could get a **SupplierAdmin** from a higher level manager and register it with another title, effectively saving a forwarding step).
- An **EventChannel** can be created in the gateway process (there is an **EventChannelFactory** facility being provided) or it can be created in any other distributed process and registered in the gateway (i.e., **EventChannel** can be distributed and the load of managing the final CORBA events can be balanced).
- CORBA manager objects consume events generated by remote managed objects in the same way they receive other events (events generated by other CORBA manager objects at the same CORBA Manager, for example).
- CORBA manager objects residing in different CORBA Managers can use **JIDM::EventPorts** to exchange events between them since they are ordinary event channels.
- Multiple scenarios for handling incoming events are possible at CORBA managers. One or several **JIDM::EventPort** objects can be installed, reception of events can be handled by means of applying different cascading techniques, push and pull styles of event communication can be combined, etc.

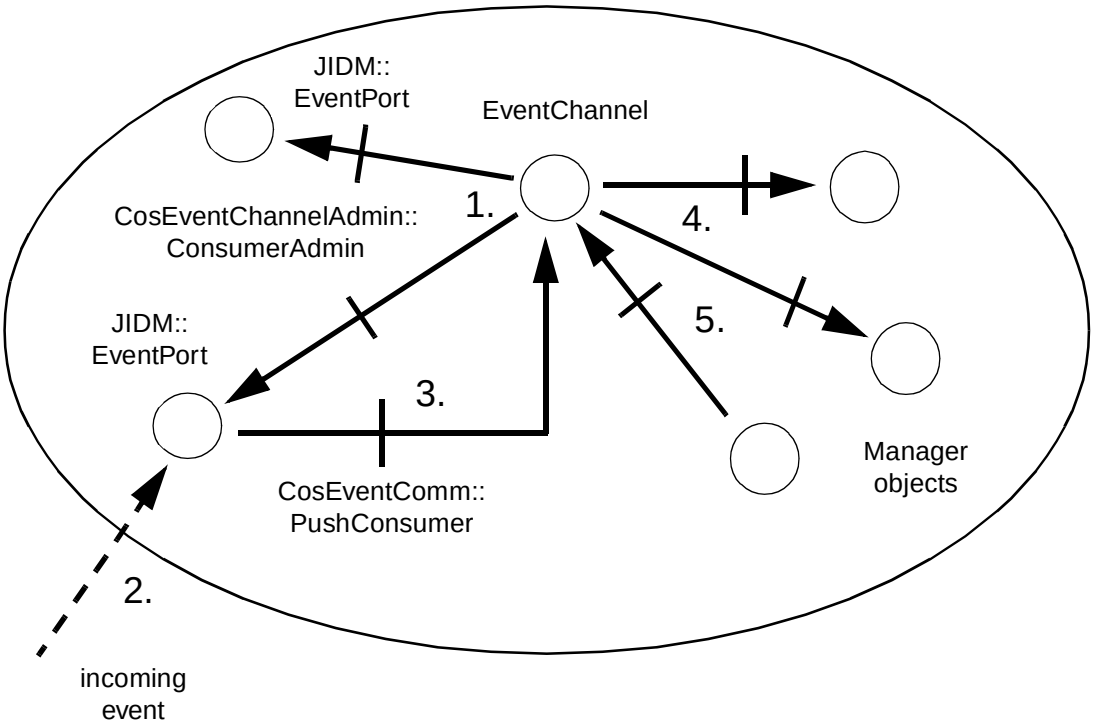

Figure 2-6 Handling Event Reports with Event Channels (push model)

Figure 2-6 represents one possible scenario - CORBA manager objects performing management functions are registered as consumers at an event channel which, in turn, has been registered as a **PushConsumer** at two local **JIDM::EventPort** objects. The basic algorithm being used is defined as follows:

1. During the start up phase of the CORBA Manager Application, a **CosEventChannelAdmin::EventChannel** object is registered as a **CosEventComm::PushConsumer** in every local **JIDM::EventPort**. CORBA manager objects actually performing the management functions are registered as consumers in this channel.

2. A **JIDM::EventPort** object receives data associated with an event.

3. The **JIDM::EventPort** invokes the **push** operation exposed by all objects that have been registered as **CosEventComm::PushConsumers**. This includes the **CosEventChannelAdmin::EventChannel** object. Data of the event is passed in the invocation as an any.

4. When the **CosEventChannelAdmin::EventChannel** object receives an event, it invokes the **push** operation exposed by all CORBA manager objects that were registered as **CosEventComm::PushConsumers** in the channel.

5. The **CosEventChannelAdmin::EventChannel** object keeps every event it receives until all CORBA manager objects that were registered as **CosEventComm::PullConsumers** invoke the pull operation on the channel or a time-out expires.

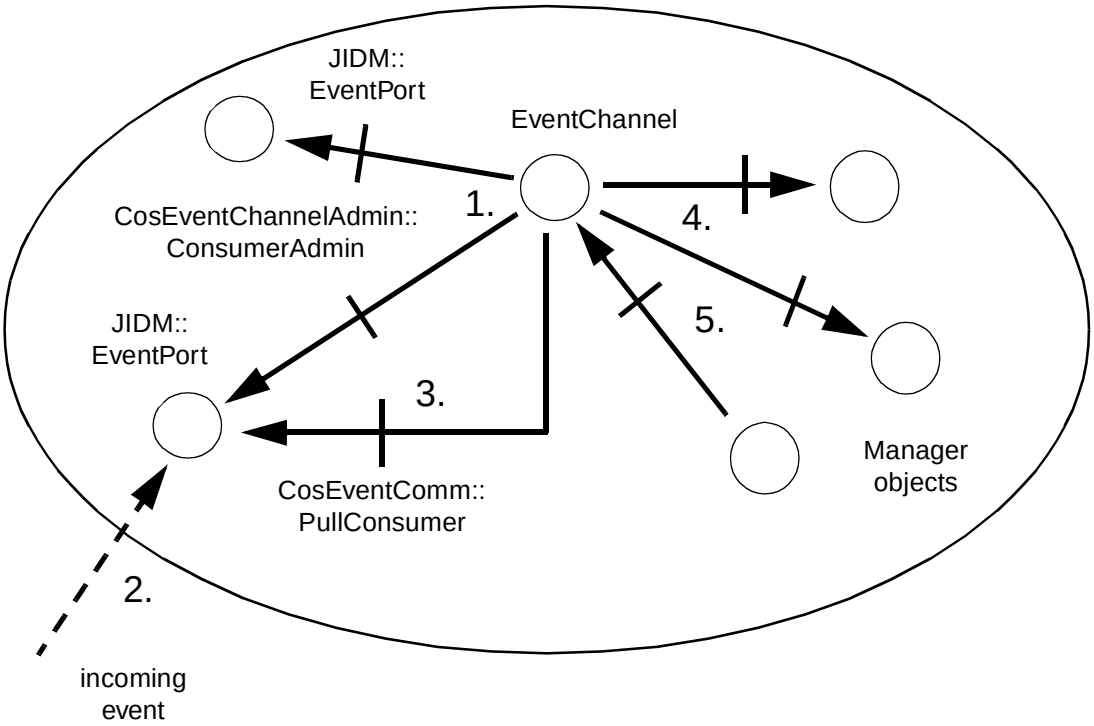

Figure 2-7 Handling Event Reports with Event Channels (pull model)

Figure 2-7 represents another possible scenario, where CORBA manager objects performing management functions are registered as consumers at one event channel that, in turn, has been registered as a **PullConsumer** at two local **JIDM::EventPort** objects. The basic algorithm being followed is:

1. During the start up phase of the CORBA Manager Application, the **CosEventChannelAdmin::EventChannel** object is registered as a **CosEventComm::PullConsumer** in every local **JIDM::EventPort**.

2. A **JIDM::EventPort** object receives data associated with an event report.

3. The **JIDM::EventPort** object holds the event so that all objects registered as **CosEventComm::PullConsumers** can consume the event. They will do it by invoking the **pull** operation exposed by the **JIDM::EventPort** object.

4. When the **CosEventChannelAdmin::EventChannel** pulls the event, it invokes the **push** operation exposed by all CORBA manager objects that have been registered as **CosEventComm::PushConsumers** in the channel.

5. The **CosEventChannelAdmin::EventChannel** object also keeps every event it pulls until all CORBA manager objects that have been registered as **CosEventComm::PullConsumers** invoke the **pull** operation on it or a time-out has expired.

2.2.5 Federation of JIDM::ProxyAgentFinders and JIDM::DomainPorts

To ensure that the service for finding **JIDM::ProxyAgent** objects is scalable, the principle of federation needs to be adopted. Federation is essential in large-scale distributed systems where the existence of a centralized ownership control cannot be assumed.

The specific service used to federate **JIDM::DomainPort** objects in a pure CORBA environment is transparent to CORBA manager clients and beyond the scope of this specification. Use of Traders or intermediate **JIDM::ProxyAgentFinder** objects connected in a graph are some examples of valid solutions. Furthermore, different federation services can be supported and combined to implement a complete solution.

Note that in this respect, the root **JIDM::ProxyAgentFinder** object that is accessible to CORBA managers simply represents a simple bootstrapping mechanism that encapsulates access to whatever Federation Service is finally used. By standardizing this interface, portability of CORBA manager clients is guaranteed. On the other hand, the **JIDM::ProxyAgentFinder** interface allows easier implementation of gateways between CORBA managers and managed object domains that are only accessible through standard management protocols (CMIP, SNMP, etc).

In a pure CORBA environment, a complete solution may be implemented based on a graph of inter-connected **JIDM::ProxyAgentFinder** objects. In such solution, there would be at least two styles of **JIDM::ProxyAgentFinder** objects:

1. The **JIDM::DomainPort** objects that actually work as factories of **JIDM::ProxyAgent** objects.

2. Intermediary **JIDM::ProxyAgentFinder** objects that pass requests on to either **JIDM::DomainPorts** or other intermediary **JIDM::ProxyAgentFinder** objects.

By configuring intermediary **JIDM::ProxyAgentFinder** objects and **JIDM::DomainPort** objects into a graph, the service for finding **JIDM::ProxyAgent** objects can be built so that it administers access to a large number of managed object domains.

Whenever a manager object invokes the **access_domain** operation, the request would traverse the graph until it reaches a **JIDM::DomainPort** object, which can satisfy the request (one that resides in the managed object domain being accessed). As the request traverses the graph, each intermediary (non-terminal) **JIDM::ProxyAgentFinder** object would decide which link the request would traverse next. Decisions are based upon information about each available link, any policies in force at that node, and value of parameters in the request.

Clearly, configuration of **JIDM::ProxyAgentFinder** graphs and definition of policies to traverse these graphs requires definition of Federating interfaces (see the Life Cycle Service under the CORBAservices heading for definition of interfaces to federate **CosLifeCycle::GenericFactory** objects). Again, different federation policies can be supported and combined to implement a complete solution. One possible alternative may be that intermediate **JIDM::ProxyAgentFinder** objects export an interface that enables them to bind a reference to a **JIDM::ProxyAgentFinder** object together with a specific filter that can be applicable to key and criteria values passed to the **access_domain** operation. This would allow registering one **JIDM::ProxyAgentFinder** object that is able to find references to **JIDM::ProxyAgent** associated with some range of title values.

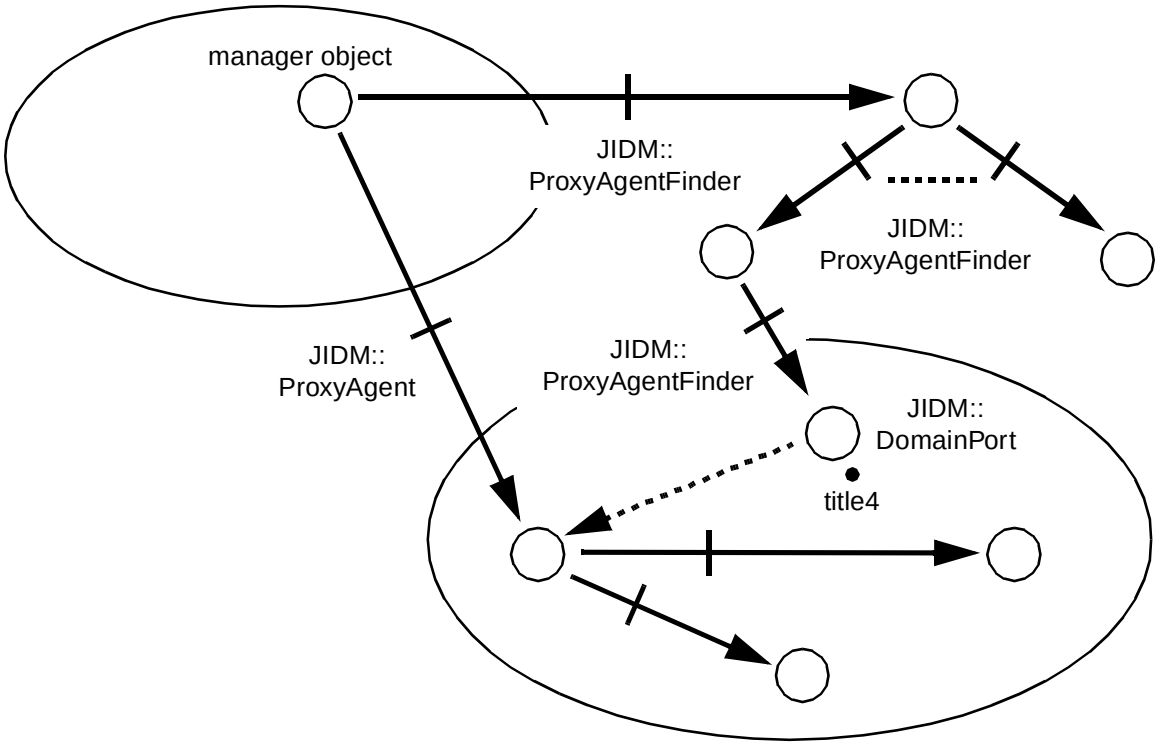

Figure 2-8 Implementing Federation with Graphs of ProxyAgentFinders

Figure 2-8 illustrates how federation of **JIDM::DomainPorts** would work if a graph of intermediate **JIDM::ProxyAgentFinder** objects is implemented. The following steps would be followed:

1. A manager object invokes the **access_domain** operation exposed by the root **JIDM::ProxyAgentFinder** object. The manager object typically obtains a reference to this **JIDM::ProxyAgentFinder** object using local initialization services.

2. The **JIDM::ProxyAgentFinder** object that first receives the **access_domain** request will typically act as an intermediary **JIDM::ProxyAgentFinder** object. Based on information available to that object, it will decide to pass requests on to other **JIDM::ProxyAgentFinder** objects.

3. The request will traverse the graph until it reaches a **JIDM::DomainPort** object, which resides at the managed object domain being accessed. That object will create or return an already existing reference to a **JIDM::ProxyAgent** object, which matches the key and criteria value passed in the request.

4. As a result of this process, a valid reference to a **JIDM::ProxyAgent** object would be passed to the manager object that requested to establish a session. Using this reference, the manager object is able to operate upon members of the managed object domain or create new members of the managed object domain.

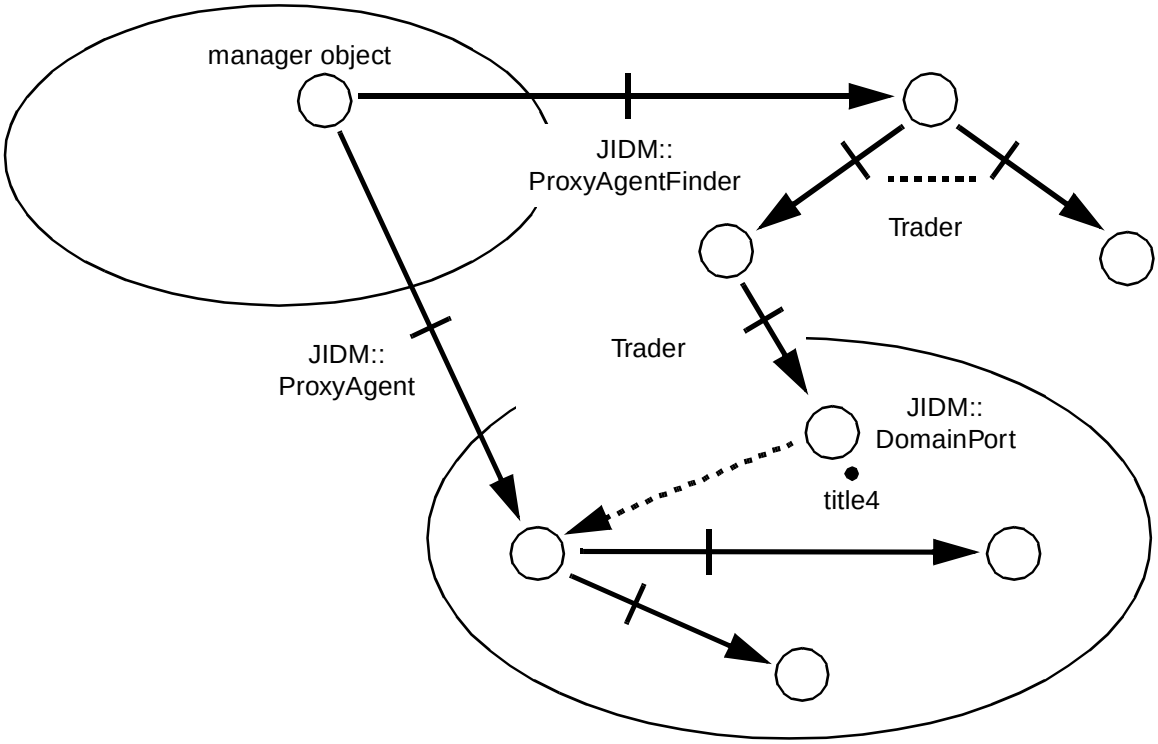

Figure 2-9 Implementing Federation with Traders

An alternative solution may be based on the use of Traders. Figure 2-9 illustrates how federation of **JIDM::DomainPorts** would work in such a case. The following steps would be followed:

1. A manager object would invoke the **access_domain** operation exposed by the root **JIDM::ProxyAgentFinder** object. The manager object would typically obtain a reference to the root **JIDM::ProxyAgentFinder** object using local initialization services.

2. The root **JIDM::ProxyAgentFinder** object would convert the request into an invocation of the **lookup** operation exposed by a **Trader** object. This **Trader** object may be federated with other **Trader** objects.

3. The request would traverse the graph of Traders until some of them find a **JIDM::DomainPort** object, which resides at the managed object domain being accessed. A reference to this **JIDM::DomainPort** object is passed to the root **JIDM::ProxyAgentFinder** object, which initiated the process.

4. The root **JIDM::ProxyAgentFinder** object would invoke **access_domain** on the **JIDM::DomainPort** object. As a result, a reference to a **JIDM::ProxyAgent** object is returned.

5. The returned reference would finally be passed to the manager object that requested to establish a session. Using this reference, the manager object is able to operate upon members of the managed object domain or create new members of the managed object domain.

2.2.6 Federation of JIDM::EventPortFinders and JIDM::EventPorts

To ensure that the service for finding **JIDM::EventPort** objects is scalable, the principle of federation also needs to be adopted. Federation is essential in large-scale distributed systems where the existence of a centralized ownership control cannot be assumed.

The specific service used to federate **JIDM::EventPort** objects in a pure CORBA environment is transparent to CORBA manager clients and is beyond the scope of this specification. Use of Traders or intermediate **JIDM::EventPortFinder** objects connected in a graph are some examples of valid solutions. Furthermore, different federation services can be supported and combined to implement a complete solution.

Note that in this respect, the root **JIDM::EventPortFinder** object that is accessible to CORBA managed objects simply represents a simple bootstrapping mechanism that encapsulates access to whatever Federation Service is finally used. By means of standardizing this interface, portability of CORBA managed object implementations is guaranteed. On the other hand, the **JIDM::EventPortFinder** interface allows easier implementation of gateways between CORBA managers and managed object domains that are only accessible through standard management protocols (CMIP, SNMP).

In a pure CORBA environment, a complete solution may be implemented based on a graph of inter-connected **JIDM::EventPortFinder** objects. By configuring intermediary **JIDM::EventPortFinder** objects into a graph, the service for finding **JIDM::EventPort** objects can be built so that it administers access to a large number of managed object domains.

Whenever a managed object invokes the **find_event_port** operation, the request would traverse the graph until it reaches the target **JIDM::EventPort** object. As the request traverses the graph, each intermediary **JIDM::EventPortFinder** object would decide which link the request would traverse next. Decisions will be based upon information about each available link, any policies in force at that node, and value of parameters in the request.

Clearly, configuration of **JIDM::EventPortFinder** graphs and definition of policies to traverse these graphs requires definition of Federating interfaces, see the Life Cycle Service under the CORBAservices heading for definition of interfaces to federate **CosLifeCycle::GenericFactory** objects). However definition of such interfaces is beyond the scope of this specification since they are transparent to the clients (managed objects). Different federation policies can be supported and combined to implement a complete solution.

One possible alternative may consist of intermediate **JIDM::EventPortFinder** objects exporting an interface that enables binding a reference to a **JIDM::EventPortFinder** object together with a specific filter that is applicable to key and criteria values passed to the **find_event_port** operation. This would allow, as an example, to register one **JIDM::EventPortFinder** object as being able to find references to **JIDM::EventPort** objects associated with titles that fall within some specific range of title values.

Figure 2-10 on page 2-34 illustrates how federation of **JIDM::EventPorts** would work if a graph of intermediate **JIDM::EventPortFinder** objects is implemented. The following steps would be followed:

1. A managed object would invoke the **find_event_port** operation exposed by a **JIDM::EventPortFinder** object. The managed object typically would obtain a reference to this **JIDM::EventPortFinder** object using local initialization services.

2. The **JIDM::EventPortFinder** object that first receives the **access_domain** request would typically act as an intermediary **JIDM::EventPortFinder** object. Based on information available to that object, it would decide to pass requests on to another **JIDM::EventPortFinder** object.

3. The request would traverse the graph until it reaches the target **JIDM::EventPort** object. The last **JIDM::EventPortFinder** object in the graph would obtain a reference to the **CosEventChannelAdmin::SupplierAdmin** object in the channel and would return that reference.

4. As a result of this process, a valid reference to a **CosEventChannelAdmin::SupplierAdmin** object would be passed to the managed object that issued the request. Using this reference, the managed object is able to register as a push or pull supplier.

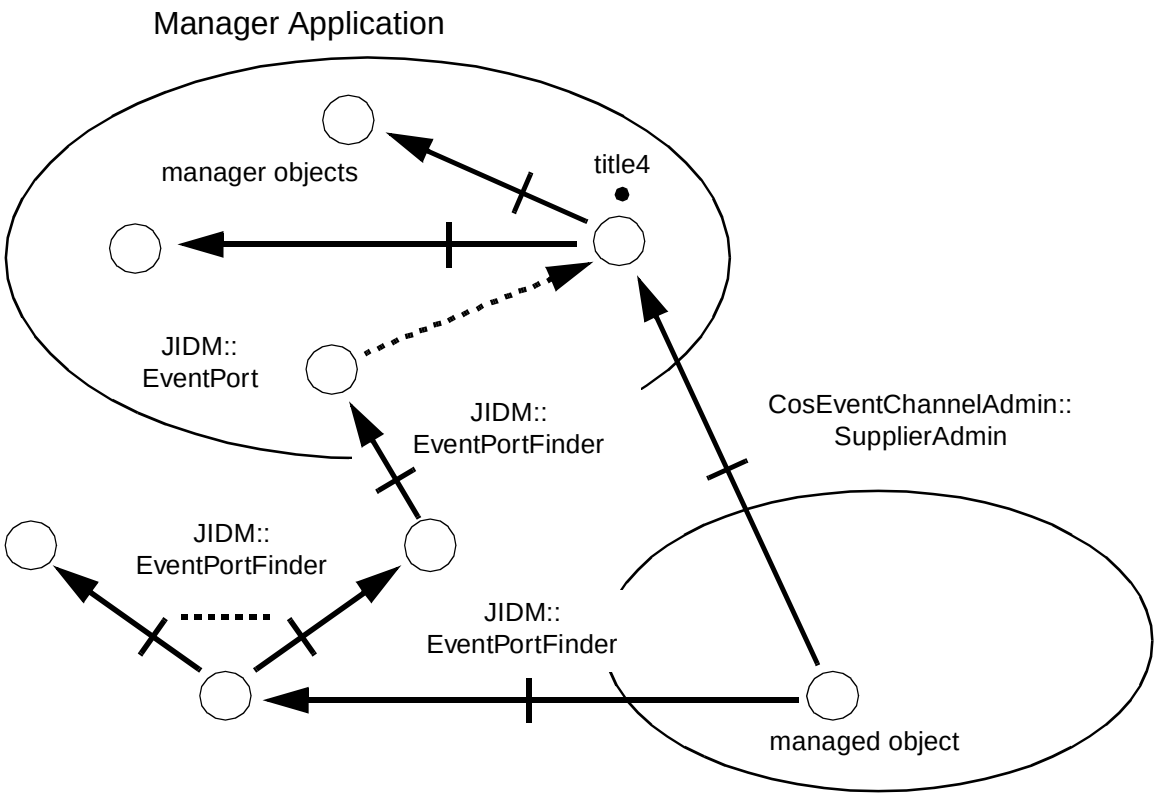

Figure 2-10 Finding JIDM::EventPort Objects

2.3 JIDM Gateways

This section is provided as information only, and does not represent a normative part of the specification. Different gateway scenarios are described where the use of this specification will be clarified. This should be considered as a high-level tutorial on some potential uses of the JIDM model. Also, some potential implementation options are discussed.

2.3.1 Manager Side Gateways

2.3.1.1 Overview

JIDM gateways must be used by any CORBA Manager Application needing to interoperate with managed object domains that are not directly accessible via CORBA but via other management-specific protocol such as CMIP or SNMP. By definition, a JIDM gateway is associated with only one management protocol. Therefore, we can refer to CORBA/CMIP gateways or CORBA/SNMP gateways whenever we want to designate explicitly which is the specific management protocol associated with a JIDM gateway at some CORBA Manager Application.

A JIDM gateway runs in one CORBA server; however, one or several JIDM gateways can coexist in the same CORBA server. Programs of the CORBA server have access to both ORB services and services encapsulating access to management-specific protocols provided by JIDM gateways at the server. Besides, there can be several CORBA servers containing JIDM gateways in the same CORBA Manager Application.

Any JIDM gateway typically has several CORBA objects associated with it.

- A **JIDM::ProxyAgentFinder** object for establishing connections to managed object domains being accessed through the gateway.
- One or several **JIDM::EventPort** objects for receiving notification of events from members of managed object domains being accessed through the gateway.

The **JIDM::ProxyAgentFinder** object is created during start-up of the CORBA server where the JIDM gateway is going to run. **JIDM::EventPort** objects at the gateway may be created during or after start-up of that server. Typically, this requires the existence of an **EventPortFactory** object at the gateway.

Several JIDM gateways can exist in a CORBA manager and one **JIDM::ProxyAgentFinder** object is typically associated with each of them. All the gateways would be registered in a root **JIDM::ProxyAgentFinder** object at the CORBA manager. CORBA managers can obtain a reference to this local root **JIDM::ProxyAgentFinder** object by using standard CORBA Initialization Services.

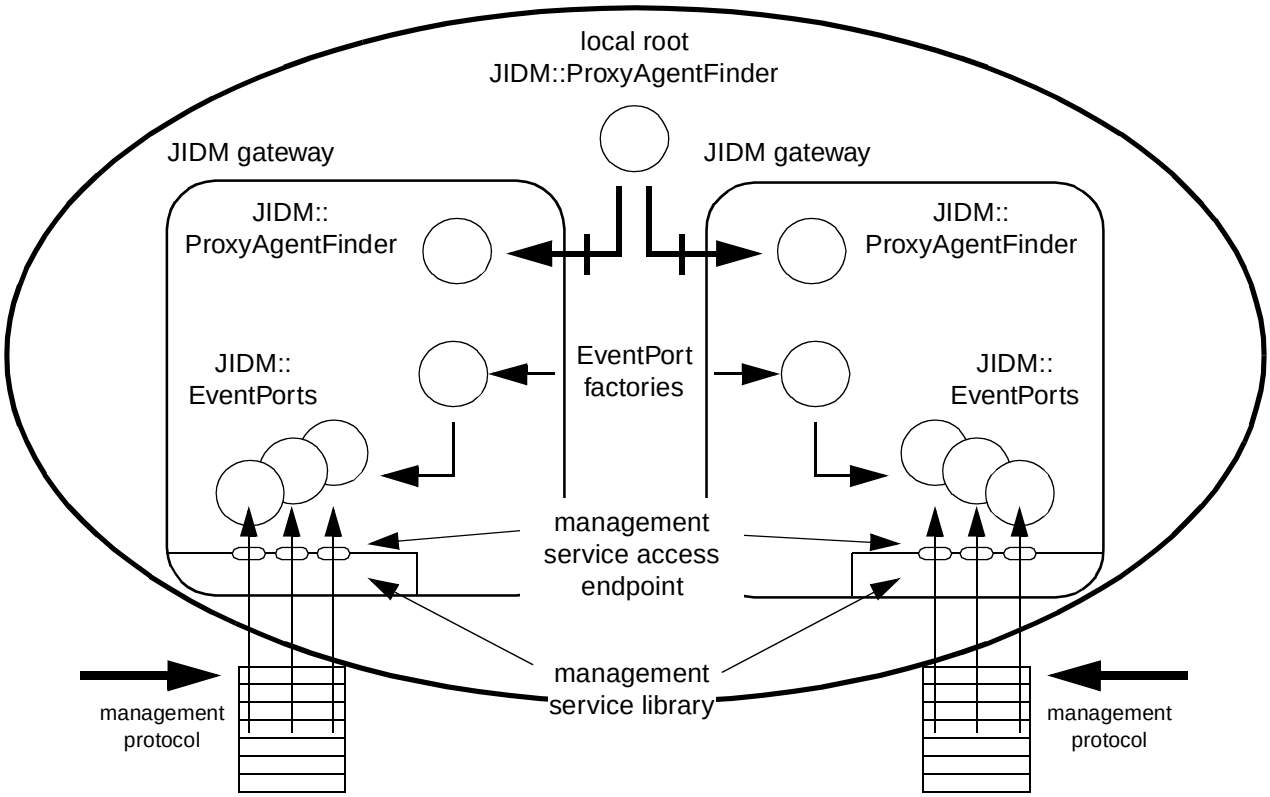

Figure 2-11 Structure of JIDM Gateways (manager side)

As a result of establishing a connection through a JIDM gateway, a **JIDM::ProxyAgent** object is created at the gateway. The **JIDM::ProxyAgent** objects created this way are responsible for:

- Creating a **CosLifeCycle::FactoryFinder** object that in turn enables creation of CORBA factories that handle creation of managed objects at the domain.
- Creating a **CosNaming::NamingContext** object that in turn enables creation of CORBA proxy managed objects for each member of the domain.

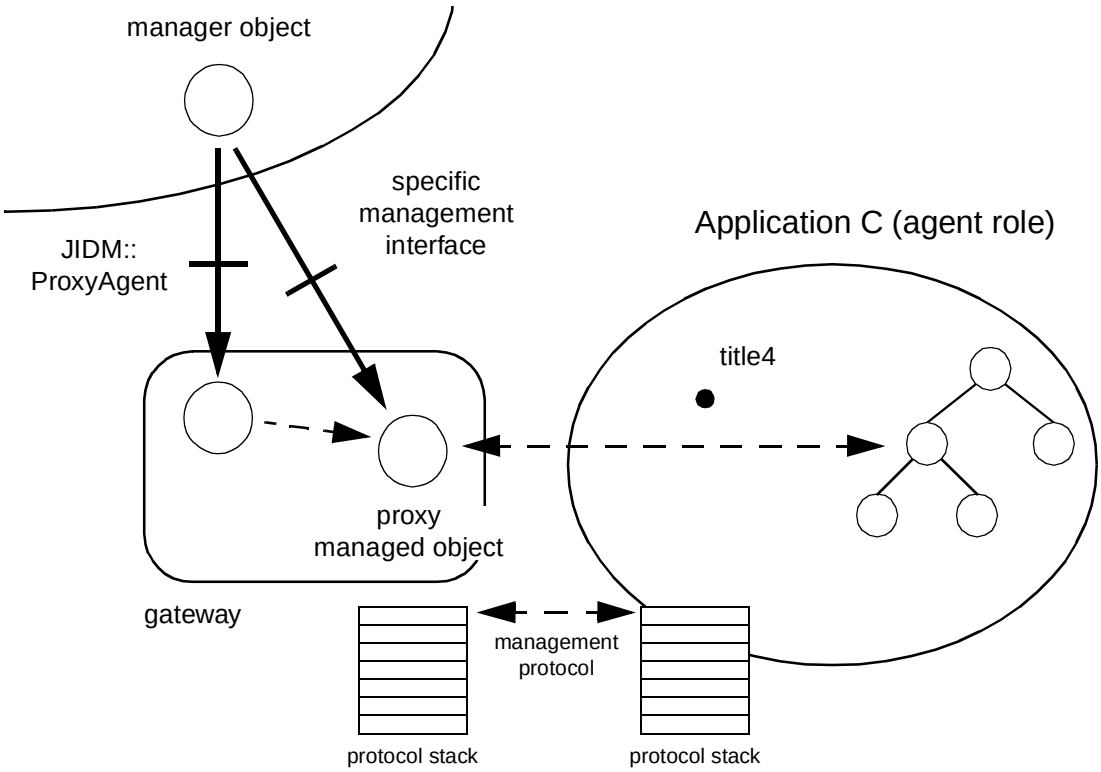

Figure 2-12 JIDM::ProxyAgents in a gateway

2.3.1.2 Getting access to managed object domains

The following steps are followed when a CORBA manager tries to get access to an external managed object domain using a JIDM gateway (see Figure 2-13 on page 2-38):

1. The CORBA manager invokes the **access_domain** operation exported by the **JIDM::ProxyAgentFinder** object located at the gateway. Information that unequivocally identifies the managed object domain to be accessed is passed in the invocation.

2. As a result of invoking the **access_domain** operation, a CORBA **JIDM::ProxyAgent** object is created at the gateway. The new **JIDM::ProxyAgent** object is bound to a management protocol communication endpoint (a service access point in OSI environments). If a specific domain title was specified in the criteria passed as argument to the **access_domain** operation, then a connection is established with the managed object domain. In such a case, the **JIDM::ProxyAgent** is responsible for managing resources associated with the connection.

3. A reference to the **JIDM::ProxyAgent** object is returned to the CORBA manager that requested access to the managed object domain being considered.

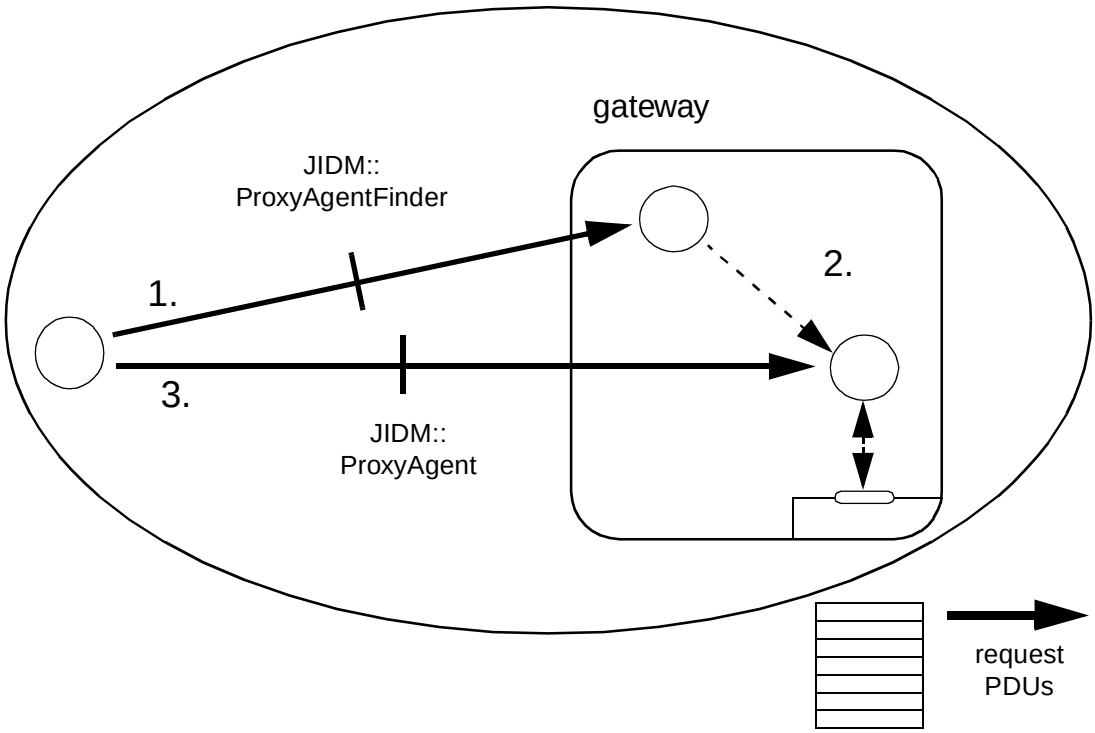

Figure 2-13 Finding References to JIDM::ProxyAgents in a JIDM Gateway

2.3.1.3 Creation of managed objects

These steps are followed when a CORBA manager creates a managed object at some domain that is accessible through a JIDM gateway (see Figure 2-14 on page 2-39):

1. The CORBA manager invokes the **get_domain_factory_finder** operation exported by the **JIDM::ProxyAgent** object.
2. The CORBA manager invokes the **find_factories** operation exported by the returned **CosLifeCycle::FactoryFinder** object, passing a valid key value.
3. The **CosLifeCycle::FactoryFinder** object finds references for appropriate managed object factories at the JIDM gateway. If there is no managed object factory matching the key, the **CosLifeCycle::FactoryFinder** object creates one. References to managed object factories are returned to the CORBA manager.
4. The CORBA manager invokes an operation on the managed object factory using the CORBA object reference it obtained. Typically, the CORBA manager narrows this object reference to a specific managed object factory interface supported by the factory (the **CosLifeCycle::GenericFactory** interface, for example).
5. The CORBA request is received by the JIDM gateway and is translated into an appropriate management create request PDU. This create request PDU is sent through the management protocol communication endpoint held by the **JIDM::ProxyAgent**.

6. When the response to the request PDU is received, the invoked operation returns with the appropriate result values.

7. If the **create** operation must return an object reference, then a CORBA proxy managed object is also created at the gateway.

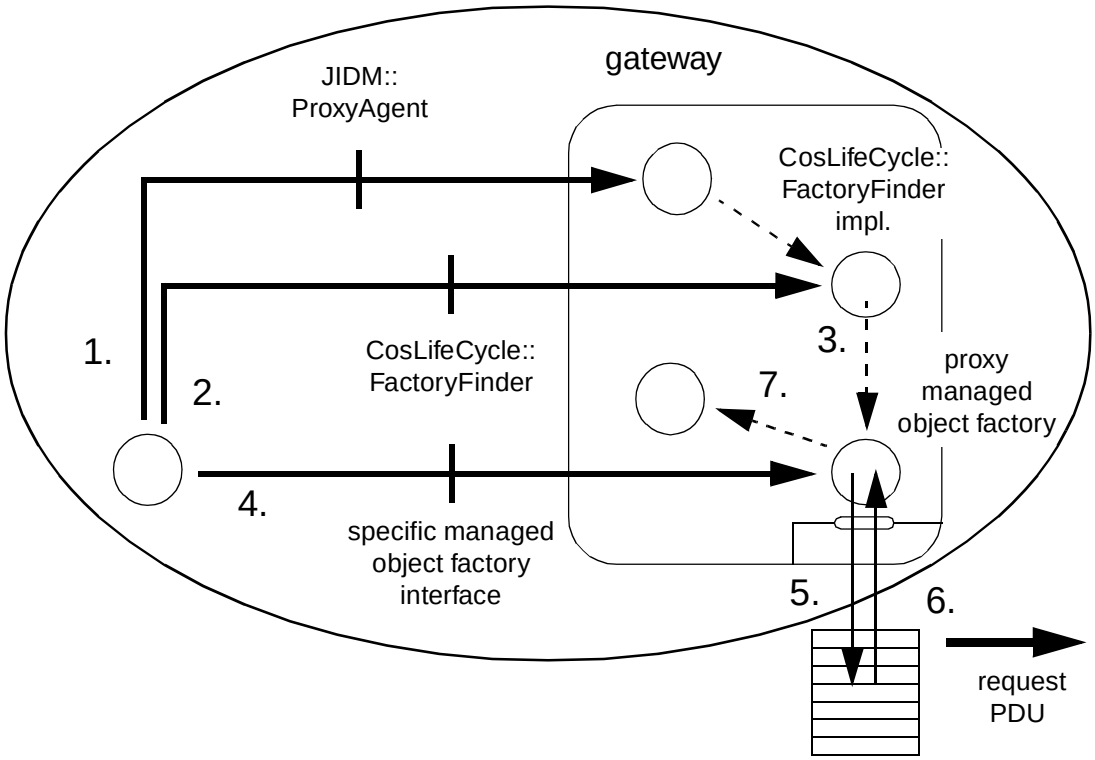

Figure 2-14 Creating Managed Objects through a JIDM Gateway

2.3.1.4 Invocation of operations on managed objects

These steps are followed when a CORBA manager invokes an operation on a managed object that is accessible through a JIDM gateway (see Figure 2-15 on page 2-40):

1. The CORBA manager invokes the **get_domain_naming_context** operation exported by the **JIDM::ProxyAgent** object.

2. A CORBA manager invokes the **resolve** operation exported by the returned **CosNaming::NamingContext** object, passing the name of the managed object upon which it wants to operate.

3. The **CosNaming::NamingContext** object finds a reference to the CORBA object acting as the proxy of the managed object and returns it to the CORBA manager that requested it. The CORBA proxy managed object resides in the JIDM gateway. The **CosNaming::NamingContext** object is responsible for creating the CORBA proxy managed object if it didn't exist at the gateway, the first time an existing managed object is accessed.

4. The CORBA manager invokes an operation on the managed object using the CORBA object reference to the corresponding proxy. IDL stubs or the standard DII can be used to perform this action. Whenever IDL stubs are used, the CORBA manager must narrow the reference, obtained from the **CosNaming::NamingContext**, to a specific OMG IDL interface (the **Printing::ManagedPrinter** interface, for example).

5. The CORBA request is received by the JIDM gateway and is translated into an appropriate management request PDU. This request PDU is sent through the management protocol communication endpoint held by the **JIDM::ProxyAgent**.

6. When the response to the request PDU is received, the invoked operation returns with the appropriate result values.

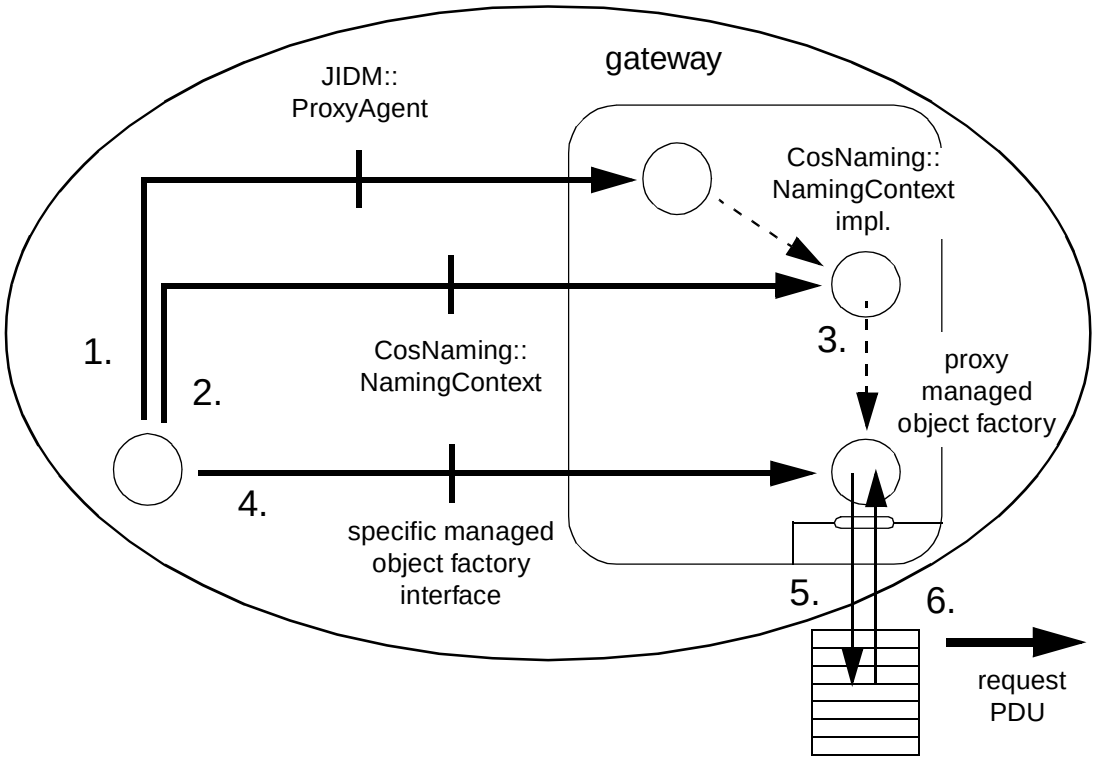

Figure 2-15 Invoking Operations on a Managed Object through a JIDM Gateway

2.3.1.5 Event reception

Events originated at managed object domains are always received through **JIDM::EventPort** objects at CORBA Managers. A mechanism is implemented at any JIDM gateway that allows event data received at a management connection endpoint to be forwarded to the appropriate **JIDM::EventPort** object.

As already mentioned in Section 2.2.4, “Reception of Events at CORBA Managers,” on page 2-25, different strategies to resolve how CORBA manager objects finally consume events can be implemented. Just to give an example, CORBA manager objects can register themselves directly to **JIDM::EventPorts** or via some additional event channel.

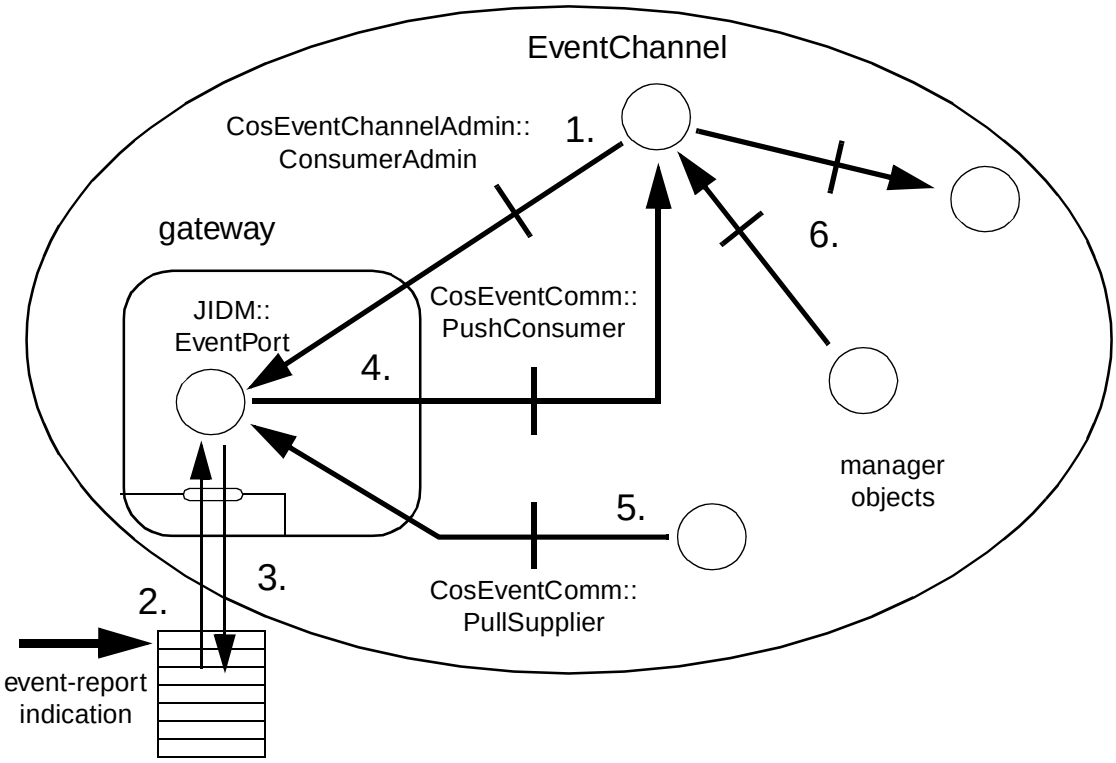

Figure 2-16 Event Reporting at JIDM Gateways (manager side)

These steps are followed when a CORBA manager receives an event through a **JIDM::EventPort** located at a gateway (see Figure 2-16):

1. During the start up phase of the CORBA Manager Application, one or more application objects register themselves as either **CosEventComm::PushConsumers** or **CosEventComm::PullConsumers** in each of the existing **JIDM::EventPorts**.

2. A PDU containing a notification of an event from a managed object is received by the JIDM gateway through some management connection endpoint. This management connection endpoint is bound to a specific title and has a **JIDM::EventPort** object associated with it, which finally receives the event data carried in the PDU.

3. The appropriate response (if applicable) is sent by the JIDM gateway back to the application that reported the event, confirming that the event was received at the Manager Application.

4. The **JIDM::EventPort** invokes the **push** operation exported by all **CosEventComm::PushConsumers** objects connected to it. Data of the event is passed in the invocation as an any.

5. The **JIDM::EventPort** maintains the event until all **CosEventComm::PullConsumers** objects connected to the port pull the event. Data of the event is obtained by consumers as an any.

6. **CosEventChannelAdmin::EventChannel** objects can be connected as consumers to the event port. In such a case, manager objects performing management functions can be connected to the channel instead of directly to the event ports.

2.3.2 Agent Side Gateways

2.3.2.1 Overview

JIDM gateways must be used by any CORBA Agent Application needing to offer a management interface based on some management-specific protocol such as CMIP or SNMP but not CORBA. By definition, a JIDM gateway is associated with one single management protocol. Therefore, we can refer to CORBA/CMIP gateways or CORBA/SNMP gateways whenever we want to designate explicitly which is the specific management protocol associated with a JIDM gateway for some given CORBA Agent Application.

A JIDM gateway runs in one CORBA server; however, one or several JIDM gateways can coexist in the same CORBA server. Programs in this server have access to both ORB services and services encapsulating access to management-specific protocols provided by JIDM gateways at the server. Besides, there can be several CORBA servers containing JIDM gateways in the same CORBA Agent Application.

Any JIDM gateway at a CORBA Agent Application has several objects associated with it (see Figure 2-17 on page 2-43):

- A **JIDM::EventPortFinder** CORBA object that enables CORBA managed objects at the agent application to establish connections to **JIDM::EventPort** objects at remote Manager Applications that are accessible through the gateway.
- A **JIDM::DomainPort** object that serves requests issued from remote Manager Applications that want to get access to managed objects at the local managed object domain.

These objects are created during start-up of the CORBA server where the JIDM gateway is located.

Several JIDM gateways can exist in a CORBA Agent and a **JIDM::EventPortFinder** object typically is associated with each of them. All the **JIDM::EventPortFinders** would be registered in a root **JIDM::EventPortFinder** object at the CORBA Agent.

An initial **CosLifeCycle::FactoryFinder** object and **CosNaming::NamingContext** object exist at any CORBA managed object domain. Whether these two interfaces are exported by the same CORBA object or different CORBA objects is an implementation issue. References to these CORBA objects can be obtained from a JIDM gateway by using the standard Initialization Services and are passed to the **JIDM::DomainPort** object at creation time.

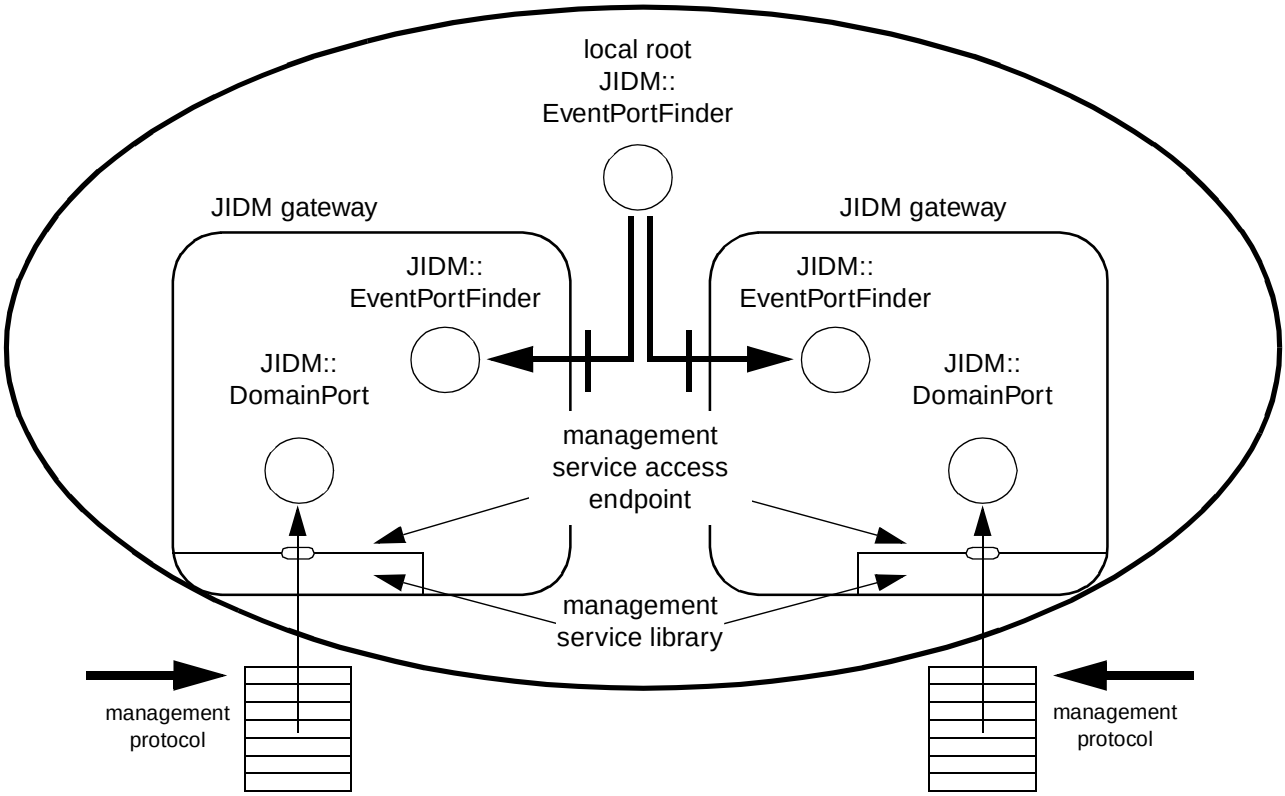

Figure 2-17 Structure of JIDM Gateways (agent side)

2.3.2.2 Handling access to managed objects

A **JIDM::DomainPort** object resides in the JIDM gateway to handle access to the managed object domain and serve association request issued from remote Manager Applications.

Every **JIDM::DomainPort** object has a title associated with it. This title is used by remote Manager Applications to identify the managed object domain associated with the **JIDM::DomainPort** object.

When a new association request is received by the **JIDM::DomainPort** object that is in a gateway, the **JIDM::DomainPort** object creates a new **JIDM::ProxyAgent** object. This object handles CMIS requests received through the newly established association.

A **JIDM::DomainPort** object in a JIDM gateway holds references to initial **CosNaming::NamingContext** and **CosLifeCycle::FactoryFinder** objects located at the managed object domain where the JIDM gateway is located. The **JIDM::DomainPort** object passes copies of these references to each **JIDM::ProxyAgent** object it creates.

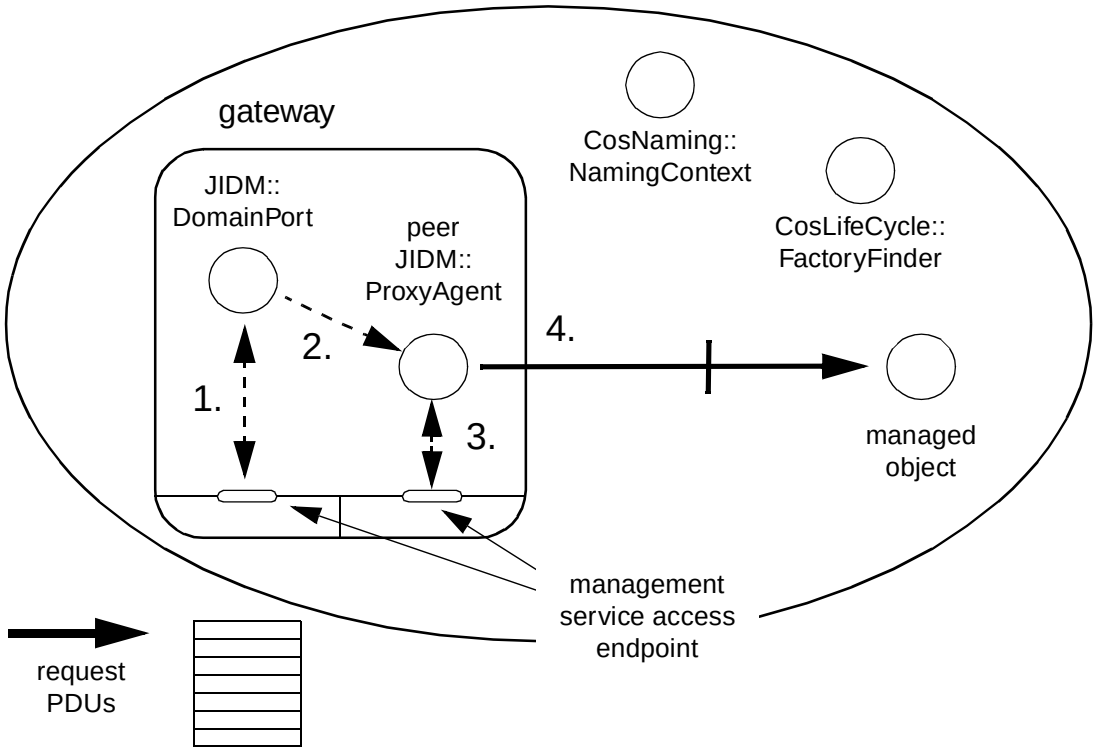

Figure 2-18 Handling Access to Local Managed Objects from a JIDM Gateway

2.3.2.3 Creation of managed objects

In CORBA Agent Applications, **JIDM::ProxyAgent** objects receive PDU indications, perform the appropriate operations, and return the appropriate PDU responses. These steps are followed each time a create PDU indication is received by a JIDM gateway:

1. A **JIDM::ProxyAgent** object receives a management create PDU indication through the management connection endpoint it holds.
2. The **JIDM::ProxyAgent** object finds an appropriate factory by invoking the **find_factories** operation provided by a **CosLifeCycle::FactoryFinder** object.
3. The **JIDM::ProxyAgent** object narrows the obtained Factory object reference to a new object reference associated with a specific factory interface. Next, it invokes the operation for creating managed objects exported by the factory being referenced.

4. The Factory object creates a new CORBA managed object, instance of the managed object type specified in the management create PDU indication.

5. The Factory object may bind a name (the one passed as the Managed object instance field in the management create PDU indication, but in IDL form) to the new CORBA managed object.

6. The Factory object may inform other CORBA managed objects, in the same managed object domain, that the new managed object has been created.

7. When the operation invoked by the **JIDM::ProxyAgent** object returns (or when an exception is raised), the **JIDM::ProxyAgent** object constructs and sends an appropriate create PDU response to the remote Manager Application.

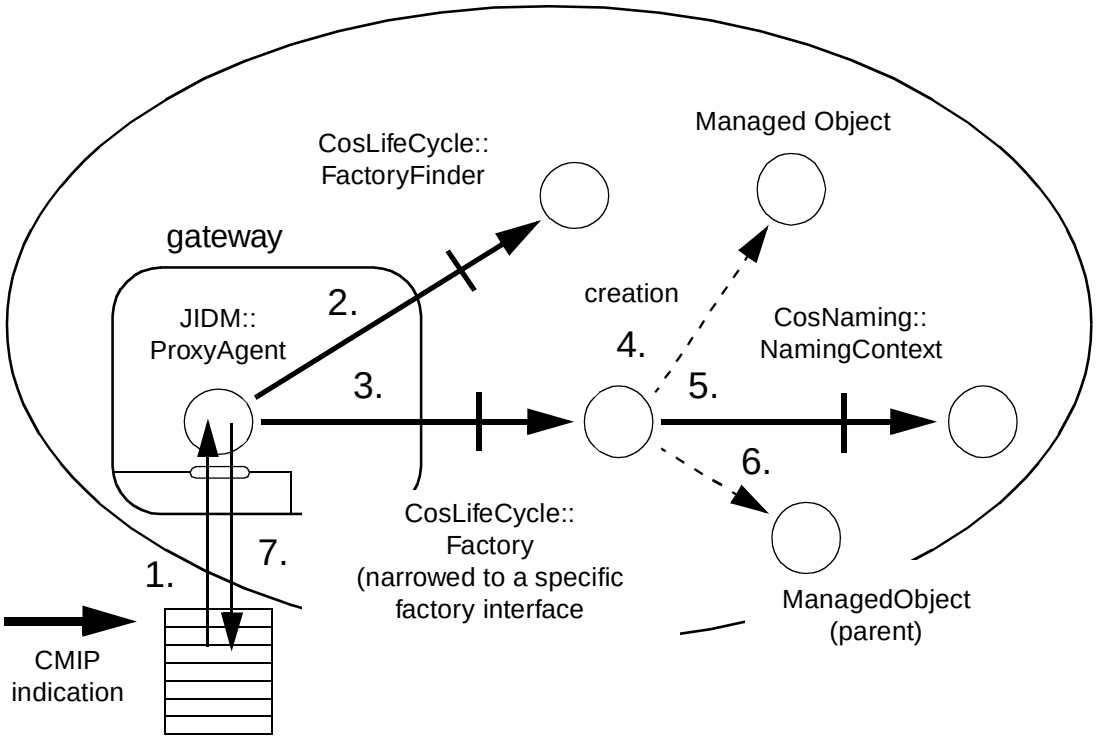

Figure 2-19 Handling Management Create PDU Indications

2.3.2.4 Invocation of operations on managed objects

The **JIDM::ProxyAgent** objects receive indications on single objects, perform the appropriate operations, and return the appropriate management PDU responses. The following steps are followed each time a PDU indication, corresponding to a management operation, is received by a JIDM gateway:

1. A **JIDM::ProxyAgent** object receives a management PDU indication, referred to a single managed object, through the management connection endpoint it holds. A name that unequivocally identifies the managed object is typically passed in the indication.

2. The **JIDM::ProxyAgent** object finds a CORBA object reference to the target managed object by invoking the resolve operation exported by a local **NamingContext** object. The name of the target managed object is passed in the invocation, once it is translated to IDL form.

3. The **JIDM::ProxyAgent** object invokes the appropriate operation on the managed object. In a Dynamic JIDM gateway, this may be accomplished by using the Dynamic Invocation API provided by the local ORB.

4. When the management operation invoked by the **JIDM::ProxyAgent** object returns (or when an exception is raised), the **JIDM::ProxyAgent** object constructs and sends an appropriate PDU response to the remote Manager Application.

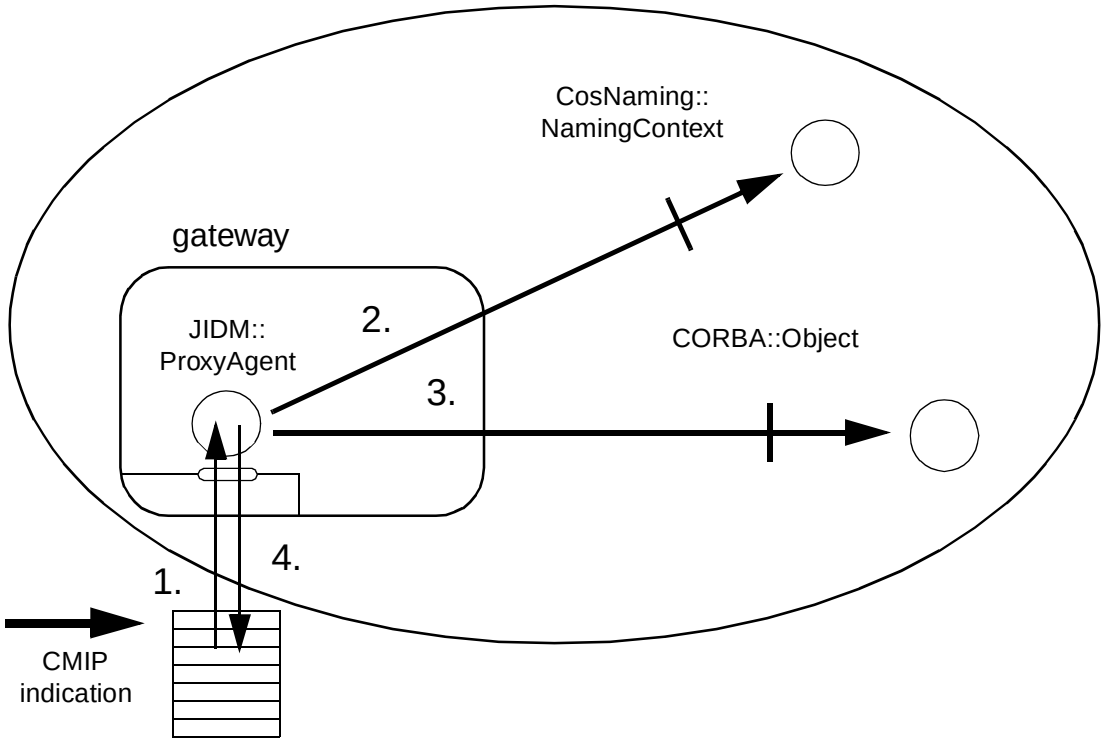

Figure 2-20 Invocation of Operations on Single Managed Objects

2.3.2.5 Event reporting

Using operations exported by **JIDM::EventPortFinder** objects located at a JIDM gateway, CORBA managed objects are able to find references to several **CosEventChannelAdmin::SupplierAdmin** objects, each of which points to a **JIDM::EventPort** associated with remote managers.

A managed object reports events to a destination (AE-title) by registering as a supplier in the corresponding **JIDM::EventPort** (via the standard **CosEventChannelAdmin::SupplierAdmin** interface returned by a **JIDM::EventPortFinder** object) and, then, supplying events to that channel.

Note that any managed object may register itself as a **CosEventComm::PushConsumer** or a **CosEventComm::PullConsumer** in a remote **JIDM::EventPort**.

As explained in Section 2.1.9, “The JIDM::EventPortFinder Interface,” on page 2-17, different strategies to resolve how CORBA managed objects finally report events can be implemented. Figure 2-21 illustrates one possible scenario where CORBA managed objects register themselves as **PushSuppliers** or **PullSuppliers** in a single object (called **EventReporter**), which in turn is registered as a **PushSupplier** in one or more remote **JIDM::EventPorts**. Basic steps are summarized as follows:

1. At creation time, the **EventReporter** object invokes the **find_event_port** operation exported by a **JIDM::EventPortFinder** object to find references associated with **CosEventChannel::SupplierAdmin** interfaces supported by a remote **JIDM::EventPort** object. It can try to find references for:
   - various **JIDM::EventPorts**, each of which is bound to one title contained in the list of destinations defined for the **EventReporter** object.
   - a single **JIDM::EventPort** bound to a wildcard address (only valid if automatic event forwarding - recipient manager resolution is supported).
2. The **JIDM::EventPortFinder** object creates a proxy **JIDM::EventPort** object if it doesn’t exist in the gateway. At the time it creates a proxy **JIDM::EventPort** object, it performs the necessary initial operations to obtain the reference to the **CosEventChannelAdmin::SupplierAdmin** object associated with the new **JIDM::EventPort** object.

   The **EventReporter** object registers itself as a **CosEventComm::PushSupplier** for each destination.
3. The **EventReporter** registers itself as a **CosEventComm::PushConsumer** in every local event channel that is necessary.
4. CORBA managed objects report events by using the standard event notification services.
5. Each event notification being generated is finally received by some event channel, connected to the **EventReporter** object.
6. The **EventReporter** object supplies the event to **JIDM::EventPort** objects corresponding to the different destinations.
7. The proxy associated with each **JIDM::EventPort** in the JIDM gateway constructs an appropriate event-report request PDU and sends it through the management communication endpoint it holds.

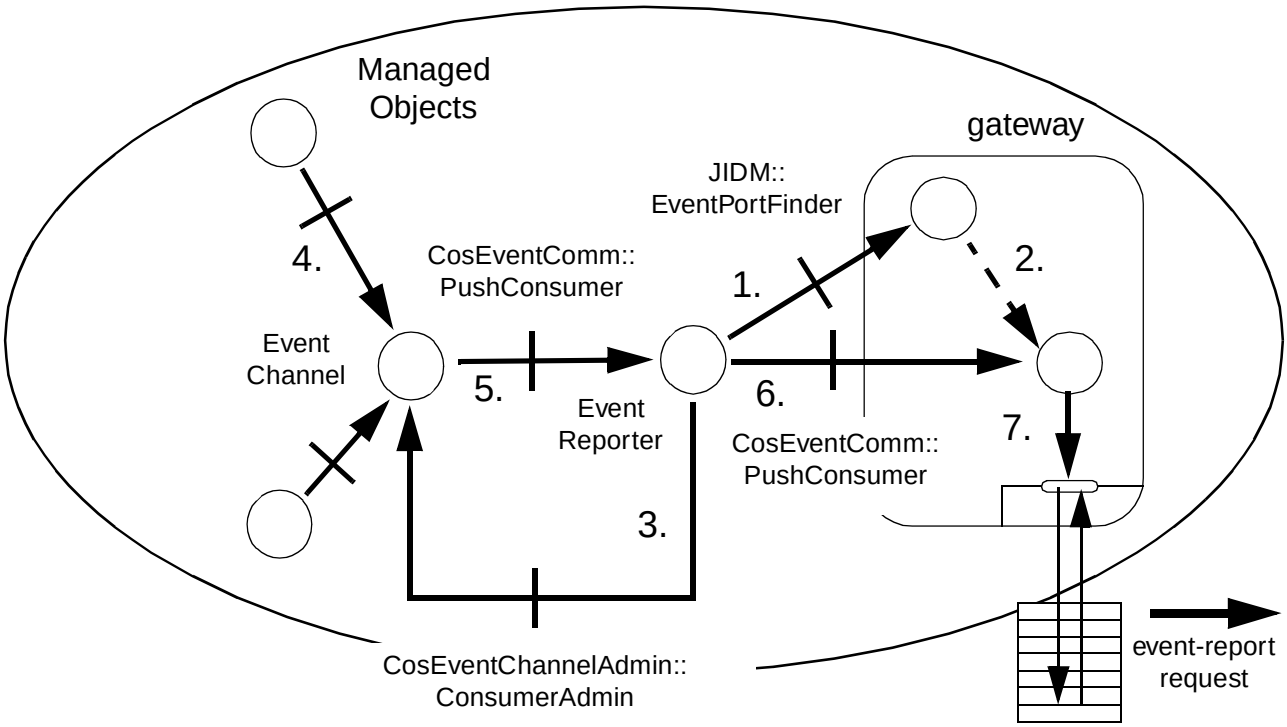

Figure 2-21 Sending Event Reports

OSI CORBA Facilities 3

Contents

This chapter contains the following sections.

3.1 The OSIMgmt Module

The **OSIMgmt** module comprises a collection of interfaces that together define a basic set of services for developing Systems Management Applications based on CORBA. This module contains the following interfaces:

- The **ProxyAgent** interface
- The **ManagedObject** interface
- The **ManagedObjectFactory** interface
- The **LocalRoot** interface
- The **LinkedReplyHandler**, **EndOfRepliesHandler**, and **MultipleRepliesHandler** interfaces
- The **RepliesIterator** and **BufferedRepliesHandler** interfaces
- The **LName** interface
- The **NamingContext** interface

This section describes these interfaces and their operations in detail.

```
#ifndef _OSIMGMT_IDL_
#define _OSIMGMT_IDL_

#include <JIDM.idl>
#include “X501Inf.idl”
#include “X711CMI.idl”

#pragma prefix “jidm.org”

// Macros used in the ‘raises’ clauses

#define ROSE_ERRORS\
    OSIMgmt::ROSEDuplicateInvocation,\
    OSIMgmt::ROSEMistypedArgument,\
    OSIMgmt::ROSEResourceLimitation, \
    OSIMgmt::ROSEUnrecognizedOperation

#define CREATE_ERRORS\
    ROSE_ERRORS, \
    OSIMgmt::AccessDenied,\
    OSIMgmt::ClassInstanceConflict,\
    OSIMgmt::DuplicateManagedObjectInstance,\
    OSIMgmt::InvalidAttributeValue,\
    OSIMgmt::InvalidObjectInstance,\
    OSIMgmt::MissingAttributeValue,\
    OSIMgmt::NoSuchAttribute,\
    OSIMgmt::NoSuchObjectClass,\
    OSIMgmt::NoSuchObjectInstance,\
    OSIMgmt::NoSuchReferenceObject,\
    OSIMgmt::ProcessingFailure,\
    OSIMgmt::ProcessingFailureEmpty

#define COMMON_ERRORS \
    ROSE_ERRORS, \
    OSIMgmt::AccessDenied, \
    OSIMgmt::ClassInstanceConflict, \
    OSIMgmt::ComplexityLimitation, \
    OSIMgmt::ComplexityLimitationEmpty, \
    OSIMgmt::InvalidScope, \
    OSIMgmt::InvalidFilter, \
    OSIMgmt::NoSuchObjectClass, \
    OSIMgmt::NoSuchObjectInstance, \
    OSIMgmt::ProcessingFailure, \
    OSIMgmt::ProcessingFailureEmpty, \
    OSIMgmt::SyncNotSupported

#define GET_ERRORS \
    COMMON_ERRORS, \
    OSIMgmt::GetListError, \
    OSIMgmt::OperationCancelled

#define SET_ERRORS \
    COMMON_ERRORS, \
    OSIMgmt::SetListError
```

```
#define ATTRIBUTE_ERRORS \
    COMMON_ERRORS, \
    OSIMgmt::GetListError, \
    OSIMgmt::SetListError

#define ACTION_ERRORS \
    COMMON_ERRORS, \
    OSIMgmt::InvalidArgumentValue, \
    OSIMgmt::NoSuchAction, \
    OSIMgmt::NoSuchArgument

#define DELETE_ERRORS \
    COMMON_ERRORS


module OSIMgmt
{
    // Definitions of ROSE and CMIS exceptions
    exception ROSEDuplicateInvocation { };
    exception ROSEMistypedArgument { };
    exception ROSEResourceLimitation { };
    exception ROSEUnrecognizedOperation { };

    exception AccessDenied { };
    exception ClassInstanceConflict
        { X711CMI::BaseManagedObjectIdType error_info; };
    exception ComplexityLimitation
        { X711CMI::ComplexityLimitationType error_info; };
    exception ComplexityLimitationEmpty { };
    exception DuplicateManagedObjectInstance
        { X711CMI::ObjectInstanceType error_info; };
    exception GetListError
        { X711CMI::GetListErrorType error_info; };
    exception InvalidArgumentValue
        { X711CMI::InvalidArgumentValueType error_info; };
    exception InvalidAttributeValue
        { X711CMI::AttributeType error_info; };
    exception InvalidFilter
        { X711CMI::CMISFilterType error_info; };
    exception InvalidScope
        { X711CMI::ScopeType error_info; };
    exception InvalidObjectInstance
        { X711CMI::ObjectInstanceType error_info; };
    exception MissingAttributeValue
        { X711CMI::MissingAttributeValueType error_info; };
    exception MistypedOperation { };
    exception NoSuchAction
        { X711CMI::NoSuchActionType error_info; };
    exception NoSuchArgument
        { X711CMI::NoSuchArgumentType error_info; };
    exception NoSuchAttribute
        { X711CMI::AttributeIdType error_info; };
    exception NoSuchObjectClass
        { X711CMI::ObjectClassType error_info; };
    exception NoSuchObjectInstance
```

```
        { X711CMI::ObjectInstanceType error_info; };
exception NoSuchReferenceObject
        { X711CMI::ObjectInstanceType error_info; };
exception OperationCancelled { };
exception ProcessingFailure
        { X711CMI::ProcessingFailureType error_info; };
exception ProcessingFailureEmpty { };
exception SetListError
        { X711CMI::SetListErrorType error_info; };
exception SyncNotSupported
        { X711CMI::CMISSyncType error_info; };
exception NoSuchEventType
        { X711CMI::NoSuchEventTypeType error_info; };
exception NoSuchInvokeId
        { X711CMI::InvokeIdTypeType error_info; };

// Using Multiple Replies exception for Actions
interface RepliesIterator; // forward declaration
exception UsingMR
        { RepliesIterator replies_iterator; };

// Definition of specific types used within this module
typedef string NameString;
typedef sequence<ASN1_ObjectIdentifier>
                        ASN1_ObjectIdentifierSeq;
struct AttributeValue {
        ASN1_ObjectIdentifier attribute_id;
        ASN1_DefinedAny value;
};
typedef sequence<AttributeValue> AttributeValueSeq;

// Type to be used in cmis_create operations
enum CreationKind
        {simple, autonaming, subordinate};


// Type to be used in scoped set operations
enum ModifyOperator
        {replace, add_member, remove_member,
                replace_with_default};

struct AttributeSetOperator {
        ModifyOperator modify_operator;
        ASN1_ObjectIdentifier attribute_id;
        ASN1_DefinedAny attribute_value;
};
typedef sequence <AttributeSetOperator>
                SetOperationArgument;

// Forward declaration for ReplyHandler interfaces
interface LinkedReplyHandler;
interface EndOfRepliesHandler;

// ProxyAgent
interface ProxyAgent : JIDM::ProxyAgent {
```

```
void cmis_create (
                    in CORBA::ScopedName interface_name,
                    in CreationKind creation_kind,
                    in CosNaming::Name object_name,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in CosNaming::Name reference_object,
                    in AttributeValueSeq req_attribute_values,
                    in LinkedReplyHandler reply_handler
);

void cmis_create_sync (
                    in CORBA::ScopedName interface_name,
                    in CreationKind creation_kind,
                    in CosNaming::Name object_name,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in CosNaming::Name reference_object,
                    in AttributeValueSeq req_attribute_values,
                    out CORBA::ScopedName created_interface_name,
                    out CosNaming::Name created_object_name,
                    out X711CMI::ASN1_GeneralizedTimeOpt creation_time,
                    out AttributeValueSeq created_attribute_values
) raises (CREATE_ERRORS);

void cmis_get (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in ASN1_ObjectIdentifierSeq attribute_id_list,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
);

void cmis_set (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in SetOperationArgument modification_list,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
);

void cmis_action (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
```

```
                    in ASN1_ObjectIdentifier action_name,
                    in ASN1_DefinedAny action_info,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

    void cmis_delete (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );
};

    const ASN1_ObjectIdentifier ACTUAL_CLASS = "2.9.3.4.3.42";

    interface ManagedObject; // forward declaration

    interface NamingContext : CosNaming::NamingContext {
        // NOTE: These operations are optional
        ManagedObject resolve_with_intf (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name
        ) raises (NotFound, CannotProceed, InvalidName);

        ManagedObject resolve_osi_name(
                    in ANSI_ObjectIdentifier managed_object_class,
                    in X711CMI::ObjectInstanceType object_instance
        ) raises (NotFound, CannotProceed, InvalidName);

        CosNaming::Name translate_osi_name (
                    in X711CMI::ObjectInstanceType object_instance
        ) raises (InvalidName);

        X711CMI::ObjectInstanceType translate_idl_name (
                    in CosNaming::Name idl_name
        ) raises (InvalidName);
    };

    // ManagedObject
    interface ManagedObject : NamingContext,
                CosLifeCycle::LifeCycleObject {
        readonly attribute CosNaming::Name object_name;

    void scoped_get (
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in ASN1_ObjectIdentifierSeq attribute_id_list,
                    in LinkedReplyHandler reply_handler,
```

```
                    in EndOfRepliesHandler end_of_replies_handler
    );

    void scoped_set (
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in SetOperationArgument modification_list,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

    void scoped_action (
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in ASN1_ObjectIdentifier action_name,
                    in ASN1_DefinedAny action_info,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

    void scoped_delete (
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

    AttributeValueSeq get_attributes (
                    in ASN1_ObjectIdentifierSeq attribute_id_list
    ) raises (GET_ERRORS);

    AttributeValueSeq set_attributes (
                    in SetOperationArgument modification_list
    ) raises (SET_ERRORS);

    ASN1_DefinedAny perform_action (
                    in ASN1_ObjectIdentifier action_name,
                    in ASN1_DefinedAny action_info
    ) raises (ACTION_ERRORS, UsingMR);

    void delete_mo () raises (DELETE_ERRORS);
};

    // ManagedObjectFactory
    interface ManagedObjectFactory {
        ManagedObject create (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in ManagedObject reference_object,
```

```
                in AttributeValueSeq requested_attribute_values
        ) raises (CREATE_ERRORS);

        ManagedObject create_with_auto_naming (
                in CORBA::ScopedName interface_name,
                in ManagedObject reference_object,
                in AttributeValueSeq requested_attribute_values
        ) raises (CREATE_ERRORS);

        ManagedObject create_subordinate (
                in CORBA::ScopedName interface_name,
                in CosNaming::Name superior_name,
                in ManagedObject reference_object,
                in AttributeValueSeq requested_attribute_values
        ) raises (CREATE_ERRORS);
};

// LocalRoot
typedef sequence<ManagedObject> ManagedObjectSeq;

interface LocalRoot : ManagedObject {
        exception NoDescendants {};
        ManagedObjectSeq list_orphans ( );

        ManagedObjectSeq
            list_orphan_descendants (in CosNaming::Name object_name)
                raises (NoDescendants);
};

// LName
interface LName {
        exception InvalidName {};

        readonly attribute boolean is_distinguished_name;
        readonly attribute unsigned long num_components;

        void from_osi_form (in X711CMI::ObjectInstanceType osi_name);
        X711CMI::ObjectInstanceType to_osi_form ()
                raises(InvalidName);
        void from_idl_form (in CosNaming::Name idl_name);
        CosNaming::Name to_idl_form ()
                raises(InvalidName);

        LName to_ancestor_name (in unsigned long levels_up)
                raises(InvalidName);
        LName to_relative_name (in unsigned long levels_up)
                raises(InvalidName);
        LName append (in LName name);
        LName append_ava (in X501Inf::AttributeValueAssertionType ava)
                raises(InvalidName);
        X501Inf::AttributeValueAssertionType get_ava (in unsigned long index)
                raises(InvalidName);

boolean equals (in LName name);
LName copy ();
```

```
        void from_string_form (in NameString name_string);
        NameString to_string_form ()
                        raises(InvalidName);

        void destroy ();
};

// ReplyHandler interfaces
interface LinkedReplyHandler {
        void send_reply (
                        in CORBA::ScopedName object_interface,
                        in CosNaming::Name object_name,
                        in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                        in any reply_info
);

        void send_mo_error (
                        in CORBA::ScopedName object_interface,
                        in CosNaming::Name object_name,
                        in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                        in short error_code,
                        in any error_info
        );

        void send_subtree_error (
                        in CORBA::ScopedName object_interface,
                        in CosNaming::Name object_name,
                        in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                        in short error_code,
                        in any error_info
        );
};
interface EndOfRepliesHandler {
        void end_of_replies ( );
};

interface MultipleRepliesHandler : LinkedReplyHandler, EndOfRepliesHandler {};

// BufferedRepliesHandler
struct Reply {
        CORBA::ScopedName  object_interface;
        CosNaming::Name  object_name;
        X711CMI::ASN1_GeneralizedTimeOpt  current_time;
        any reply_info;
};
typedef sequence<Reply> ReplyList;

interface RepliesIterator  {
        exception MoError {
                CORBA::ScopedName object_interface;
                CosNaming::Name object_name;
                X711CMI::ASN1_GeneralizedTimeOpt current_time;
                short error_code;
                any error_info;
```

```
        };

        exception SubtreeError {
            CORBA::ScopedName object_interface;
            CosNaming::Name object_name;
            X711CMI::ASN1_GeneralizedTimeOpt current_time;
            short error_code;
            any error_info;
        };

        boolean  get_reply (out Reply r) raises
                (MoError, SubtreeError);

        boolean  get_n_replies (in unsigned long how_many, out ReplyList r_list)
            raises (MoError, SubtreeError);

        boolean  finished (out unsigned long num_pending);
        void destroy ();
    };

    interface BufferedRepliesHandler : MultipleRepliesHandler, RepliesIterator {};

};

#define UsingMR OSIMgmt::UsingMR

#endif /* _OSIMGMT_IDL_ */
```

3.1.1 The OSIMgmt::LName Interface

In the OSI Systems Management Reference Model, any managed object is contained within one and only one containing managed object. The containment relationship is used for naming managed objects. Actually, any managed object is named by the combination of:

- The name of its containing object (superior object).
- Information uniquely identifying this object within the scope of its containing object.

Each managed object must be unambiguously identified within the scope of its superior (container) object by means of an attribute value assertion (AVA) denoting that a specified attribute has a specified value. When used for naming, an AVA is also called a relative distinguished name (RDN).

In OSI Systems Management, the name of a managed object can be expressed in two forms:

1. **Global form**: This form specifies an RDN sequence that unequivocally identifies the managed object with respect to the global root.

2. **Local form**: This form specifies an RDN sequence that unequivocally identifies the managed object with respect to a predefined context. For OSI systems management, this context is the system managed object and the local form name for the system managed object is the empty sequence.

The global name of a managed object is constructed by concatenating its local name to the global name of the system managed object representing the managed system where the managed object is located.

The local name of a managed object is constructed by appending the RDN that identifies the managed object within the scope of its superior object to the local form name of its superior object.

Through the use of the **OSIMgmt::LName** library, OSI names can be translated into **CosNaming::Names** and vice versa. Note that, using this library, code of a client doesn't have to use the 'Names Library' defined for the CosNaming Service.

```
typedef string NameString;

interface LName {
    exception InvalidName {};

    readonly attribute boolean is_distinguished_name;
    readonly attribute unsigned long num_components;

    void from_osi_form (in X711CMI::ObjectInstanceType osi_name);
    X711CMI::ObjectInstanceType to_osi_form ()
                raises(InvalidName);
    void from_idl_form (in CosNaming::Name idl_name);
    CosNaming::Name to_idl_form ()
                raises(InvalidName);

    LName to_ancestor_name (in unsigned long levels_up)
                raises(InvalidName);
    LName to_relative_name (in unsigned long levels_up)
                raises(InvalidName);
    LName append (in LName name);
    LName append_ava (in X501Inf::AttributeValueAssertionType ava)
                raises(InvalidName);
    X501Inf::AttributeValueAssertionType get_ava (in unsigned long index)
                raises(InvalidName);
    boolean equals (in LName name);
    LName copy ();

    void from_string_form (in NameString name_string);
    NameString to_string_form ()
                raises(InvalidName);

    void destroy ();
};
```

Although nothing prevents the use of **OSIMgmt::LNames** as regular CORBA objects that can be remotely accessed, they will be typically provided as library objects that will be locally accessed by clients of managed objects (CORBA managers).

3.1.1.1 Description of the LName operations

Besides the operations used to translate between different name formats (in this case, OSI name format, IDL naming format, and string format), the **OSIMgmt::Lname** interface defines several additional operations designed to ease the task of programming with an **OSIMgmt::LName** object:

- **to_ancestor_name** creates a new **OSIMgmt::LName** object that corresponds to the name of the ancestor managed object situated some number *n* of levels up (it deletes the last *n* AVAs of the name in OSI form).
- **to_relative_name** creates a new **OSIMgmt::LName** object that refer to the same managed object through a name relative to the ancestor managed object situated some number *n* of levels up (it deletes the first *m*-*n* AVAs of the name in OSI form, where *m* was the length of the name in OSI form).
- **append** creates a new **OSIMgmt::LName** object by appending the components of the name represented by the **OSIMgmt::LName** object passed as argument to the components of the current **OSIMgmt::Lname**.
- **append_ava** creates a new **OSIMgmt::LName** object by appending the provided AVA to the current name in OSI form.
- **get_ava** returns the AVA situated in a given position of the name represented by the **OSIMgmt::LName** object in OSI form.
- **equals** returns TRUE if the **OSIMgmt::LName** object represents the same name as the **OSIMgmt::LName** object passed as argument; note that this is name equality, not object equality (two names might refer to the same object, but be completely different).
- **copy** returns a reference to a new **OSIMgmt::LName** object whose state is copied from the current **OSIMgmt::LName** object.

Any attempt to invoke **to_ancestor_name**, **to_relative_name**, and **get_ava** passing a value bigger than the actual length of the name represented by the **LName** object will cause the BAD_PARAM exception to be raised.

Any attempt to extract or copy a value from an unitialized **LName** object will cause the InvalidName exception to be raised. **LName**s are initialized when they have been created from an already initialized **LName** object, or after a call to one of the **from_*** operations is successful.

3.1.1.2 Translation between CosNaming::Names and OSI ObjectInstance Names

Translation of OSI ObjectInstance names into **CosNaming::Names** implies performing the following steps:

1. To create an object of type **OSIMgmt::LName**.
2. To initialize the internal state of the **OSIMgmt::LName** object with a **X711CMI::ObjectInstanceType** value, by invoking the **from_osi_form** operation.

3. To produce a **CosNaming::Name** value by invoking the **to_idl_form** operation.

The reverse operation implies performing the following steps:

1. To create an object of type **OSIMgmt::LName**.
2. Initializes the internal state of the **OSIMgmt::LName** object with a **CosNaming::Name** value, by invoking the **from_idl_form** operation.
3. To produce a **X711CMI::ObjectInstanceType** value by invoking the **to_osi_form** operation.

The **OSIMgmt::LName** objects must be destroyed if not further used. They can be destroyed by invoking the **destroy** operation they expose.

The following example shows the code used to bind a name to a CORBA object reference that is pointing to an EFD managed object.

```
CosNaming::NamingContext_ptr ctx;
X721::eventForwardingDiscriminator_ptr efd;
X711CMI::ObjectInstanceType local_name;

// The OSI name of the EFD object was initialized some way:

local_name = ...;

// An OSIMgmt::LName variable is initialized:

OSIMgmt::LName_ptr efd_name = new OSIMgmt::LName ();
efd_name->from_osi_form (local_name);

// A name, in idl form, is bound with the reference to the EFD object:

ctx->bind (efd, efd_name->to_idl_form());

// free the space associated to the name of the managed object:

efd_name->destroy ();
```

3.1.1.3 Representation of CosNaming::Names

The internal representation of **CosNaming::Names** derived from OSI names is transparent to clients of managed objects. To develop portable applications, a programmer does not need to know how OSI names (**X711CMI::ObjectInstanceType** values) are translated into **CosNaming::Name** values. However, to ensure interworking between applications that are linked to different implementations of the **OSIMgmt::LName** library, a standard representation of **CosNaming::Names** is specified.

This section describes how **X711CMI::ObjectInstanceType** names are mapped into **CosNaming::Names**.

Each AVA in the OSI ObjectInstance Name will correspond to a **CosNaming::NameComponent** in the IDL form. The **kind** field in the **CosNaming::NameComponent** will always be an empty string. The **id** field in the **CosNaming::NameComponent** will correspond to a string with the format:

“<OID>=<value>”

where **<OID>** corresponds to the value of the registration OID of the Attribute template, in dot notation, and **<value>** denotes the value of such Attribute, in string format. Blank spaces are not allowed before or after the character ‘=’.

Simple values are mapped according to the following rules:

If **<value>** is...	then it...
an integer	is the decimal string representation of the integer itself, possibly preceded by a type indicator. All signed integer types are represented with an explicit sign (zero is considered positive for this matter), while all unsigned integer types do not have an explicit sign. If the value is of type “long”, a ‘0’ precedes the actual value. If it is of type “long long”, it is preceded by “00”. For example, the value 3 is represented by “+003” if its type is “long long”, and by “3” if its type is “unsigned short.”
a string	is the string embedded in double quotes (“). If the string contains a double quote (“) or backslash (\), it is preceded by the backslash character (as in \” or \\).
a boolean	is the string TRUE or FALSE.
a NULL	is the NULL string.
a sequence<octet>	is the sequence of octets printed as characters embedded in single quotes (‘). If a given octet is not printable, it is printed in octal representation (character \000 in octal, for example). If the single quote (‘) or backslash (\) character has to be included, it is preceded by the backslash character (as in \’ or \\).
a BIT STRING	will be received as a sequence<octet> and will be mapped the same way (because it cannot be distinguished from real sequence<octet>).
an ENUMERATED type	is the string corresponding to the value without quoting.

Complex values are represented according to the following rules:

If **<value>** is of a...	then it is represented as a string that starts with...
SEQUENCE type	character “{“, and contains the string representation of each component of the value separated by commas and ends with character “}.” Blank spaces are not allowed before and after “,”, after “{“ and before “}.”
SEQUENCE OF type	character “[“, contains the string representation of each component of the value separated by commas and ends with character “]”. Blank spaces are not allowed before and after “,”, after “[“ and before “].”
CHOICE type	the name of the selected field enclosed in parentheses, followed by the string representation of its value. Blank spaces are not allowed after and before parentheses around identifiers of selected fields.

Mapping of **ObjectInstance** names in **nonSpecificForm** is not supported. Therefore, a **CosNaming::Name** will correspond to the mapping of either an OSI Distinguished Name or an OSI RDNSequence. If it corresponds to an OSI Distinguished Name, then the first **CosNaming::NameComponent** will denote the Root object. The **id** field will be equal to the string “**root**” and the **kind** field will be equal to the empty string. This will help clients using OSIMgmt Facilities as well as implementations of interfaces defined as part of OSIMgmt Facilities (**cmis_get**, **cmis_set**, etc.) to distinguish whether a **CosNaming::Name** is local or global.

Figure 3-1 illustrates how the OSI **ObjectInstance** name in global form corresponding to an **X721::logRecord** object is mapped to a **CosNaming::Name**.

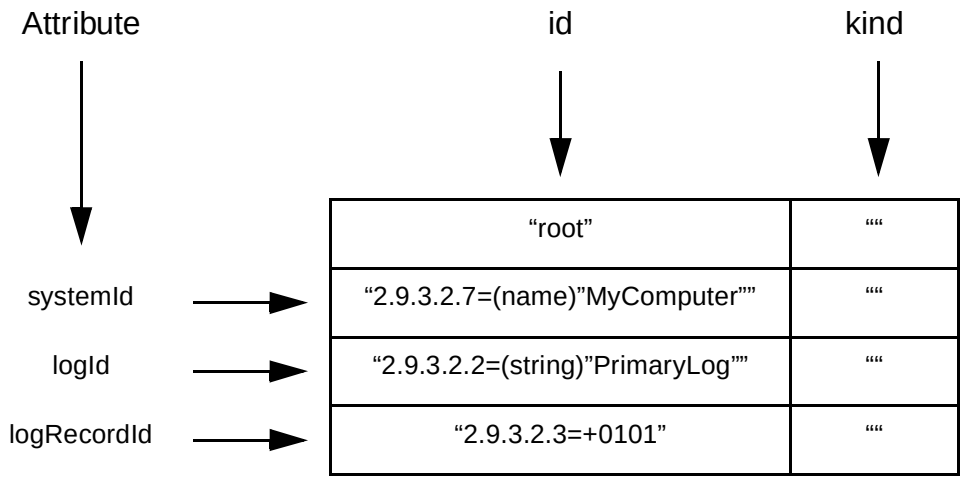

Figure 3-1 CosNaming::Name Associated with an X721::log Object

3.1.1.4 Representation of CosNaming::Names in string format

This specification defines a simple string format to represent managed object names. Names represented in this format (values of type **OSIMgmt::NameString**) can be converted (obtained) into (from) OSI Object Instance names or **CosNaming::Names** using **OSIMgmt::LName** objects.

The defined string format is aligned with the proposal presented in [interopNames]. The single character '/' is used to separate the **id** field values associated with each component of the managed object name in IDL form (which in turn correspond to AVAs in the OSI form). The value of **kind** fields is not represented since they correspond to empty strings. Therefore, the string used to represent a managed object name in local form matches the following format:

"<oid-1>=<value-1>/<oid-2>=<value-2>/.../<oid-n>=<value-n>"

while the string used to represent a managed object name global form matches the following format:

"root/<oid-1>=<value-1>/<oid-2>=<value-2>/.../<oid-n>=<value-n>"

The following example shows the code used to obtain a CORBA object reference that points to a given logRecord managed object, given its name.

```
CosNaming::NamingContext_ptr ctx;
OSIMgmt::NameString log_record_name;

// An OSIMgmt::LName variable is initialized with the name
// of the managed object in string format:

local_name = "2.9.3.2.2=(string)\"PrimaryLog\"/2.9.3.2.3=+0101";
OSIMgmt::LName_ptr log_record_name = new OSIMgmt::LName ();
log_record_name->from_string_form (local_name);

// A reference to the logRecord managed object is found by means of
// invoking resolve on the initial CosNaming::NamingContext
// located at the managed object domain:

CORBA::Object_ptr obj = ctx->resolve (log_record_name->to_idl_form());

// The reference obtained from resolve is narrowed, in order
// to invoke operations on the logRecord object:

X721::logRecord_ptr log_record = X721::logRecord::_narrow (obj);
ASN1_GeneralizedTime logging_time = log_record->loggingTimeGet ();

// free the space associated to the name of the managed object:

log_record_name->destroy ();
```

3.1.2 The OSIMgmt::ProxyAgent Interface

CORBA manager objects that require access to managed objects that are members of some given OSI managed object domain must establish a connection with that domain.

As a result of establishing the connection, an **OSIMgmt::ProxyAgent** object (an object that exports the **OSIMgmt::ProxyAgent** interface) is created. The **OSIMgmt::ProxyAgent** objects export the **JIDM::ProxyAgent** interface and support additional operations that are specific to OSI Management.

```
enum CreationKind
    {simple, autonaming, subordinate};

interface ProxyAgent : JIDM::ProxyAgent {

    void cmis_create (
                    in CORBA::ScopedName interface_name,
                    in CreationKind creation_kind,
                    in CosNaming::Name object_name,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in CosNaming::Name reference_object,
                    in AttributeValueSeq req_attribute_values,
                    in LinkedReplyHandler reply_handler
    );

    void cmis_create_sync (
                    in CORBA::ScopedName interface_name,
                    in CreationKind creation_kind,
                    in CosNaming::Name object_name,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in CosNaming::Name reference_object,
                    in AttributeValueSeq req_attribute_values,
                    out CORBA::ScopedName created_interface_name,
                    out CosNaming::Name created_object_name,
                    out X711CMI::ASN1_GeneralizedTimeOpt creation_time,
                    out AttributeValueSeq created_attribute_values
    ) raises (CREATE_ERRORS);

    void cmis_get (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in ASN1_ObjectIdentifierSeq attribute_id_list,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

                .
    void cmis_set (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
```

```
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in SetOperationArgument modification_list,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

    void cmis_action (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in ASN1_ObjectIdentifier action_name,
                    in ASN1_DefinedAny action_info,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );

    void cmis_delete (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in X711CMI::ScopeType scope,
                    in X711CMI::CMISFilterType filter,
                    in X711CMI::CMISSyncType synchronization,
                    in X711CMI::AccessControlTypeOpt access_control,
                    in LinkedReplyHandler reply_handler,
                    in EndOfRepliesHandler end_of_replies_handler
    );
};
```

Connections are established by invoking the **access_domain** operation exposed by a root **JIDM::ProxyAgentFinder** object as explained in Section 2.1.4, “The JIDM::ProxyAgentFinder Interface,” on page 2-11. The value associated with the “XSM environment” **Key** parameter passed to the **access_domain** operation is “OSI Management.” Note that the **access_domain** operation returns a reference to a **JIDM::ProxyAgent** interface. If the client wants to get visibility of the specific operations defined for the **OSIMgmt::ProxyAgent** interface, this reference must be narrowed.

Table 3-1 presents the names and meaning for criteria that can be passed in the invocation to the **access_domain** operation when trying to access an OSI managed domain. While the domain title criterion is mandatory, the rest of criteria components are optional.

Table 3-1 OSIMgmt Conventions for Proxy Agent Finding Criteria

criterion name	type of value	meaning
“domain title”	X227ACS::AE_titleType	AE-title associated with the managed object domain for which access is requested. The wildcard address is allowed.

criterion name	type of value	meaning
“controller object”	JIDM::ProxyAgentController	reference associated with a JIDM::ProxyAgentController object registered by the manager (OPTIONAL).
“access control”	X711CMI::AccessControlType	Information to be used as input to access control functions in establishing default access privileges for all exchanges on the association (OPTIONAL).
“requestor title”	X227ACS::AE_titleType	Title used to denote the Manager that requested access to the OSI managed object domain (OPTIONAL).

Semantics of the domain title and controller object parameters were specified in Section 2.1.4, “The JIDM::ProxyAgentFinder Interface,” on page 2-11. The criteria, in the case of OSI Systems Management Reference model, may include additional parameters, namely:

- An access control parameter, carrying access control information required to set up the connection and to be used as default access privileges.
- A requestor title parameter, used to identify the CORBA manager application that requests the connection.

It must be pointed out that invoking the **access_domain** operation with two different <key, criteria> pairs will result in creation of two different connections and, consequently, two different **OSIMgmt::ProxyAgent** objects. As an example, passing the same AE-title value but two different access control parameter values or two different controller objects to get access to an OSI managed object domain, would imply creation of two different **OSIMgmt::ProxyAgents**.

The requestor title is mainly required in those scenarios where a requestor needs to create a new connection, not shared with other requestors who use the same destination AE-title and access control parameter values. Note that sharing an already existing **OSIMgmt::ProxyAgent** object would mean to accept that other OSI Managers may destroy that object.

Since **OSIMgmt::ProxyAgent** objects are **JIDM::ProxyAgent** objects, they provide the means by which CORBA manager objects are able to obtain references to:

- An initial **CosLifeCycle::FactoryFinder** object located at the OSI managed object domain.
- An initial **CosNaming::NamingContext** object located at the OSI managed object domain.

Invoking the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object, CORBA manager objects may find factories that enable creation of new members of the OSI managed object domain.

Invoking the **resolve** operation exposed by the initial **CosNaming::NamingContext** object, CORBA manager objects may obtain CORBA object references to existing members of the OSI managed object domain.

In a pure CORBA environment (i.e., both manager and managed object domains are based on CORBA), the **OSIMgmt::ProxyAgent** would typically hold references to the initial **CosLifeCycle::FactoryFinder** and **CosNaming::NamingContext** objects located at the OSI domain being accessed (see Figure 3-2). Whether these two interfaces are exported by the same CORBA object or different CORBA objects at the domain is an implementation issue.

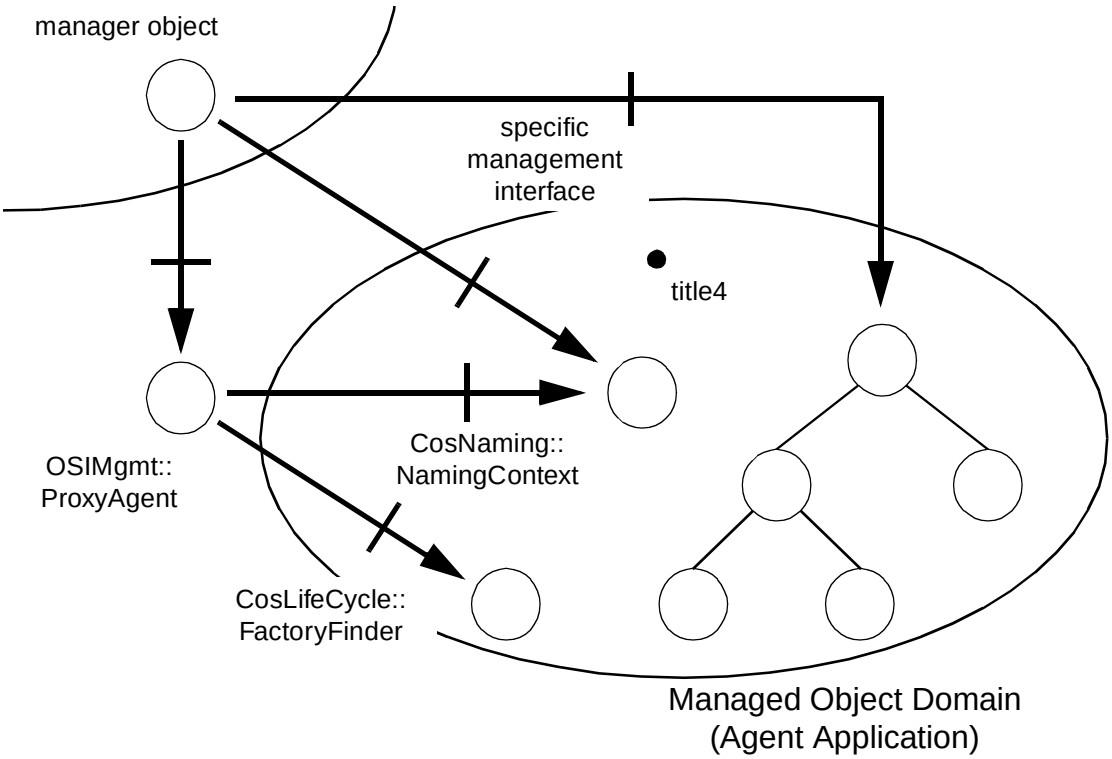

Figure 3-2 OSIMgmt::ProxyAgents in a CORBA Environment

Once a CORBA manager object obtains a CORBA object reference associated with an OSI managed object, it can invoke operations exposed by the object. It will do so by using the standard ORB services defined in *CORBA The Common Object Request Broker: Architecture and Specification:*

- the Dynamic Invocation Interface (DII), or
- IDL stubs generated from definitions in OMG IDL of interfaces exported by the object, which might have been generated from GDMO definitions according to XoJIDM (see Appendix A, “References”).

3.1.2.1 Description of the ProxyAgent operations

The get_domain_factory_finder operation

The **get_domain_factory_finder** operation obtains a reference to the initial **CosLifeCycle::FactoryFinder** object located at the domain being accessed through an **OSIMgmt::ProxyAgent** object. As already explained in Section 2.1.2, "The JIDM::ProxyAgent Interface," on page 2-4, CORBA manager objects can locate appropriate managed object factories by invoking the **find_factories** operation exposed by this initial **CosLifeCycle::FactoryFinder** object.

The space of keys established for OSI Management environments is described in Table 3-2.

Table 3-2 OSIMgmt Conventions for Factory Finder Keys

id field	kind field	meaning
fully scoped name of object interface	"superior object interface"	Find factories that create objects whose superior object supports the named interface.
fully scoped name of object interface	"object interface"	Find factories that create objects supporting the named interface.
fully scoped name of factory interface	"factory interface"	Find factories supporting the named factory interface.

CORBA Managers can create managed objects by using operations exposed by specific factories whose interfaces are derived from name-binding GDMO templates or operations exposed by generic factories.

For specific factories, one of the two following scenarios may be supported:

1. A specific factory interface is defined for each managed object interface that supports a different operation for each GDMO name-binding template.
2. A specific factory interface is defined for each GDMO name-binding template.

In respect to generic factories, one (or several) of the three following scenarios may be supported:

1. The standard **CosLifeCycle::GenericFactory** interface is used.
2. The **OSIMgmt::ManagedObjectFactory** interface is used.
3. One of the standard factory interfaces defined in SYSMAN facilities (see Appendix A, "References") is used.

In any case, the factory object would be responsible to check if the new managed object can be contained in the designated superior object. This implies checking for some name-binding template declaring that this relationship is valid.

With these considerations in mind, the alternatives for finding factories in OSI Systems Management environments are more precisely described as follows.

Only the name of the object interface is specified

Here, it is implicitly assumed that there is a specific factory interface associated with the managed object interface. Such interface includes a separate operation for each GDMO name-binding template that is associated with the managed object interface. CORBA managers know the name and operations associated with the factory in advance so they can properly narrow and use the reference returned by the **find_factories** operation.

Only the name of the object factory interface is specified

Here, references returned by the **find_factories** operation can be narrowed to the IDL interface whose name has been specified. The CORBA manager object who invoked the operation knows the signature and semantics of operations supported by the designated object factory interface. This option is the one used to obtain references to factories derived from GDMO name-binding templates or to obtain references to generic factories exporting the **CosLifeCycle::GenericFactory** interface, the **OSIMgmt::ManagedObjectFactory** interface, or any of the generic factory interfaces defined in SYSMAN facilities (see Appendix A, “References”).

Both the name of the object interface and the superior object interface are specified

Here, the CORBA manager object provides the necessary information to find the factory associated to a specific GDMO name-binding template for which a specific factory interface is defined.

In cases where objects are created through **CosLifeCycle::GenericFactory** objects, the **Key** value passed in the invocation to the **create_object** operation is the name of the interface exported by the new managed object. The **Criteria** value is a sequence of <name, value> pairs that correspond to the rest of the arguments needed for creation of the managed object as specified in Table 3-3 (name of the managed object, name of superior object, reference object, attribute list, etc).

Table 3-3 OSIMgmt Conventions for Managed Object Creation Criteria

criterion name	type of value	interpretation
“managed object interface”	CORBA::ScopedName	Name of interface exported by the new managed object.
“managed object name”	CosNaming::Name	When this parameter is supplied, it contains the name of the new managed object.
“superior object name”	CosNaming::Name	When this parameter is supplied, it contains the name of the managed object which is to be the superior of the new managed object.

criterion name	type of value	interpretation
“reference object”	OSIMgmt::ManagedObject	When this parameter is supplied, it contains the value of a reference to an existing managed object to be considered as reference for initialization.
“attribute list”	OSIMgmt::AttributeValueSeq	When this parameter is supplied, it contains a set of attribute identifiers and values to be assigned to the new managed object.

The get_domain_naming_context operation

The **get_domain_naming_context** operation obtains a reference to the initial **CosNaming::NamingContext** object located at the domain being accessed through an **OSIMgmt::ProxyAgent** object.

As already explained in Section 2.1.2, “The JIDM::ProxyAgent Interface,” on page 2-4, CORBA manager objects can obtain CORBA object references to members of a managed object domain as a result of invoking the **resolve** operation exposed by the initial **CosNaming::NamingContext** object located at the domain. The **resolve** operation may also be used to obtain reference to **CosNaming::NamingContext** objects subordinated to the initial **CosNaming::NamingContext** object.

Managed objects will be named according to OSI Naming Principles defined in X720 (see Appendix A, “References”). CORBA manager objects will typically perform the following steps to obtain a reference to an OSI managed object:

1. Construct the name of the managed object in OSI form.
2. Translate the name from OSI to idl form (see Section 3.1.1, “The OSIMgmt::LName Interface,” on page 3-10).
3. Invoke the **resolve** operation exposed by the initial **CosNaming::NamingContext** located at the domain where the object is located.

The following example shows how the fragment of code used to find a LogRecord object by name should look..

```
OSIMgmt::ProxyAgent_ptr agent;

// a reference to a JIDM::ProxyAgent is obtained as a result of
// establishing a connection to the managed object domain where
// the printer is located:
agent = ...;
.....

// a reference to the initial CosNaming::NamingContext object
// is obtained:
```

```
CosNaming::NamingContext_ptr ctx = agent -> get_domain_naming_context ();
.....

// the name of the log record is constructed:
OSIMgmt::LName_ptr log_name = new OSIMgmt::LName ();
log_name->from_string_form ("2.9.3.2.2=(string)\"PrimaryLog\"
        /2.9.3.2.3=0101");

// find a reference to the object with the log_name value in IDL form:
CORBA::Object_ptr obj = ctx->resolve (log_name->to_idl_form());
log_name->destroy ();

// narrows the value returned by the resolve operation:
X721::logRecord_ptr a_log_rec = X711::logRecord::_narrow (obj);

// operations on the log can now be invoked:
ASN1_GeneralizedTime logging_time = a_log_rec->loggingTimeGet ();
```

CMIS operations

OSIMgmt::ProxyAgent objects support operations that enable CORBA Managers to operate upon selected descendants of managed objects that are members of a managed object domain. These operations are referred to as scoped operations.

A detailed description about CMIS operations is presented in the Section 3.1.6, "Description of CMIS Operations," on page 3-33.

The destroy operation

Any **OSIMgmt::ProxyAgent** object exposes the **destroy** operation, which disposes the object. Disposing an **OSIMgmt::ProxyAgent** object means closing the connection established to the corresponding managed object domain. If the **OSIMgmt::ProxyAgent** object was running in a JIDM gateway server, destruction of the object implies freeing resources used to maintain the associated connection (closing a XMP descriptor, for example).

Destruction of an **OSIMgmt::ProxyAgent** object can take place either gracefully or non-gracefully, as described in Section 2.1.2, "The JIDM::ProxyAgent Interface," on page 2-4. A reference to a **JIDM::ProxyAgentController** object may be passed at the manager side, as described in Section 2.1.3, "The JIDM::ProxyAgentController Interface," on page 2-9.

3.1.3 The OSIMgmt::NamingContext Interface

The **OSIMgmt::NamingContext** interface provides a placeholder for specialized and extended naming operations that may be performed in an OSI management context. This interface extends the basic **CosNaming::NamingContext**.

In this section, a basic set of such specialized operations is described. Note that all operations are OPTIONAL, and no implementation is required to support any of them to be considered fully compliant with this specification.

```
const ASN1_ObjectIdentifier ACTUAL_CLASS = "2.9.3.4.3.42";

interface NamingContext : CosNaming::NamingContext {
    // NOTE: These operations are optional
    ManagedObject resolve_with_intf (
                        in CORBA::ScopedName interface_name,
                        in CosNaming::Name object_name
    ) raises (NotFound, CannotProceed, InvalidName);

    ManagedObject resolve_osi_name (
                    in ASN1_ObjectIdentifier managed_object_class,
                    in X711CMI::ObjectInstanceType object_instance
    ) raises (NotFound, CannotProceed, InvalidName);

    CosNaming::Name translate_osi_name (
                    in X711CMI::ObjectInstanceType object_instance
    ) raises (InvalidName);

    X711CMI::ObjectInstanceType translate_idl_name (
                    in CosNaming::Name idl_name
    ) raises (InvalidName);
};
```

The resolve_with_intf operation

This operation is equivalent to the **CosNaming::NamingContext::resolve** operation, but takes an extra parameter that indicates the managed object class supported by the object being located. This operation is useful when accessing a managed domain that is unable to perform location operations based solely on object instance names, that is, when accessing agents that do not support the ActualClass functionality, as specified in [X720].

The exceptions raised by this operation are the same, and have the same semantics, as those raised by the **CosNaming::NamingContext::resolve** operation.

The resolve_osi_name operation

This operation obtains a reference to a managed object given its OSI name (and, potentially, the managed object class to which it belongs). The OSI name includes information such as whether the name is in global form or in local form, and the sequence of attribute value assertions forming the path to the object being located.

If the class of the object being located is not known, the constant ACTUAL_CLASS may be used instead (provided that the managed domain being accessed supports this functionality). Additionally, the empty string is considered equivalent to ACTUAL_CLASS when passed to this operation as the value of the **object_class** parameter.

The exceptions raised by this operation are the same, and have the same semantics, as those raised by the **CosNaming::NamingContext::resolve** operation.

The translate_osi_name operation

This operation returns the **CosNaming::Name** corresponding to the **X711CMI::ObjectInstanceType** passed as input parameter. This operation is useful if the **OSIMgmt::LName** functionality is not available.

This operation may raise the InvalidName exception if the input name has not been properly intialized (contains a valid name). Note that translating the name does not require an object with that name to exist.

The translate_idl_name operation

This operation returns the **X711CMI::ObjectInstanceType** corresponding to the **CosNaming::Name** passed as input parameter. This operation is useful if the **OSIMgmt::LName** functionality is not available.

This operation may raise the InvalidName exception if the input name has not been properly intialized (contains a valid name). Note that translating the name does not require an object with that name to exist.

3.1.4 The OSIMgmt::ManagedObject interface

The standard **X721::top** interface inherits from **OSIMgmt::ManagedObject** interface. As a consequence, all management interfaces generated by a GDMO to IDL compiler inherit (indirectly) from the **OSIMgmt::ManagedObject** interface. This inheritance tree is shown in Figure 3-3.

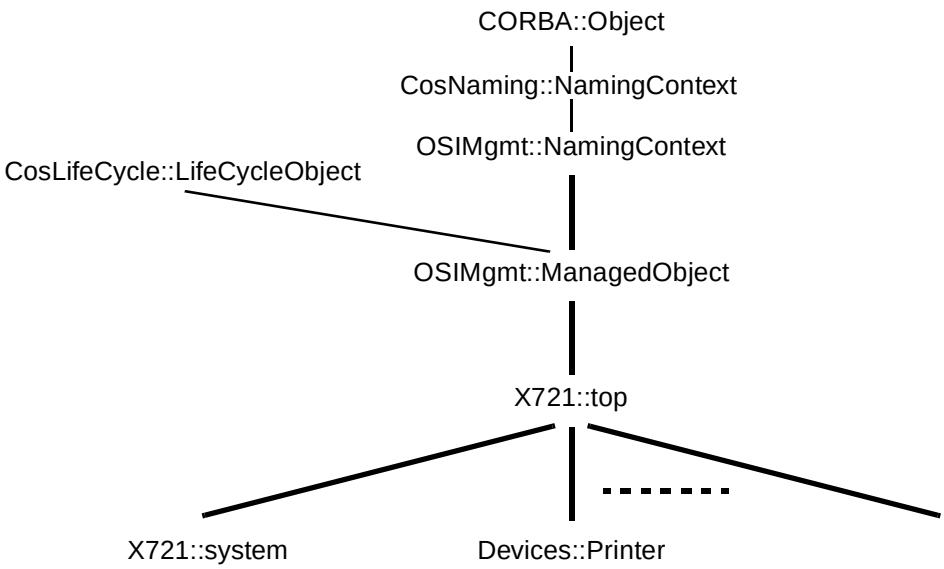

Figure 3-3 Inheritance Tree

Operations exposed through the **OSIMgmt::ManagedObject** interface enable clients of an OSI managed object to obtain the name of the managed object and invoke operations either on the object itself or on selected descendants of the managed object.

Every **OSIMgmt::ManagedObject** CORBA object supports the principles on transparency specified in Section 2.1.1, “JIDM Managed Objects,” on page 2-3..

```
interface ManagedObject : NamingContext, CosLifeCycle::LifeCycleObject {
    readonly attribute CosNaming::Name object_name;

    void scoped_get (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in ASN1_ObjectIdentifierSeq attribute_id_list,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handle);

    void scoped_set (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in SetOperationArgument modification_list,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
    );

    void scoped_action (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in ASN1_ObjectIdentifier action_name,
                in ASN1_DefinedAny action_info,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
    );

    void scoped_delete (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,

                .
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
    );

    AttributeValueSeq get_attributes (
                in ASN1_ObjectIdentifierSeq attribute_id_list
    ) raises (GET_ERRORS);
```

```
        AttributeValueSeq set_attributes (
                in SetOperationArgument modification_list
        ) raises (SET_ERRORS);

        ASN1_DefinedAny perform_action (
                in ASN1_ObjectIdentifier action_name,
                in ASN1_DefinedAny action_info
        ) raises (ACTION_ERRORS, UsingMR);

        void delete_mo () raises (DELETE_ERRORS);
};
```

3.1.4.1 Naming

The **OSIMgmt::ManagedObject** interface inherits indirectly from the standard **CosNaming::NamingContext** interface, via **OSIMgmt::NamingContext** interface. This means that every managed object exposes operations defined in the **CosNaming::NamingContext** interface. However, only the **resolve** and **list** operations may be invoked by CORBA managers. CORBA managers that invoke any of the other operations in the **CosNaming::NamingContext** interface should receive the NO_PERMISSION exception. Note that this restriction does not apply to CORBA agents that may use the other **CosNaming::NamingContext** operations to register new CORBA managed objects.

The CORBA Naming Service under the CORBAservices heading specifies a transitive rule for naming resolution:

ctx->resolve (C1;C2;...;Cn-1;Cn) =
(ctx->resolve (C1;C2;...;Cn-1))->resolve (Cn)

Being able to resolve to a leaf in an agent naming tree implies being able to resolve to the same object from any intermediate object in the naming tree, using a relative name. Given that the naming tree and the containment tree are the same in OSI management, this transitive rule mandates inheritance of **CosNaming::NamingContext** by every non-terminal element of the naming (i.e., containment) tree.

The inheritance of the **NamingContext** interface only implies interface inheritance, it does not imply inheritance from any standard off-the-shelf implementation of the CORBA Naming Service. In particular, all operations except “resolve” and “list” should raise the standard NO_PERMISSION exception when invoked by CORBA managers, and the “resolve” and “list” operations may have specialized implementations optimized for the lookup of OSI names.

Typically, the resolution of a name consists of forwarding the request down the tree through each context, the last one setting the response, and sending it back upward in the tree, to the client. This process can be long in the case of deep, distributed object trees. But the implementation is free to use any efficient algorithm given that it provides the same functionality, such as hash tables or delegation. Therefore, this specification does not limit the scalability or performance of applications implementing it.

Although the resolution of names is governed according to the CORBA naming transitive rule, the **NamingContext** tree doesn't have to match the OSI naming tree structure.

This implies that a managed object may raise the CannotProceed exception whenever the **resolve** operation is invoked on it, thus delegating name resolution to an alternative **CosNaming::NamingContext** object.

If we consider the previous rule, invoking

ctx->resolve (C1;C2;...;Cn-1)

may raise the CannotProceed exception and return ctx' and C1;C2;...;Cn-1 with it. The rule is then formulated as follows:

ctx->resolve (C1;C2;...;Cn-1;Cn) =
(ctx'->resolve (C1;C2;...;Cn-1))->resolve (Cn).

Table 3-4 describes the exceptions raised by the **resolve** operation(s) as they apply in OSI contexts. Note that this description complies with the description given for the standard **CosNaming** service in the Naming Service specification, and the one given generically in Section 2.1.2, "The JIDM::ProxyAgent Interface," on page 2-4.

Table 3-4 Exceptions Raised by the OSI Resolve Operations

Exception Raised	Description
NotFound	Indicates the name does not identify an existing managed object; this is equivalent to the OSI NoSuchObjectInstance error code.
CannotProceed	Indicates that implementation of the resolve operation has given up for some reason. However, if the **CosNaming::NamingContext** reference contained in the exception is not nil, the client may be able to continue the operation using the returned name. If the exception contains a nil **CosNaming::NamingContext** reference, then the situation is not recoverable, which may happen, for example, in situations such as those that cause the OSI ProcessingFailure error code.

Exception Raised	Description
InvalidName	Indicates the name is invalid. In OSI management it can occur in at least the following cases: • A name, in its OSI form (X711CMI::ObjectInstance), does not contain a valid value as defined in X.720; this is equivalent to the OSI InvalidObjectInstance error code. • A name, in its IDL form (**CosNaming::Name**), does not contain a valid value as defined in Section 3.1.1, "The OSIMgmt::LName Interface," on page 3-10. • An object class, provided in either resolve_with_intf or resolve_osi_name, does not exist; this is equivalent to the OSI NoSuchObjectClass error code. • The managed object named in either resolve_with_intf or resolve_osi_name does not support the object class provided; this is equivalent to the OSI ClassInstanceConflict error code. • The name has a 0 length.

3.1.4.2 Description of the ManagedObject attributes and operations

Inherited operations from CosLifeCycle::LifeCycleObject

The **OSIMgmt::ManagedObject** interface inherits from the standard **CosLifeCycle::LifeCycleObject** interface. This means that every managed object exposes the operations defined in the **CosLifeCycle::LifeCycleObject** interface. Specifically, the following semantics for the operations are specified:

- The **copy** operation performs similarly to a **CosLifeCycle::GenericFactory::create_object** operation. The object in which the operation is invoked acts as the "reference object" for the **create_object** operation (it is automatically inserted into the creation criteria parameter).
- The **move** operation is not appropriate in OSI management environments, so if invoked it should raise the NotMovable exception.
- The **remove** operation deletes the object from the managed domain. Note that deletion of a managed object may cause deletion of its descendants, or might be forbidden if the object has descendants, if such behaviors have been defined. If, for whatever reason, the object could not be destroyed, the NotRemovable exception will be raised.

The object_name attribute

This read-only attribute gives access to the name of the managed object in IDL form.

CMIS operations

OSIMgmt::ManagedObject objects support operations that enable CORBA Managers to operate upon selected descendants of the managed object. These operations are referred to as scoped operations, and provide a similar mechanism to that provided by **OSIMgmt::ProxyAgent** objects.

A detailed description about CMIS operations is presented in the Section 3.1.6, “Description of CMIS Operations,” on page 3-33.

Generic multi-attribute operations

Besides providing scoped operations (that might affect several managed objects), the **OSIMgmt::ManagedObject** interface also exports operations to manipulate several attributes of the managed object at the same time. Specifically, operations to get the values associated to multiple attributes (**get_attributes**) and to set the values of multiple attributes (**set_attributes**) are provided. These operations are synchronous (i.e., they block until the response is available).

The arguments of these two operations have the same meanings as described in Section 3.1.6, “Description of CMIS Operations,” on page 3-33. The return values represent the attribute values obtained from the managed object.

The perform_action operation

This operation provides a generic mechanism to invoke an action on the managed object. Any action supported by the specific managed object being accessed might be invoked in this way. The action invocation takes two arguments, with the same meanings as described in Section 3.1.6, “Description of CMIS Operations,” on page 3-33. The return value corresponds to the return type appropriate for the invoked action.

In cases where the managed object returned multiple replies to a single action invocation, the UsingMR exception will be raised. See Section 3.1.9, “Handling ACTIONs with multiple replies,” on page 3-45 for more information on actions with multiple replies.

The delete_mo operation

The **OSIMgmt::ManagedObject** includes a **delete_mo** operation that enables the deletion of a managed object, without the need to specify any scoping, filtering, synchronization, and access control arguments. Note that deletion of a managed object may cause deletion of its descendants, if such behavior has been defined. This call is synchronous (i.e., it blocks until the corresponding managed object in the managed domain has effectively been deleted).

3.1.5 The OSIMgmt::ManagedObjectFactory Interface

An **OSIMgmt::ManagedObjectFactory** can be used to create managed objects of more than one type. As a consequence of this, operations exposed through the **OSIMgmt::ManagedObjectFactory** interface always receive the name of the interface as an input argument.

References to objects exporting the **OSIMgmt::ManagedObjectFactory** interface are typically located by specifying the name of this interface in a "factory interface" component in the criteria used to find factories.

```
struct AttributeValue {
    ASN1_ObjectIdentifier attribute_id;
    ASN1_DefinedAny value;
};
typedef sequence<AttributeValue> AttributeValueSeq;

interface ManagedObjectFactory {

ManagedObject create (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name object_name,
                    in ManagedObject reference_object,
                    in AttributeValueSeq requested_attribute_values
) raises (CREATE_ERRORS);

ManagedObject create_with_auto_naming (
                    in CORBA::ScopedName interface_name,
                    in ManagedObject reference_object,
                    in AttributeValueSeq requested_attribute_values
) raises (CREATE_ERRORS);

ManagedObject create_subordinate (
                    in CORBA::ScopedName interface_name,
                    in CosNaming::Name superior_name,
                    in ManagedObject reference_object,
                    in AttributeValueSeq requested_attribute_values
) raises (CREATE_ERRORS);
};
```

Three operations are exposed trough the **OSIMgmt::ManagedObjectFactory** interface.

1. The **create** operation, which basically comprises all the parameters required to create a managed object using OSI Management principles.
2. The **create_with_autonaming** operation, which leaves to the **OSIMgmt::ManagedObjectFactory** the responsibility to assign a valid name to the new managed object.
3. The **create_subordinate** operation that creates a new managed object as subordinate of an existing object.

All three operations may receive a reference to an existing managed object whose state is copied into the state of the new managed object. That reference may be nil, in which case no reference object is considered.

3.1.6 Description of CMIS Operations

Application of scoped operations involves two phases: scoping and filtering. The base managed object of a scoped operation is defined as the root of the subtree of managed objects to which scoping and filtering is going to be applied.

Scoping entails the identification of those descendant(s) of the base managed object to which a filter is to be applied. Filtering entails the application of a set of tests to each member of the set of previously scoped descendants to extract the subset of those objects that satisfy the filter. The operation is then applied to all the objects in the subset of scoped descendants that satisfy the filter.

As a result of these scoped operations, multiple responses to a single request may happen. This type of interaction is not possible in CORBA, and therefore a callback mechanism is used by means of registering callback objects in the manager application that are the ones responsible to receive the responses (in either asynchronous or deferred synchronous modes). These callback objects are referred to as "Handlers," and are described in Section 3.1.7, "The OSIMgmt::LinkedReplyHandler, EndOfRepliesHandler, and MultipleRepliesHandler Interfaces," on page 3-38 (asynchronous handlers) and in Section 3.1.8, "The OSIMgmt::BufferedRepliesHandler Interface," on page 3-42 (deferred synchronous handlers).

3.1.6.1 Behavior common to all scoped operations

To determine the base managed object applicable to scoped operations, a different mechanism is used by **OSIMgmt::ProxyAgent** objects and by **OSIMgmt::ManagedObject** objects.

In cases of **OSIMgmt::ProxyAgent** objects, the following parameters are used to determine the base managed object:

- **interface_name**: Specifies the fully scoped name of the interface exported by the base managed object of the scoped operation.
- **object_name**: Specifies the IDL name of the base managed object of the scoped operation.

In cases of **OSIMgmt::ManagedObject** objects, the base managed object for the scoped operation is the managed object itself; therefore, these two parameters are not needed.

The following parameters are passed both to **OSIMgmt::ProxyAgent** objects and to **OSIMgmt::ManagedObject** objects when invoking scoped operations to control the set of managed objects to which the operation is to be applied (scope and filter), and to specify interacion characteristics (synchronization and access control).

- **scope**: Indicates the subtree, rooted at the base managed object that is to be searched. The different types of scoping that may be performed are:
  - the base object alone
  - the n-th level subordinates of the base object
  - the base object and all of its subordinates down to and including the n-th level
  - the base object and all of its subordinates (whole subtree)
- **filter**: Specifies the set of assertions that defines the filter test to be applied to the set of managed object(s) that result from applying the scope. Multiple assertions may be grouped using the logical operators AND, OR, and NOT. Each assertion may be a test for equality, ordering, presence, or set comparison. Assertions about the value of an attribute are evaluated according to the matching rules associated with the attribute syntax. If an attribute value assertion is present in the filter and that attribute is not present in the scoped managed object, then the result of the test for that attribute value assertion is evaluated as FALSE. The managed object(s) for which the filter test evaluates to TRUE is selected for the application of the operation. If the filter is not specified, all of the managed objects included by the scope are selected.
- **synchronization**: Indicates how the invoking operation should be synchronized across the selected object instances. Two ways of synchronizing a series of operations are defined, as specified in [X710]:
  - Best effort (**X711CMI::CMISSyncType(bestEffort)**): this synchronization only requires that all managed objects selected for the operation are requested to perform it, without any guarantee regarding the success of such request.
  - Atomic (**X711CMI::CMISSyncType(atomic**)): If the base managed object alone is selected for the operation, this parameter is ignored. Atomic synchronization requires that all managed objects selected for the operation are checked to ascertain if they are able to successfully perform the operation. If one or more are not able to successfully perform the operation, then none perform it; otherwise, all perform it.
- **access_control**: Information to be used as input to access control functions. This parameter is optional, and its type is **X711CMI::AccessControlTypeOpt**.
  - If present, this access control parameter is to be used in the current invocation.
  - If absent, the default access control parameter specified at **OSIMgmt::ProxyAgent** creation time (if any) should be used.
  - If neither was specified, then no access control information should be used.

The **reply_handler** and **end_of_replies_handler** parameters are passed both to **OSIMgmt::ProxyAgent** objects and to **OSIMgmt::ManagedObject** objects when invoking scoped operations to specify the callback objects to use to receive responses to the scoped operation:

The **reply_handler** and **end_of_replies_handler** parameters specify the object references where the replies to the scoped operation are to be returned (for more details see Section 3.1.7, “The OSIMgmt::LinkedReplyHandler, EndOfRepliesHandler, and MultipleRepliesHandler Interfaces,” on page 3-38).

If these callback objects become unreachable during the process of performing a scoped operation (due to communications problems, or to the explicit deletion of the callback objects), the managed domain implementation should interrupt processing of the scoped operation, if possible. Specifically, in the case of **cmis_get** or **scoped_get** operations this situation is the way to inform the managed domain that an on-going **Get** operation must be canceled.

In cases where unconfirmed operations are being requested (for example, an unconfirmed scoped set operation), both parameters should be specified as **nil** object references; in these cases, no responses will be received.

If the **reply_handler** parameter is specified, but the **end_of_replies_handler** is specified as a **nil** object reference, then the behavior of the invoked operation is slightly different. The operation invocation blocks until all responses have been received through the **reply_handler** object passed in the call, that is, the operation itself becomes the end of replies indication. This allows sequential scoped operations to be invoked without extra coding.

Note – The ORB might timeout such invocations if they take too long. Configure your ORB timeout appropriately.

- If the **reply_handler** parameter is specified as a **nil** object reference, but the
  - **end_of_replies_handler** is not, or
- if both are **nil**, but the
  - operation does not accept unconfirmed operation,

then the CORBA standard exception BAD_PARAM is raised.

The results of invoking a scoped operation are returned through the **OSIMgmt::LinkedReplyHandler** object specified when the scoped operation was invoked. Information associated with each reply is passed by invoking the **send_reply** operation. If an error occurs during the process, the **send_mo_error** or **send_subtree_error** operations are called instead, depending on the type of error that occurred.

For more details, see Section 3.1.7, “The OSIMgmt::LinkedReplyHandler, EndOfRepliesHandler, and MultipleRepliesHandler Interfaces,” on page 3-38.

The GET operations

Besides the parameters specified above, the scoped GET operations (**cmis_get**, **scoped_get**) carry an extra parameter **attribute_id_list**. This parameter is a sequence of **ASN1_ObjectIdentifiers** corresponding to the **Attributes** (or **Attribute Groups**) to be retrieved by the operation. Note that to fill this parameter the JIDM Specification Translation process defines IDL constants of the right type and contents, to facilitate this process (see XoJIDM, Appendix A, “References” for details). If the sequence is empty, this is interpreted as if the complete list of attributes would have been requested.

The SET operations

The **set** operation may perform one of the following modifications to attributes.

- Replace the values of specified attributes with supplied values. The replacement is exact, unless the attribute definition explicitly states otherwise.
- Replace the values of specified attributes with default values.
- Add or remove members to set-valued attributes that are defined to allow the addition or removal of members.

To be able to specify the above modifications, the scoped SET operations (**cmis_set**, **scoped_set**) carry the following extra parameter:

- **modification_list** - this parameter contains a set of attribute modification specifications, each of which contains:
  - **attribute_id**: the registration **ASN1_ObjectIdentifer** of the attribute or attribute group whose value(s) is to be modified; an attribute group identifier shall only be specified when the set to default modify operator is specified.
  - **attribute_value**: the value(s) to be used in the modification of the attribute; the use of this attribute is defined by the modify operator. This parameter is optional when the set to default modify operator is specified and if supplied, shall be ignored. If no value is to be passed, then a CORBA **any** with **tc_kind** equal to **tk_null** should be passed.
  - **modify_operator**: the way in which the attribute value(s) (if supplied) is applied to modify the attribute.

If the set of attribute modifications is empty, then no modification is requested.

The ACTION operations

The **action** operation requests the managed objects to perform the specified action and to indicate the result of that action. With respect to confirmations, action operations may be defined to always require confirmation or to allow the invoker to request a confirmation or not.

Action operations may be defined to generate more than one response per managed object that performs the operation.

As actions are generic by definition, the scoped ACTION operations (**cmis_action**, **scoped_action**) carry the following extra parameters:

- **action_name**: This argument specifies the registration ASN1_ObjectIdentifier of the action to be performed. Note that to fill this parameter, the JIDM Specification Translation process defines IDL constants of the right type and contents to facilitate this process (see XoJIDM, Appendix A, “References” for details).
- **action_info**: This argument carries the specific parameters that correspond to the action, that is, the IDL type that is the mapping of the information syntax for this action. If the action has no information syntax, a CORBA **any** with **tc_kind** equal to **tk_null** will be passed in this argument.

The DELETE operations

The scoped DELETE operations (**cmis_delete**, **scoped_delete**) are used to request the managed objects selected as a result of applying the scoping and filtering arguments to delete themselves. These operations do not require extra parameters except the common ones.

The CREATE operations

These operations are only available through the **OSIMgmt::ProxyAgent** interface (that is, they are not available from the **OSIMgmt::ManagedObject** interface) and provide another mechanism to create objects in the managed domain.

These operations are not scoped, that is, they only affect one object (the one being created), and therefore do not follow the common behavior outlined above.

Two flavors of the same operation are provided, one synchronous (**cmis_create_sync**) and another asynchronous (**cmis_create**).

Both flavors of the **cmis_create** operation use the same input parameters:

- **interface_name**: Specifies the fully scoped name of the interface to be exported by the newly created object.
- **creation_kind**: Specifies the type of creation mechanism to be used, and identifies the use of the next parameter. The possible values are:
  - **simple**: Create the object named in the following parameter.
  - **autonaming**: Ignore the contents of the following parameter, and automatically assign a name for the newly created object.
  - **subordinate**: The name specified in the next parameter is the name of the superior object from the one to be created.
- **object_name**: Specifies the IDL name of the managed object to be created (if **creation_kind** is **simple**) or the name of the superior object (if **creation_kind** is **subordinate**). If case **creation_kind** is **autonaming**, the contents of this parameter are ignored.
- **access_control**: This parameter is information to be used as input to access control functions, it is optional, and its type is **X711CMI::AccessControlTypeOpt**. If present, then this access control parameter is to be used in the current invocation. If absent, then the default access control parameter specified at **OSIMgmt::ProxyAgent** creation time (if any) should be used. If neither was specified, then no access control information should be used.
- **reference_object**: Indicates the name of a managed object that should be used as a reference when creating the new object.
- **req_attribute_values**: Specifies a set of attribute values to be assigned at object creation time.

Both operations differ in the way they receive responses:

- **cmis_create**: Performs the operation in an asynchronous way, returning the one and only result into the **OSIMgmt::LinkedReplyHandler** object passed as input parameter. Note that this is an exception to the process described in Section 3.1.7, “The OSIMgmt::LinkedReplyHandler, EndOfRepliesHandler, and MultipleRepliesHandler Interfaces,” on page 3-38. There is always one and only one response sent to the input **LinkedReplyHandler** object, and there is no call to the **end_of_replies** method. The rest of the processing, including the use of the different methods in the **LinkedReplyHandler** interface, follows exactly what is explained in Section 3.1.7, “The OSIMgmt::LinkedReplyHandler, EndOfRepliesHandler, and MultipleRepliesHandler Interfaces,” on page 3-38.
- **cmis_create_sync**: Given that there is only one response possible, a synchronous mechanism is also provided for this operation. In this case, there are several output parameters carrying the result of the operation:
  - **created_interface_name**: Interface supported by the newly created object.
  - **created_object_name**: Name of the created object.
  - **creation_time**: Time when the new object was created (optional).
  - **created_attribute_values**: Values assigned to the attributes of the new object.

  Error responses are returned as exceptions in this case.

3.1.7 The OSIMgmt::LinkedReplyHandler, EndOfRepliesHandler, and MultipleRepliesHandler Interfaces

The **LinkerReplyHandler/EndOfRepliesHandler** are facilities that allow managed object(s)/proxy agent(s) to send multiple replies to a single scoped operation, using the asynchronous model. The deferred synchronous model is implemented using the **BufferedRepliesHandler** (see Section 3.1.8, “The OSIMgmt::BufferedRepliesHandler Interface,” on page 3-42)..

```
interface LinkedReplyHandler {
    void send_reply (
                in CORBA::ScopedName object_interface;
                in CosNaming::Name object_name;
                in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                in any reply_info
);

    void send_mo_error (
                in CORBA::ScopedName object_interface,
                in CosNaming::Name object_name,
                in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                in short error_code,
                in any error_info
);

    void send_subtree_error (
                in CORBA::ScopedName object_interface,
                in CosNaming::Name object_name,
                in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                in short error_code,
```

```
                    in any error_info
    );
};

interface EndOfRepliesHandler {
        void end_of_replies ( );
};

interface MultipleRepliesHandler : LinkedReplyHandler,
        EndOfRepliesHandler {};
```

These interfaces are used as arguments for all scoped operations in **OSIMgmt::ProxyAgent** and **OSIMgmt::ManagedObject** interfaces.

The scoped operations will be invoked on each managed object within the specified scope, which passes the filter condition. The corresponding reply will be sent separately to the **OSIMgmt::LinkedReplyHandler** object whose reference was passed when the scoped operation was invoked.

After all invocations to **send_reply**, **send_mo_error** and **send_subtree_error** have returned, the **end_of_replies** operation is invoked, indicating the complete finalization of the process. Note that with this behavior race conditions are avoided.

LinkedReplyHandler/MultipleRepliesHandler objects may be implemented both as local objects (in the client address space) or as remote objects (accessed via CORBA invocations) either in the gateway/CORBA agent/CORBA managed object address space or in a separate CORBA service process.

3.1.7.1 Descriptions of the LinkedReplyHandler operations

Common arguments to the LinkedReplyHandler operations

The following arguments are common to all **OSIMgmt::LinkedReplyHandler** operations:

- **object_interface**: This parameter specifies the fully scoped name of the interface exported by the managed object that generated the current reply.
- **object_name:** This parameter specifies the IDL name of the managed object that generated the current reply.

Using an empty string in the **object_interface** and a 0 length sequence in the **object_ name** arguments refers to the base object of the corresponding scoped operation.

- **current_time**: Specifies the time at which the response was generated. This parameter is optional and might not be specified.
- **reply_info/error_info**: Carries the information associated with the current reply. If no value is to be passed, then a CORBA **any** with **tc_kind** equal to **tk_null** will be passed.

The send_reply operation

The **send_reply** operation is used to pass information associated with each reply from managed objects involved in a scoped operation.

The **reply_info** argument is a CORBA **any**, and its contents are different depending on the scoped operation that was performed, as follows:

Table 3-5 Contents of reply_info parameter to send_reply

scoped operation	Type carried in the reply_info parameter
ProxyAgent::cmis_create	X711CMI::CreateResultAttributeListType
ProxyAgent::cmis_get ManagedObject::scoped_get	X711CMI::GetResultAttributeListType
ProxyAgent::cmis_set ManagedObject::scoped_set	X711CMI::SetResultAttributeListType
ProxyAgent::cmis_action ManagedObject::scoped_action	IDL mapped type corresponding to the ACTION REPLY SYNTAX; if the action has no reply syntax, CORBA any with tc_kind equal to tk_null
ProxyAgent::cmis_delete ManagedObject::scoped_delete	CORBA any with tc_kind equal to tk_null

The send_mo_error operation

The **send_mo_error** operation indicates that an error has been found for a managed object. The **error_code** argument indicates the type of error that has happened. The **error_info** argument is a CORBA **any** whose contents might be different depending on the scoped operation that was performed and the error that occurred.

Also, there are certain cases where the **send_mo_error** operation does not provide any additional information, in which case an **any** with **tc_kind** equal to the **tk_null** is passed as value of the **error_info** parameter.

Table 3-6 Contents of error_code and error_info parameters to send_mo_error

scoped operation	error_code	Type in the error_info parameter
ProxyAgent::cmis_get ManagedObject::scoped_get	4	X711CMI::GetListErrorGetInfoListType
ProxyAgent::cmis_set ManagedObject::scoped_set	5	X711CMI::SetListErrorSetInfoListType
ProxyAgent::cmis_action ManagedObject::scoped_action	16	X711CMI::ActionErrorInfoType
ProxyAgent::cmis_delete ManagedObject::scoped_delete	17	X711CMI::DeleteErrorDeleteErrorInfoType

scoped operation	error_code	Type in the error_info parameter
all in case of ProcessingFailure	8	X711CMI::SpecificErrorInfoType, or CORBA any with tc_kind equal to tk_null

The send_subtree_error operation

The **send_subtree_error** operation indicates a fatal error in a certain managed object subtree. The client should not expect any further replies from objects in the affected managed object subtree. The interface and name of the base managed object of the subtree that experienced the fatal error are passed as arguments to this operation.

The **error_code** argument indicates the type of error that has happened, and the **error_info** argument is a CORBA **any**, whose contents might be different depending on the error that occurred.

Also, there are certain error cases where there is no additional information, in which case an **any** with **tc_kind** equal to the **tk_null** is passed as value of the **error_info** parameter on the **send_subtree_error** operation.

Table 3-7 Contents of error_code and error_info parameters to send_subtree_error

Error condition	error_code	Type in the error_info parameter
AccessDenied	1	empty
ClassInstanceConflict	2	empty
ComplexityLimitation	3	X711CMI::ComplexityLimitationType, or empty
GetListError	4	X711CMI::GetListErrorGetInfoListType
SetListError	5	X711CMI::SetListErrorGetInfoListType
InvalidArgumentValue	6	X711CMI::InvalidArgumentValueType
OperationCancelled	7	empty
ProcessingFailure	8	X711CMI::SpecificErrorInfoType, or empty
InvalidFilter	9	X711CMI::CMISFilterType
InvalidScope	10	X711CMI::ScopeType
SyncNotSupported	11	X711CMI::CMISSyncType
NoSuchAction	12	X711CMI::NoSuchActionType
NoSuchArgument	13	X711CMI::NoSuchArgumentType
NoSuchObjectClass	14	X711CMI::ObjectClassType
NoSuchObjectInstance	15	X711CMI::ObjectInstanceType

Error condition	error_code	Type in the error_info parameter
DuplicateManagedObjectInstance	18	X711CMI::ObjectInstanceType
InvalidAttributeValue	19	X711CMI::AttributeType
InvalidObjectInstance	20	X711CMI::ObjectInstanceType
MissingAttributeValue	21	X711CMI::MissingAttributeValueType
NoSuchAttribute	22	X711CMI::AttributeIdType
NoSuchReferenceObject	23	X711CMI::ObjectInstanceType
MistypedOperation NoSuchEventType NoSuchInvokeId	-1	empty (these errors cannot happen in any scoped operation)
communication failure	-2	empty
ROSE rejection	-3	implementation specific
others unknowns	-4	implementation specific

3.1.7.2 Descriptions of the EndOfRepliesHandler operations

The end_of_replies operation

The **end_of_replies** operation indicates that no more replies are going to be received from the managed objects involved in a scoped operation; therefore, signaling the complete finalization of the process. This operation is invoked after all invocations to **send_reply**, **send_mo_error**, and **send_subtree_error** from objects within the scope/filter parameters specified in the scoped operation have returned. Note that with this behavior race conditions are avoided.

3.1.8 The OSIMgmt::BufferedRepliesHandler Interface

The **OSIMgmt::BufferedRepliesHandler** is a facility that allows client programmers to use a deferred synchronous model to retrieve responses for multiple replies in an on-demand basis. This complements the fully asynchronous model provided by the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** interfaces.

```
struct Reply {
        CORBA::ScopedName  object_interface;
        CosNaming::Name  object_name;
        X711CMI::ASN1_GeneralizedTimeOpt  current_time;
        any reply_info;
};
typedef sequence<Reply> ReplyList;

interface RepliesIterator  {
```

```
        exception MoError {
            CORBA::ScopedName object_interface;
            CosNaming::Name object_name;
            X711CMI::ASN1_GeneralizedTimeOpt current_time;
            short error_code;
            any error_info;
        };

        exception SubtreeError {
            CORBA::ScopedName object_interface;
            CosNaming::Name object_name;
            X711CMI::ASN1_GeneralizedTimeOpt current_time;
            short error_code;
            any error_info;
        };

        boolean  get_reply (out Reply r) raises (MoError, SubtreeError);
        boolean  get_n_replies (in unsigned long how_many, out ReplyList r_list)
                    raises (MoError, SubtreeError);

        boolean  finished (out unsigned long num_pending);
        void destroy ();
};

interface BufferedRepliesHandler : MultipleRepliesHandler, RepliesIterator {};
```

Note that the **OSIMgmt::BufferedRepliesHandler** interface is a pure extension of the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** interfaces. Therefore, objects exporting this interface can be passed in to operations that take **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** as parameters.

The use model for this interface is that of an Iterator, from the client's perspective. That is, the only operations a client should use are those defined in the **RepliesIterator** interface. The other operations are directly invoked from the Managed Domain, as a result of a scoped operation or action with multiple replies.

These objects can be implemented either in the manager side or in the managed domain side of an interaction, or even provided by an external service.

3.1.8.1 Descriptions of BufferedReplyHandler types and operations

The Reply type

The **Reply** type is the structure to hold one reply; the different fields in the structure match those in the signature of the **send_reply** operation of the **LinkedReplyHandler** interface.

The MoError exception

The MoError exception corresponds to the **send_mo_error** operation of the **LinkedReplyHandler** interface, and the types and values carried by the exception match exactly the parameters of the operation.

The SubtreeError exception

The SubtreeError exception corresponds to the **send_subtree_error** operation of the **LinkedReplyHandler** interface, and the types and values carried by the exception match exactly the parameters of the operation.

The RepliesIterator interface

The **RepliesIterator** is, as its name indicates, an Iterator type interface, where there are operations to access a list of items in the iterator in an ordered manner. Items are accessed once, and only once, regardless of the operation used to access them. The Iterator cannot be backed (re-access) or re-initiated.

Note that, in multi-threaded environments, where responses are retrieved from multiple threads (or even from multiple processes), each response will only be received once, so care must be taken in these circumstances.

The get_reply operation

The **get_reply** operation blocks until the next reply is available, returning it when ready. It returns true when the operation has actually retrieved a response, it returns false when there are no more responses pending to be received (i.e., the **end_of_replies** has been received). In this case, the value returned in the Reply is undefined (must be ignored). Once this operation has returned false, all subsequent invocation to it will also return false. In case an error response is the first to be returned (because it was buffered or received while blocked), the corresponding exception is raised.

The get_n_replies operation

The **get_n_replies** call blocks until **how_many** replies are available, an error is received or the iterator reaches its end, whatever happens first. The **ReplyList** will contain at most **how_many** non-error replies. Specific implementations may impose an appropriate maximum number for the **how_many** parameter to prevent excessive consumption of resources.

If an error is received, the operation will return all valid replies up to but not including the error, unless the error is the first reply, in which case the corresponding exception is raised. When the error is not returned by this operation, it remains in the buffer as the first response to be retrieved, and therefore the next call to **get_reply** or **get_n_replies** will raise the exception corresponding to the error.

The operation returns false if there are no more replies to be pulled out and the end of the iterator has been reached, true; otherwise, this means that if this operation is invoked after the Iterator has reached its end, then an empty list is returned and the return value is false.

Note that the semantic of the iterator is orthogonal to the way replies and errors are received through the **LinkedReplyHandler** interface and it allows a programming model where errors are handled separately from normal replies.

The finished operation

The **finished** operation returns true if the Iterator has completed its background work (and therefore knows about ALL responses) and false if the operation is still under way (that is, there might be more responses unknown to the iterator at this time). In both cases, the number of replies immediately available for retrieval is returned (including pending error responses). This operation does not block.

This operation can be used to work in polling (non blocking) mode with the Iterator, as the client can always ask for what it knows the Iterator already has received/processed. This polling mechanism should only be used from a single thread.

The destroy operation

The **destroy** operation destroys the iterator; any calls to the iterator invoked after this would return the OBJECT_DOES_NOT_EXIST standard exception.

3.1.9 Handling ACTIONs with multiple replies

Where actions have a reply syntax, objects have the option of using multiple replies (i.e., returning a sequence of PDUs, each of the type given in the reply syntax, containing part of the reply). Multiple replies allow data to be returned as it becomes available and have been used for monitoring progress.

While these are not widely used and could easily be replaced by notifications, it was necessary to provide this capability with the same functionality as for generic interfaces.

When actions that generate multiple replies from a single object are invoked through the scoped operations interfaces (**cmis_action** or **scoped_action**) no special action has to be taken, as the mechanism to process multiple replies is already in place.

Actions may also be invoked using IDL operations generated by the GDMO to IDL translation process or using the **OSIMgmt::ManagedObject::perform_action** operation. These interfaces are synchronous (the response is the return value from the operation); therefore, not allowing the reception of multiple replies.

For these cases, an additional user exception may be raised by the action call, in the event that multiple replies to the same request are generated. This is the UsingMR exception.

The UsingMR exception carries one parameter, a reference to an **OSIMgmt::RepliesIterator** object described in Section 3.1.8, “The OSIMgmt::BufferedRepliesHandler Interface,” on page 3-42, that is provided by the managed domain and should be used by the manager application to retrieve all responses to the action. In case the managed domain does not provide this mechanism, and yet the multiple replies are generated, then a **nil** object reference might be passed in the exception, and the manager should invoke the operation using the scoped operations.

```
exception UsingMR
    { RepliesIterator replies_iterator; };
```

3.1.10 The OSIMgmt::LocalRoot interface

The term ‘local orphan managed objects,’ associated with an OSI managed object domain, is used to designate those managed objects of which the superior objects are located in a different OSI managed object domain.

In every CORBA-based OSI managed object domain there will exist a CORBA object that plays the role of a ‘local root.’ This local root object will act as the superior of all local orphan managed objects in the application (i.e., it will hold references to local orphan managed objects), and exports the **OSIMgmt::LocalRoot** interface.

In the case that a local orphan object is created in a certain managed domain, the local root object for that domain must be notified that a new subordinate has been created.

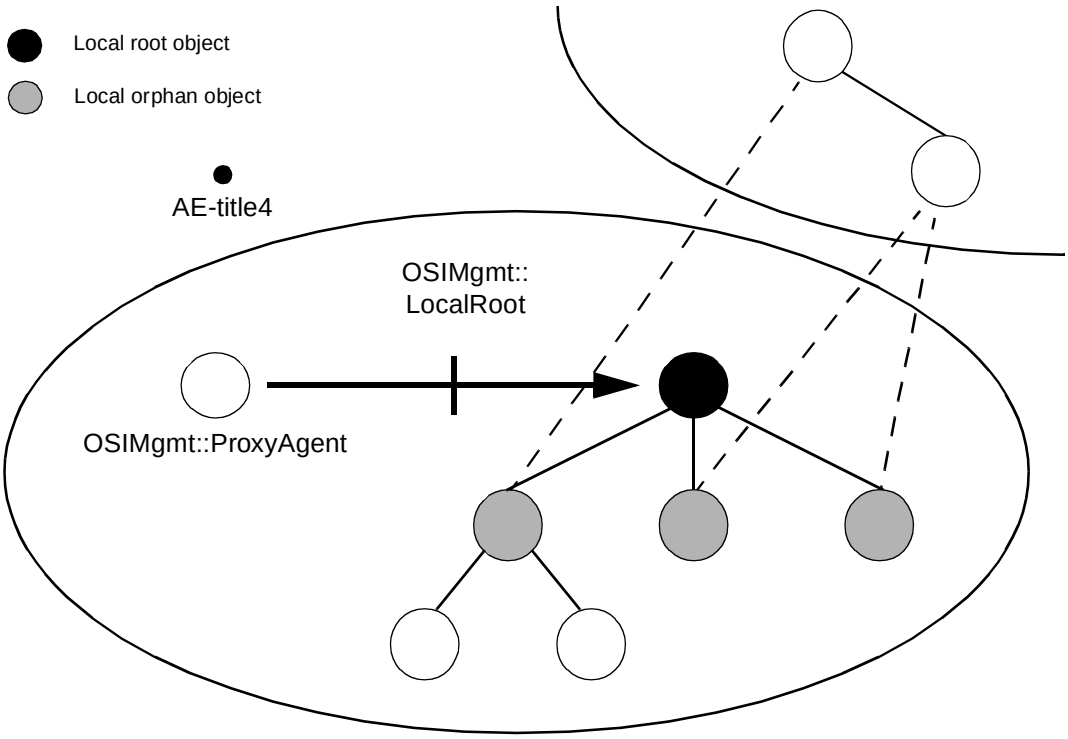

Figure 3-4 Dealing with Local Orphan Managed Objects

A reference to the local root object is maintained by the **JIDM::DomainPort** object associated to the managed object domain. The **JIDM::DomainPort** object passes this reference to every **OSIMgmt::ProxyAgent** object it creates.

The local root object is a managed object that exports the **OSIMgmt::LocalRoot** interface.

```
typedef sequence<ManagedObject> ManagedObjectSeq;

interface LocalRoot : ManagedObject {
    exception NoDescendants {};
```

```
        // list all local orphan managed objects:
        ManagedObjectSeq list_orphans ( );

        // list of local orphans that are descendants of the object
        // whose name is specified:
        ManagedObjectSeq
        list_orphan_descendants (in CosNaming::Name object_name)
                raises (NoDescendants);
};
```

Note that the reference returned by the **get_domain_naming_context** operation points to the system managed object. A reference to the root managed object supporting the **NamingContext** interface will be bound under the initial **CosNaming::NamingContext** of a managed object domain (typically corresponding to the system managed object). Other **NamingContexts** that exist in the domain may contain that binding as well, thus, allowing resolution of **DistinguishedNames** in their context.

3.2 Programming Model

This section is provided as information only, and does not represent a normative part of this specification. In this section, different scenarios are described where the use of this specification will be clarified. This should be considered as a high level tutorial on some potential uses of the JIDM model for OSI management.

3.2.1 Programming Semantics

CORBA manager programs create and invoke operations on managed objects in the same way they create and invoke operations on ordinary CORBA objects located in the same CORBA domain. Analogously, they receive events supplied by managed objects as if they were ordinary CORBA objects supplying events to an event channel located in the CORBA domain. Whether this actually happens or not is transparent to the CORBA manager program.

This concept of transparency is specifically supported by the fulfillment of the semantic rules presented in Section 2.1.1, “JIDM Managed Objects,” on page 2-3.

3.2.2 Creating Managed Objects

Creating a managed object implies to perform the list of actions described in Section 2.1.2, “The JIDM::ProxyAgent Interface,” on page 2-4:

1. Obtain a reference to an **OSIMgmt::ProxyAgent** object that enables access to the domain where the managed object is going to be created.
2. Obtain a reference to the initial **CosLifeCycle::FactoryFinder** located at the domain.

3. Invoke the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object to find a factory for the new managed object.

4. Select a factory among the several factory objects that may meet the criteria for finding factories passed to the **find_factories** operation.

5. Invoke an appropriate operation, exposed by the selected factory, to create the managed object.

Valid key values for finding factories in OSI Systems Management environments were described in section Section 2.1.2, “The JIDM::ProxyAgent Interface,” on page 2-4.

Figure 3-5 illustrates how, in a pure CORBA environment, manager objects will create a new OSI managed object.

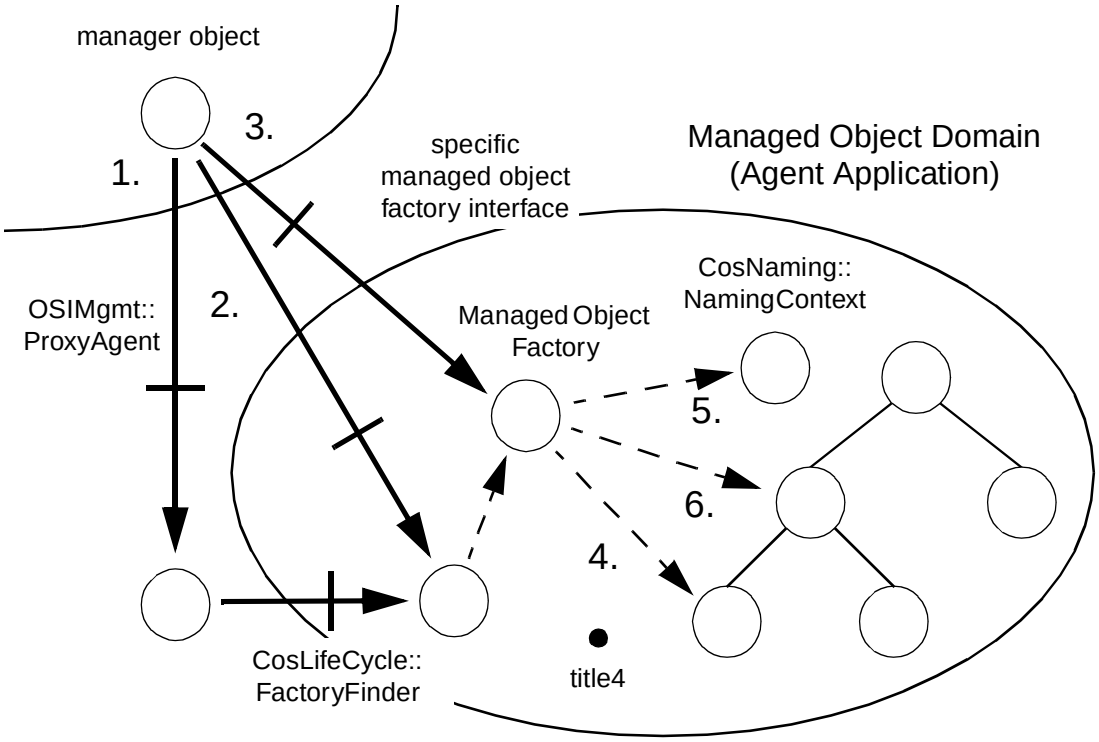

Figure 3-5 Creating an OSI managed object directly through CORBA

As with JIDM facilities, the **OSIMgmt::ProxyAgent** created as a result of establishing a connection to a CORBA managed object domain would typically hold references to the root **CosNaming::NamingContext** object and **CosLifeCycle::FactoryFinder** objects located at the domain. These steps are followed:

1. The CORBA manager invokes the **get_domain_factory_finder** operation exposed by the **OSIMgmt::ProxyAgent** object. As a result, a reference to the initial **CosLifeCycle::FactoryFinder** located at the domain being accessed is returned.

2. The CORBA manager object invokes the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object. As a result, a reference to a managed object factory is obtained and returned to the CORBA manager object that requested it.

3. The CORBA manager object invokes a suitable operation on the managed object factory using the CORBA object reference previously obtained. Typically, the CORBA manager will narrow this reference to a well-known managed object factory interface (see Section 3.1.2, “The OSIMgmt::ProxyAgent Interface,” on page 3-17).

4. The managed object factory creates the CORBA managed object and obtains a reference as a result.

5. The managed object factory binds the obtained reference with a name in the local root **CosNaming::NamingContext** object.

6. The managed object factory notifies the superior managed object that a new subordinate has been created.

7. Finally, if everything is all right, the managed object factory returns a reference to the CORBA manager object. Otherwise, it returns an exception.

Any superior managed object will hold a reference to every managed object which is a subordinate of it. That is why a superior managed object must be notified about creation of subordinate managed objects. Superior managed objects must hold references to subordinate managed objects in order to handle scoping and filtering as well as to handle deletions.

Different implementation approaches are possible for step 5:

- The **CosLifeCycle::Factory** perform the required actions to allow initialization of the object when it is first activated.
- The **CosLifeCycle::Factory** invokes an initialization operation exposed by the object. (The XoJIDM Working Group should discuss if this operation requires to be specified in the standard **OSIMgmt::ManagedObject** interface.)

Different implementation approaches are also possible for step 6:

- Every managed object exports the **CosNaming::NamingContext** interface and keeps name bindings associated to its subordinates (note that this would imply that the structure of the CORBA naming tree would be equivalent to the OSI naming tree (i.e., managed objects support the **resolve** operation and don’t raise the CannotProceed exception).
- The structure of the CORBA naming tree doesn’t match the structure of the OSI naming tree. For example, it is more plain and avoid deep nesting (note that this approach implies that managed objects support the **resolve** operation but may raise the CannotProceed exception).

and, finally, for step 7 the following implementation approaches are also valid:

- Make the factory object coincide with the superior of the new managed object (note that this implies a clear optimization).

- Make the factory object notify the superior object that a new child has been born by means of invoking an operation that the superior object exposes for this purpose. (The XoJIDM Working Group should discuss if this operation is going to be specified in the **OSIMgmt::ManagedObject** interface.)

In a CORBA managed object domain, propagation of operations is handled by the objects themselves: each object is responsible to forward the operation to its descendants. However, the way in which they perform the propagation is an implementation matter. Analogously, the mechanisms used to notify to the base managed object that all the replies have been sent is an implementation matter.

A simple way of implementing propagation of operations would consist in using recursion as illustrated in the pseudo-code listed below.

Recursive propagation of scoped actions

```
obj->scoped_action (scope, filter, sync, act, arg,
             replies_handler, end_handler)
::=
if <obj satisfies the filter> {
             // perform the action on the object and return the result:
             result = obj->act (...);
             replies_handler->send_reply (obj_intf, obj_name, result);
};

// create an OSIMgmt::EndOfRepliesHandler (end_subs) which will
// wait until all descentants notify they have finished propagation:
end_subs = ...;

// propagate action through descendants:
sub1->scoped_action (scope1, filter, sync, act, arg, replies, end_subs);
.....................................................................
subn->scoped_action (scopen, filter, sync, act, arg, replies, end_subs);
```

In the above example, the base managed object passes a reference to an object (**`end_subs`**) that will be responsible for:

1. Waiting until all subordinates of '**`obj`**' invoke the **`end_of_replies`** operation exposed by the '**`end_subs`**' object.
2. Trigger invocation of the **`end_of_replies`** operation exposed by the '**`end`**' object.

In the algorithm we have presented, invocation of a scoped operation returns immediately. The **OSIMgmt::ProxyAgent** object doesn't handle the end of replies but passes a reference to the object that will handle it. There is another possibility which consists in that invocation of a scoped operation blocks until all replies have been sent. In this case, the **OSIMgmt::ProxyAgent** object will be responsible for sending the last CMIP response, indicating the end of replies. By convention, this behavior is experimented whenever a nil object reference is passed as the **OSIMgmt::EndOfRepliesHandler** argument.

Note that synchronous invocations imply support for multi-threading in the CORBA/CMIP gateway process. (The **OSIMgmt::ProxyAgent** and the **OSIMgmt::LinkedReplyHandler** objects must concurrently execute.)

3.2.3 Invoking Operations on Single Managed Objects

Invoking an operation on a single managed object implies that the following actions are performed:

1. Obtain a reference to an **OSIMgmt::ProxyAgent** object that enables access to some domain of which the managed object is member.
2. Obtain a reference to the initial **CosNaming::NamingContext** located at the domain, by means of invoking the **get_domain_naming_context** operation exposed by the **OSIMgmt::ProxyAgent** object.
3. Construct the name that unequivocally identifies the managed object within the domain.
4. Invoke the **resolve** operation exposed by the initial **CosNaming::NamingContext** object located at the domain, thus obtaining a CORBA object reference pointing to the managed object.
5. Invoke the operation on the managed object.

Example code for invoking an operation on a managed object

The following example shows the code used to set the destination attribute exposed by an Event Forwarding Discriminator.

```
OSIMgmt::ProxyAgent_ptr agent;
OSIMgmt::LName local_name;
X721Att::DestinationType new_destinations;
.....

        // a reference to a OSIMgmt::ProxyAgent is obtained as a result of
        // establishing a connection to the managed object domain where
        // the EFD is located:

agent = ...;
.....

// a reference to the initial CosNaming::NamingContext object
// is obtained:

CosNaming::NamingContext_ptr ctx = agent->get_domain_naming_context ();

// the name of the EFD object is constructed:

local_name -> for_string_form ("2.9.3.2.1=(string)'MyEFD'");
.....

        // a reference to the EFD object is obtained and narrowed
```

```
                    // to the X711::eventForwardingDiscriminator interface:

CORBA::Object_ptr obj = ctx->resolve (local_name -> to_idl_form ());

X721::eventForwardingDiscriminator_ptr efd =
                                X711::eventForwardingDiscriminator::_narrow (obj);

                    // Finally, the destinationSet operation is invoked on
                    // the managed object:

efd -> destinationSet (new_destinations);
```

Note that the same result can be obtained without narrowing the reference returned by a **JIDM::ProxyAgentFinder** object to an **OSIMgmt::ProxyAgent** interface, when the **access_domain** operation was invoked.

Figure 3-6 illustrates how CORBA manager objects invoke operations on a single managed object in a pure CORBA environment.

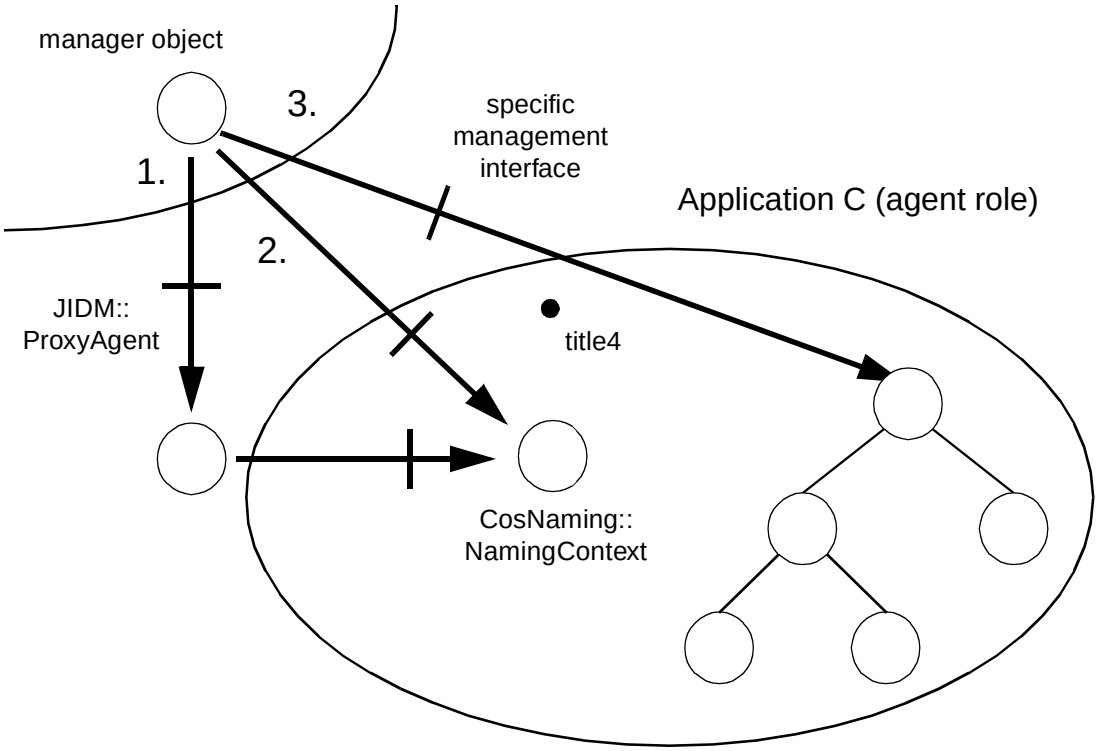

Figure 3-6 Invoking operations on a managed object directly through CORBA

As previously explained, the **OSIMgmt::ProxyAgent** created as a result of establishing a connection to a CORBA managed object domain would typically hold references to the root **CosNaming::NamingContext** object and **CosLifeCycle::FactoryFinder** object located at the domain. Thus, the following steps will be followed:

1. The CORBA manager object invokes the **get_domain_naming_context** operation exposed by the **OSIMgmt::ProxyAgent** object, in order to obtain a reference to the initial **CosNaming::NamingContext** object.

2. The CORBA manager object invokes the **resolve** operation exposed by the initial **CosNaming::NamingContext** object, passing the name of the managed object upon which it wants to operate. As a result of this, a CORBA object reference to the managed object is obtained and returned to the CORBA manager object that requested it.

3. The CORBA manager object invokes an operation on the managed object using the CORBA object reference previously obtained. IDL stubs or the standard DII can be used when invoking operations on single managed objects. If IDL stubs are used, the CORBA manager object must first narrow the reference to a specific OMG IDL interface.

3.2.4 Invoking Operations with Scope and Filtering

Scoped operations can be invoked either through **OSIMgmt::ProxyAgent** or **OSIMgmt::ManagedObjects**. In the last case, the base managed object in the selection is the one being referred.

Figure 3-7 illustrates how CORBA manager objects invoke a scoped operation in a pure CORBA environment.

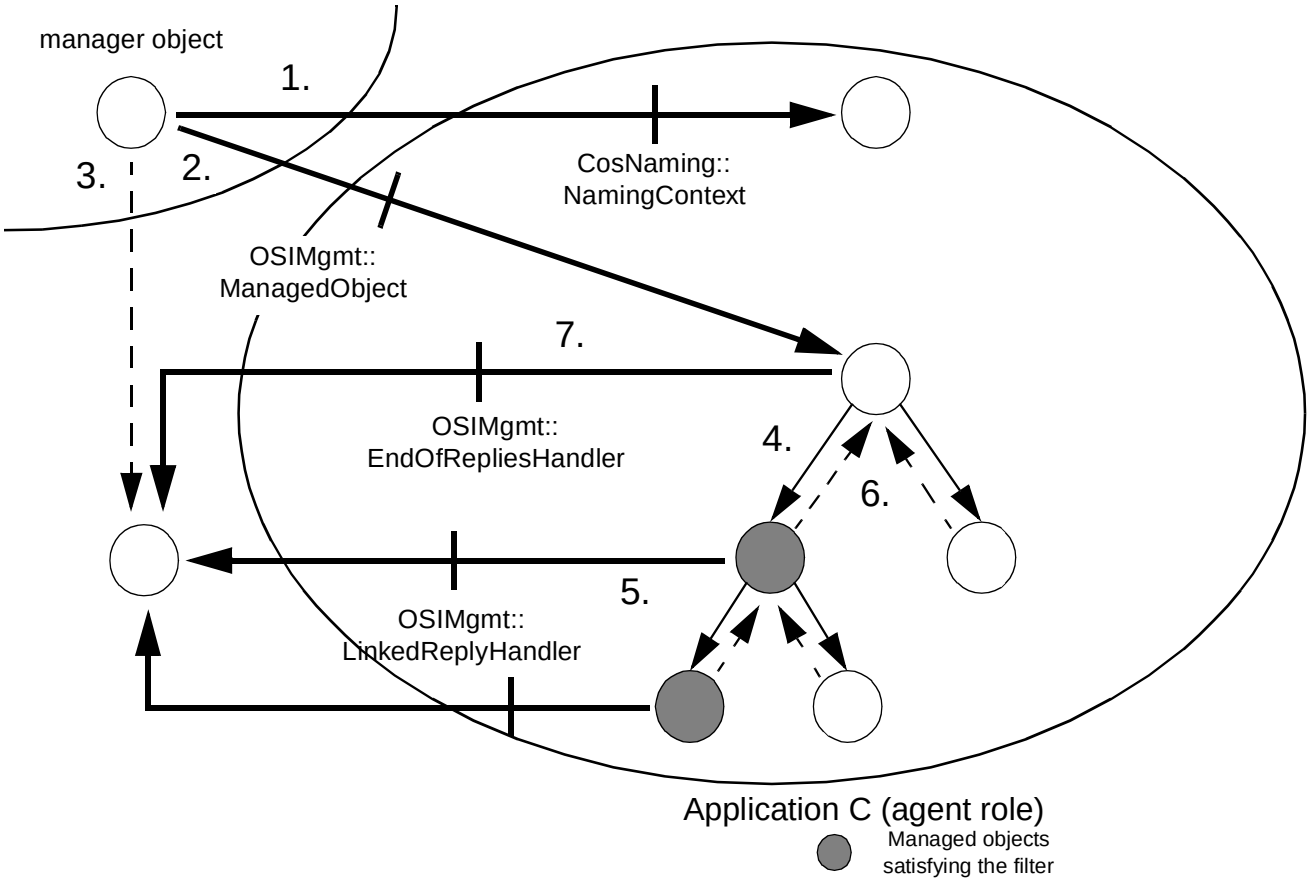

Figure 3-7 Invoking operation with scope and filtering

The following steps will be followed in case the scoped operations are invoked through operations exposed by the base managed object:

1. A CORBA manager object invokes the **resolve** operation exposed by the initial **CosNaming::NamingContext** object, passing the name of the managed object used as the base managed object in the scoped operation. As a result, a reference to the managed object is returned.

2. The CORBA manager object narrows the obtained CORBA object reference to a new object reference, bound to the **OSIMgmt::ManagedObject** interface, and invokes the appropriate scoped operation (**scoped_get**, **scoped_set**, **scoped_action**, or **scoped_delete**).

3. The CORBA manager objects select an object that exports the **OSIMgmt::MultipleRepliesHandler** interface (can create it). When invoking, the CORBA manager object passes a reference to this object (note that the CORBA manager object may be the one exporting the **OSIMgmt::MultipleRepliesHandler** interface).

4. The base managed object propagates the requests to its descendants.

5. Replies from each of the managed objects satisfying the filter, within the defined scope, are received by the **OSIMgmt::MultipleRepliesHandler** object. Information associated to each of the replies is passed when invoking the **send_reply** operation exposed by the **OSIMgmt::MultipleRepliesHandler** object.

6. The base managed object is notified when scoped descendants that pass the filter have sent their replies to the **OSIMgmt::MultipleRepliesHandler** object.

7. The base managed object invokes the **end_of_replies** operation exposed by the **OSIMgmt::MultipleRepliesHandler** object. If an error occurs during the whole process, an exception is generated, converted into a CORBA any value and passed to the **OSIMgmt::LinkedRepliesHandler** object by invoking either the **send_mo_error** or **send_subtree_error** operations.

Code for invoking scoped operations

The following example shows how the fragment of code used to find a Log object by name should look.

```
OSIMgmt::ManagedObject_ptr managed_object;
X711Inf::DistinguishedNameType object_name;
OSIMgmt::LName_ptr local_name;
OSIMgmt::MultipleRepliesHandler_ptr handler;
.................
local_name -> for_osi_form (object_name);
CORBA::Object_ptr obj = ctx -> resolve (local_name -> to_idl_form ());
managed_object = OSIMgmt::ManagedObject::_narrow (obj);

managed_object -> scoped_action (scope, filter, sync, access_control,
                                                    “reset”, arg, handler, handler);
```

3.2.5 Iterator Interfaces for Scoped Operations

In other OMG specifications, the iterator pattern is used heavily for operations that return an unbounded list of responses. This is how it could be done, with the specified interfaces, as follows. The scoped operations both in **ProxyAgent** and **ManagedObject** share the last parameters in the list as in:

```
cmis_get(..., in LinkedReplyHandler lrh, in EndOfRepliesHandler eorh)
```

The iterator interfaces for other services look like:

```
iter_get(..., in unsigned long how_many, out ReplyList rlist,
      out ReplyIterator riter)
```

This operation can be implemented by the following pseudocode, using our currently proposed interfaces.

Using ReplyIterator

```
iter_get(..., in unsigned long how_many,
      out ReplyList rlist,
      out ReplyIterator riter)
{
 BufferedRepliesHandler brh = create_brh(...)   // or other creation
                                                //    call
      cmis_get(..., brh, brh)
      brh -> get_n_replies(how_many, rlist)
      riter = brh  // widening does not require narrow()
}
```

Note that, although this code is simple, implementing this in either a pure CORBA Agent, pure CORBA Managed object or a gateway may impose unacceptable performance and scalability constraints in the implementation as potentially unbounded buffering must occur in a single point, and concentration of responses through a single CORBA object will also happen.

Other more flexible implementations are allowed by means of combining the simpler pieces together, as done above, therefore avoiding the scalability and performance problems (at least, reducing them).

Besides, the BRH objects can be implemented both as local objects (in the client address space) or as remote objects (accessed via CORBA invocations) either in the gateway/CORBA agent/CORBA managed object address space or in a separate CORBA service process.

3.2.6 Reception of Events at CORBA Managers

Different strategies to resolve how CORBA manager objects finally consume events (see Chapter 2 for details).

3.2.7 Forwarding Events from CORBA Managed Object Domains

Different strategies to resolve how CORBA managed objects finally report events can be implemented (see Chapter 2 for details). The only distinction is that the interface of the EventReporter object is well-known and correspond to the standard Event Forwarding Discriminator interface.

3.3 CORBA/CMIP Gateways

This section is provided as information only, and does not represent a normative part of this specification.

In this section, different gateway scenarios are described where the use of this specification will be clarified. This should be considered as a high level tutorial on some potential uses of the JIDM model for OSI management. Also, some potential implementation options are discussed.

3.3.1 Manager Side Gateways

3.3.1.1 Overview

CORBA/CMIP gateways must be used by any CORBA Manager Application needing to interoperate with managed object domains that are not directly accessible via CORBA but are accessible via CMIP.

A CORBA/CMIP gateway runs in one CORBA server. However, a CORBA/CMIP gateway can coexist with one or several JIDM gateways in the same CORBA server. Programs of the CORBA server have access to both; ORB services and services encapsulating access to management-specific protocols provided by JIDM gateways at the server.

Any CORBA/CMIP gateway has several CORBA objects associated with it:

- A **JIDM::ProxyAgentFinder** object for establishing connections to OSI managed object domains accessible via CMIP through the gateway.
- One or several **JIDM::EventPort** objects for receiving notification of events from members of OSI managed object domains accessible via CMIP through the gateway.

The **JIDM::ProxyAgentFinder** object is created during start-up of the CORBA server where the JIDM gateway is going to run. **JIDM::EventPort** objects at the gateway may be created during or after start-up of that server. This typically requires the existence of an **EventPortFactory** object at the gateway.

As previously explained, several JIDM gateways can exist in a CORBA manager and a **JIDM::ProxyAgentFinder** object is associated with each of them. All of them would be registered in a root **JIDM::ProxyAgentFinder** object at the CORBA manager. CORBA managers typically obtain a reference to this local root **JIDM::ProxyAgentFinder** object by using standard CORBA Initialization Services.

The **JIDM::ProxyAgentFinder** object is created during start-up of the CORBA server where the JIDM gateway is going to run. **JIDM::EventPort** objects at the gateway may be created during or after start-up of that server.

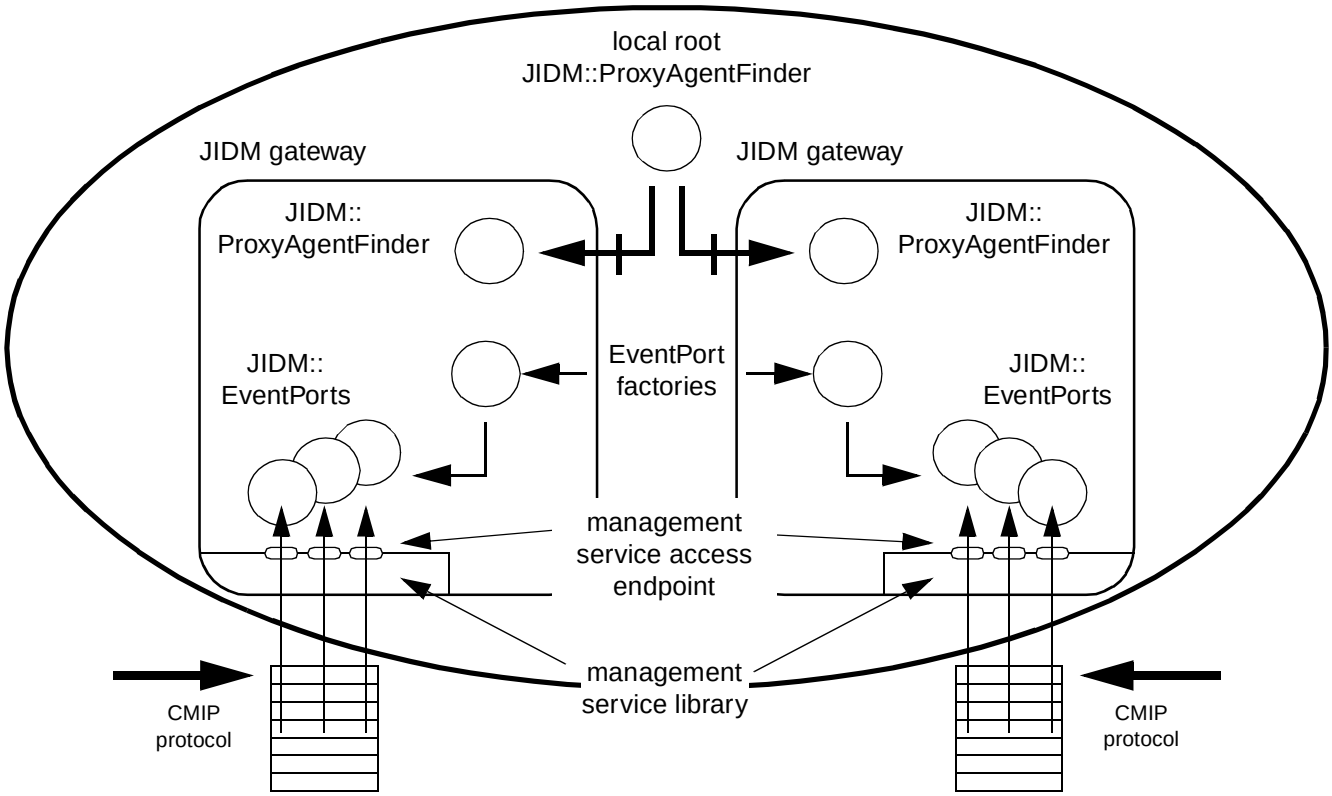

Figure 3-8 Structure of CORBA/CMIP gateways (manager side)

As a result of establishing a connection through a CORBA/CMIP gateway, an **OSIMgmt::ProxyAgent** object is created at the gateway. **OSIMgmt::ProxyAgent** objects created this way are responsible for:

- Creating a **CosLifeCycle::FactoryFinder** object that in turn enables creation of CORBA factories that handle creation of managed objects at the domain.
- Creating a **CosNaming::NamingContext** object that in turn enables creation of CORBA proxy managed objects for each member of the domain.
- Sending scoped operation requests.

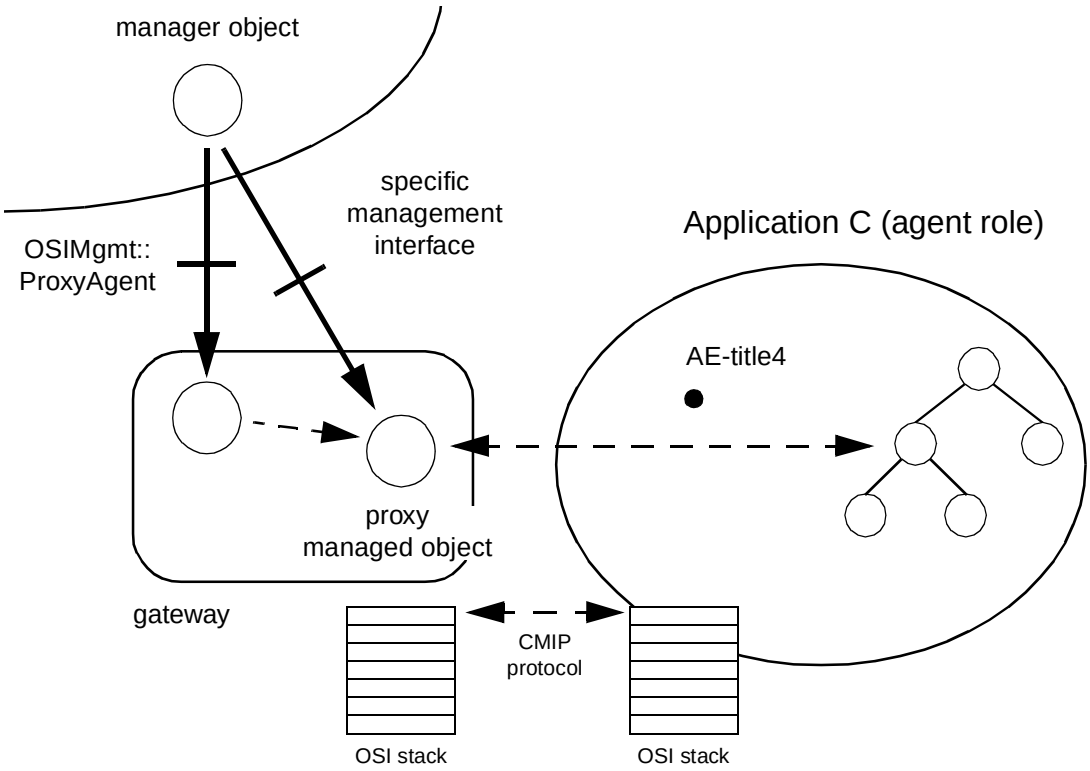

Figure 3-9 OSIMgmt::ProxyAgents in a gateway

3.3.1.2 Getting access to managed object domains

The following steps are used when a CORBA manager object tries to get access to a external managed object domain using a CORBA/CMIP gateway (see Figure 3-9):

1. The CORBA manager object invokes the **access_domain** operation exported by the **JIDM::ProxyAgentFinder** object located at the gateway. Information that unequivocally identifies the managed object domain to be accessed is passed in the invocation.

2. As a result of invoking the **access_domain** operation, a CORBA **OSIMgmt::ProxyAgent** object is created at the gateway. The new **OSIMgmt::ProxyAgent** object is bound to a CMIP communication endpoint (a CMIS access point). If a specific domain title was specified in the criteria passed as argument to the **access_domain** operation, then a connection is established with the managed object domain. In such a case, the **OSIMgmt::ProxyAgent** is responsible to manage resources associated with the connection.

3. A reference to the **OSIMgmt::ProxyAgent** object is returned to the CORBA manager object that requested access to the managed object domain being considered. This reference is returned as a reference to a **JIDM::ProxyAgent** object. To use specific operations in the **OSIMgmt::ProxyAgent** interface, manager objects must narrow the reference they receive.

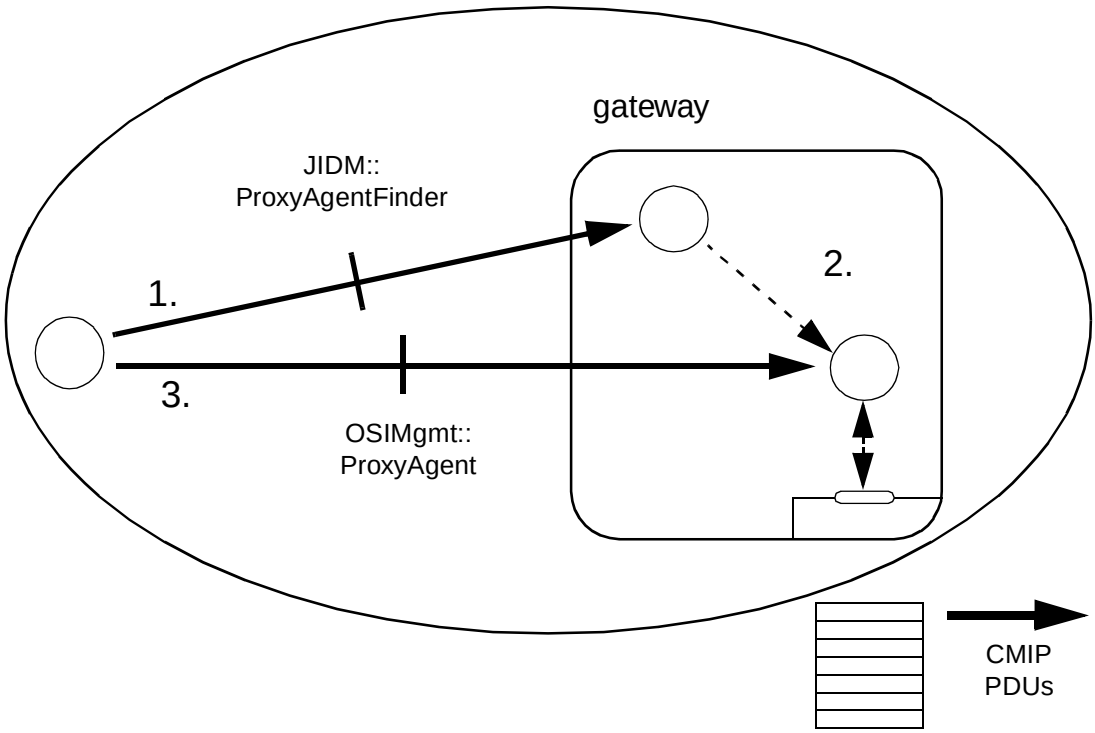

Figure 3-10 Finding references to OSIMgmt::ProxyAgents in a JIDM gateway

Each **OSIMgmt::ProxyAgent** object encapsulates access to a domain by establishing a session with that domain.

Different solutions can be implemented:

- **OSIMgmt::ProxyAgent** objects located in the same CORBA/CMIP gateway may share the same session, but each of them is associated with a different context.
- Each **OSIMgmt::ProxyAgent** object has a different session associated with it.

3.3.1.3 Creation of managed objects

Implementors of create operations exported by proxy managed object factories are responsible for constructing the appropriate CMIP m-create requests and for returning the appropriate results.

Different types of factories can be found according to criteria passed in the invocation of the **find_factories** operation exported by the initial **CosLifeCycle::FactoryFinder** visible through the CORBA/CMIP gateway:

- **OSIMgmt::ManagedObjectFactory**
- **CosLifeCycle::GenericFactory**

The following steps are followed when a CORBA manager creates a managed object at some domain that is accessible through a CORBA/CMIP gateway (see Figure 3-11 on page 3-62):

1. The CORBA manager invokes the **get_domain_factory_finder** operation exported by the **OSIMgmt::ProxyAgent** object.
2. The CORBA manager invokes the **find_factories** operation exported by the returned **CosLifeCycle::FactoryFinder** object, passing a valid key value.
3. The **CosLifeCycle::FactoryFinder** object finds references for appropriate managed object factories at the JIDM gateway. If there is no managed object factory matching the key, the **CosLifeCycle::FactoryFinder** object creates one. References to managed object factories are returned to the CORBA manager.
4. The CORBA manager object invokes an operation on the managed object factory using the CORBA object reference it obtained. Typically, the CORBA manager object narrows this object reference to a specific managed object factory interface supported by the factory (the **CosLifeCycle::GenericFactory** or the **OSIMgmt::ManagedObjectFactory** interface, for example).
5. The CORBA request is received by the CORBA/CMIP gateway and is translated into an appropriate m-create request PDU. This m-create request PDU is sent through the association handled by the **OSIMgmt::ProxyAgent**.
6. When the response to the m-create request PDU is received, the invoked operation returns with the appropriate result values.
7. If the create operation must return an object reference then a CORBA proxy managed object is also created at the gateway.

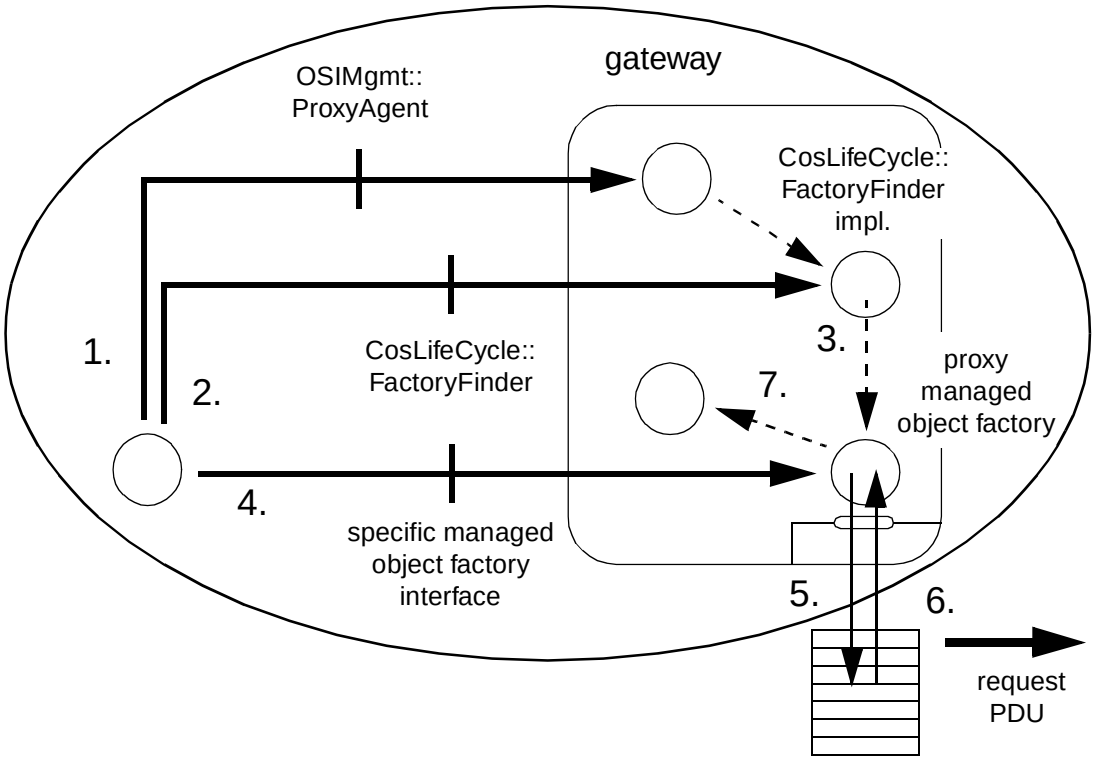

Figure 3-11 Creating managed objects through a CORBA/CMIP gateway

3.3.1.4 Invocation of operations on single managed objects

The following steps are followed when a CORBA manager invokes an operation on a managed object that is accessible through a CORBA/CMIP gateway (see Figure 3-12 on page 3-63):

1. The CORBA manager invokes the **get_domain_naming_context** operation exported by the **OSIMgmt::ProxyAgent** object.

2. A CORBA manager object invokes the **resolve** operation exported by the returned **CosNaming::NamingContext** object, passing the name of the managed object upon which it wants to operate.

3. The **CosNaming::NamingContext** object finds a reference to the CORBA object acting as the proxy of the managed object and returns it to the CORBA manager that requested it. The CORBA proxy managed object resides in the JIDM gateway. The **CosNaming::NamingContext** object is responsible for creating the CORBA proxy managed object if it didn't exist at the gateway, the first time an existing managed object is accessed.

4. The CORBA manager object invokes an operation on the managed object using the CORBA object reference to the corresponding proxy. IDL stubs or the standard DII can be used to perform this action. Whenever IDL stubs are used, the CORBA manager must narrow the reference, obtained from the **CosNaming::NamingContext**, to a specific OMG IDL interface.

5. The CORBA request is received by the CORBA/CMIP gateway and translated into an appropriate management request PDU. This request PDU is sent through the association handled by the **OSIMgmt::ProxyAgent**.

6. When the response to the request PDU is received, the invoked operation returns with the appropriate result values.

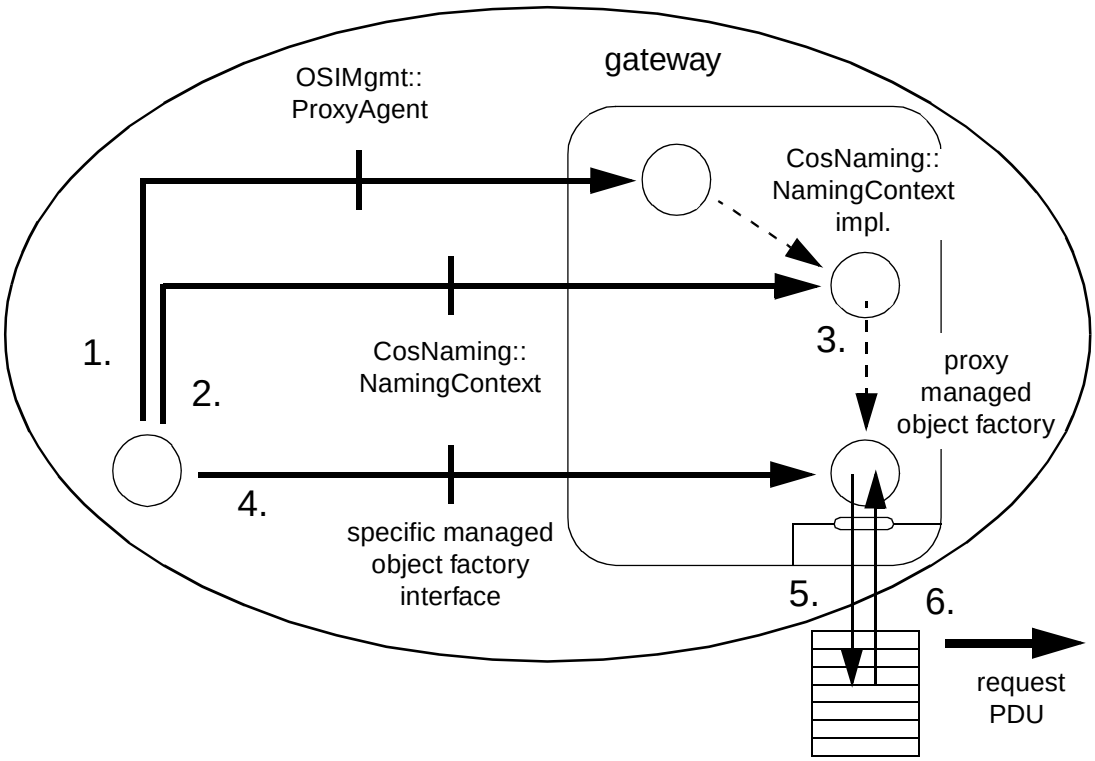

Figure 3-12 Invoking operations on a managed object through a CORBA/CMIP gateway

3.3.1.5 Invoking operations with scope and filtering

The following steps are followed when a CORBA manager invokes an operation on a managed object that is accessible through a CORBA/CMIP gateway (see Figure 3-13 on page 3-65):

1. A CORBA manager object invokes the **resolve** operation exported by the initial **CosNaming::NamingContext** object located at the managed object domain, passing the name of the managed object used as base of the scoped operation.

2. The **CosNaming::NamingContext** object finds a reference to the CORBA object acting as the proxy of the managed object, in the CORBA/CMIP gateway, and returns it to the CORBA manager object requesting it.

3. The CORBA manager object narrows the obtained CORBA object reference to a new object reference, bound to the **OSIMgmt::ManagedObject** interface, and invokes the appropriate scoped operation (**scoped_get**, **scoped_set**, **scoped_action**, or **scoped_delete**).

4. In the invocation, the CORBA manager object passes a reference to an **OSIMgmt::MultipleRepliesHandler** object (this may be a reference itself if the manager exports this interface).

5. The CORBA request is received by the CORBA/CMIP gateway and is translated into an appropriate CMIP request PDU. This CMIP request PDU is sent through the CMIP communication endpoint associated with the **OSIMgmt::ProxyAgent** through which the reference to proxy managed object was obtained. If a nil object reference was passed as the **OSIMgmt::EndOfRepliesHandler**, the CMIP request PDU is sent unconfirmed.

6. Replies from each of the managed objects satisfying the filter, within the defined scope, are received by the CORBA/CMIP gateway.

7. Information associated with each of the replies is passed by invoking the **send_reply** operation exported by the **OSIMgmt::MultipleRepliesHandler** object. Once all replies have been received, the CORBA/CMIP gateway invokes the **end_of_replies** operation exported by the **OSIMgmt::MultipleRepliesHandler** object.

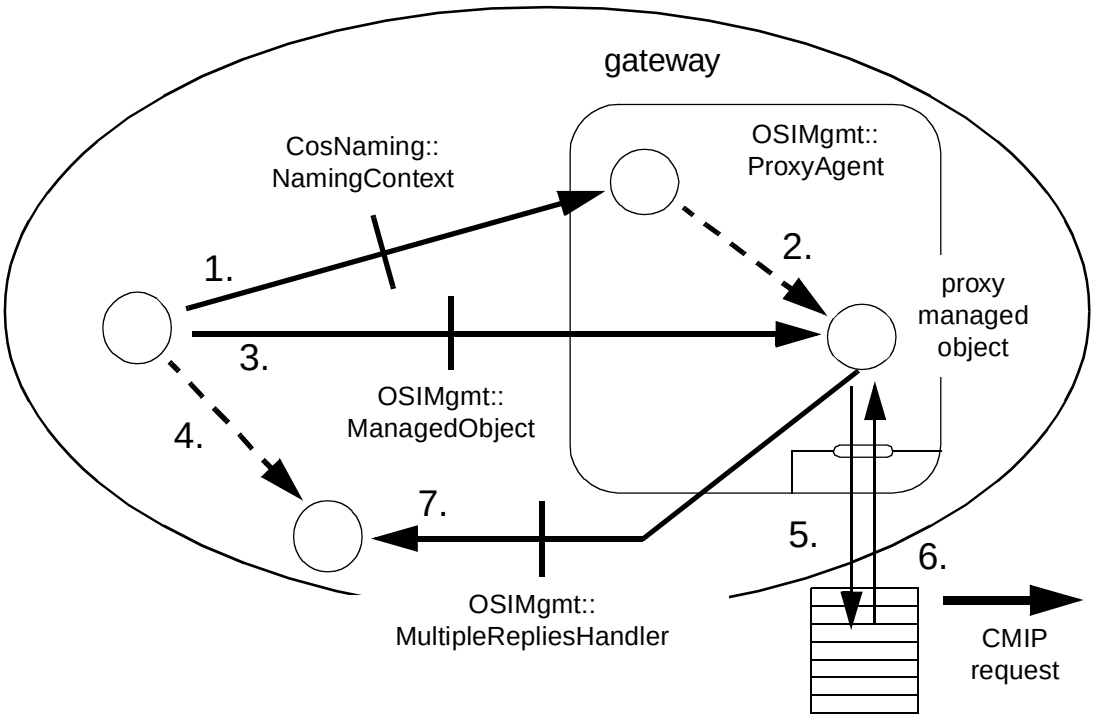

Figure 3-13 Invoking operation with scope and filtering

The gateway is much simpler to program. Actually, it neither needs to maintain a local copy of the naming tree nor to test which managed objects satisfy the filtering. It only needs to:

- create an object in the CORBA/CMIP gateway that exports the **LinkedReplyHandler** and the **EndOfRepliesHandler** interfaces.
- invoke the scoped operation exported by the base managed object through the **OSIMgmt::ManagedObject** interface.

Different strategies can be implemented to resolve how each object is going to forward operations to its descendants and detect that there aren't pending replies to its descendants, but they are transparent to the gateway.

CORBA manager objects can also invoke operations that are directly exported by an **OSIMgmt::ProxyAgent** and which basically correspond to an encapsulation of CMIS primitives. Note that references to CORBA proxy managed objects are not necessary in that case (the name of the interface and the managed object are passed as arguments). This may be a way to solve scalability problems.

3.3.1.6 *Event reception*

Events originated at managed object domains are always received through **JIDM::EventPort** objects at CORBA Managers. A mechanism is implemented at any CORBA/CMIP gateway that allows event data received at a management connection endpoint to be forwarded to the appropriate **JIDM::EventPort** object.

As already mentioned in Section 2.2.4, “Reception of Events at CORBA Managers,” on page 2-25, different strategies to resolve how CORBA manager objects finally consume events can be implemented. For example, CORBA manager objects can register themselves directly to **OSIMgmt::EventPorts** or via some additional event channel.

The following steps are followed when a CORBA manager receives an event through a **JIDM::EventPort** at a CORBA/CMIP gateway:

1. During the start up phase of the CORBA Manager Application, one or more application objects register themselves either as **CosEventComm::PushConsumers** or **CosEventComm::PullConsumers** in each of the existing **OSIMgmt::EventPorts**.

2. An m-event-report indication PDU containing notification of an event from a managed object is received by the CORBA/CMIP gateway through some association. This association is bound to a specific title and has a **JIDM::EventPort** object associated with it, which finally receives the event data carried in the PDU.

3. The appropriate response is sent by the CORBA/CMIP gateway back to the application that reported the event, confirming that the event was received at the Manager Application.

4. The **JIDM::EventPort** invokes the push operation exported by all **CosEventComm::PushConsumers** objects connected to it. Data of the event is passed in the invocation as an any.

5. The **JIDM::EventPort** maintains the event until all **CosEventComm::PullConsumers** objects connected to the port pull the event. Data of the event is obtained by consumers as an any.

6. **CosEventChannelAdmin::EventChannel** objects can be connected as consumers to the event port. In such a case, manager objects performing management functions can be connected to the channel instead of directly to the event ports.

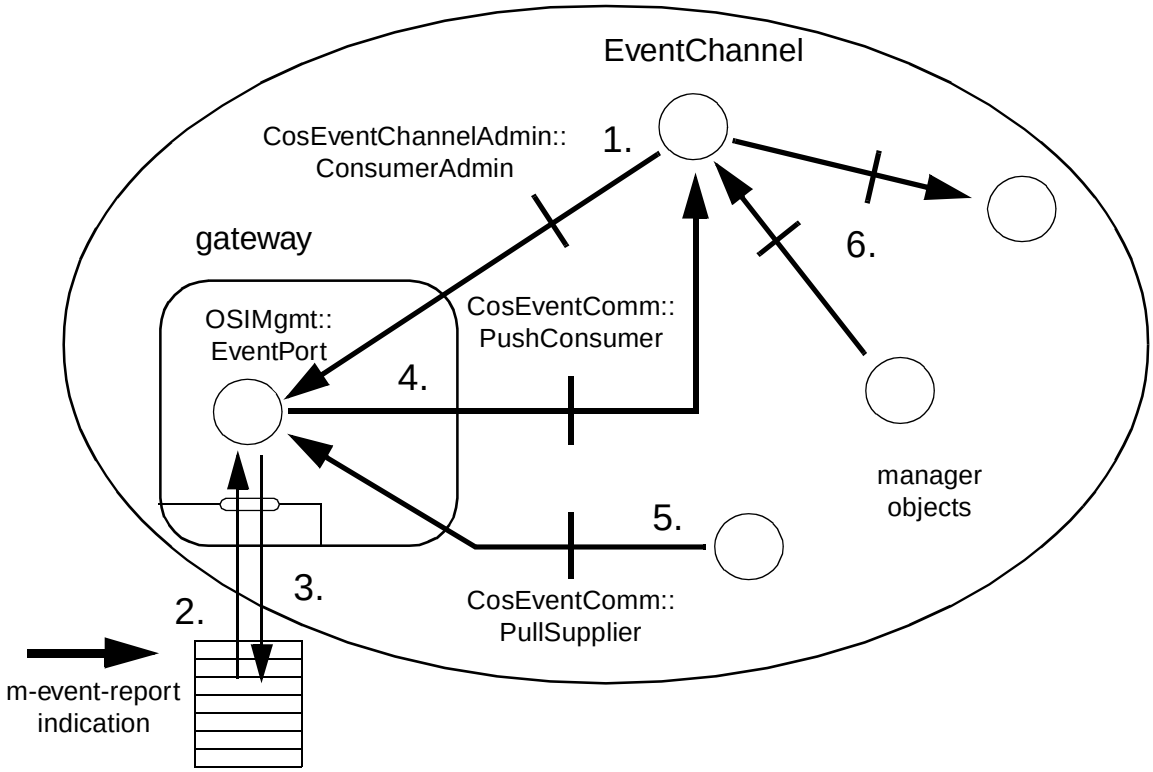

Figure 3-14 Event reporting at CORBA/CMIP gateways (manager side)

3.3.1.7 CMISE service level scenarios

The **OSIMgmt::ProxyAgent** objects provide CMIS service level IDL methods, that would allow any CORBA manager application to perform all CMIS operations without the need to have any further CORBA object references to the corresponding CORBA objects. This type of interaction is most useful in gateway situations, although it is applicable to pure CORBA environments as well.

This section presents some scenarios as they would apply to CORBA manager to OSI agent gateway environments. These scenarios also assume that the **ProxyAgent** object has been previously located or created by the manager object by invoking the **access_domain** operation on the **JIDM::ProxyAgentFinder** object.

Scenario 1 - Creating managed objects

Asynchronous create operation

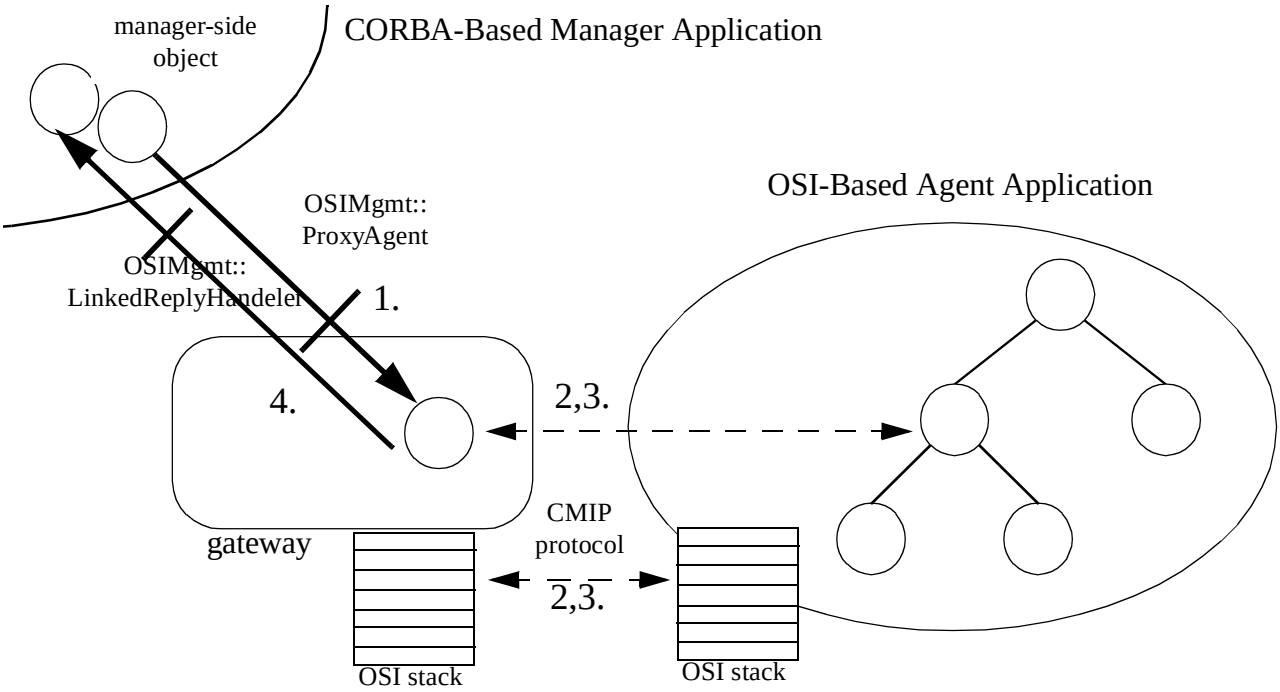

Figure 3-15 Asynchronous creation of a managed object through a CORBA/CMIP gateway

The following steps are taken each time an asynchronous M-CREATE request is sent through the CORBA/CMIP Gateway.

1. The CORBA manager object invokes the **cmis_create** operation against the **OSIMgmt::ProxyAgent** object. The asynchronous **cmis_create** method takes the additional **OSIMgmt::LinkedReplyHandler** object reference against which the response method will be invoked. The other input arguments are the X711CMI CMIS data arguments, the creation method (**creation_kind**), the **object_name** (distinguished name), and the **interface_name** of the object being created.

2. The **OSIMgmt::ProxyAgent** object implementation transforms the IDL data into a CMIP PDU and sends it over the OSI stack to the OSI agent. Once the PDU has been sent, the asynchronous **cmis_create** call returns.

3. At some later point, the response comes back to the gateway from the OSI agent over the OSI stack. The gateway implementation internally converts the response to its IDL equivalent.

4. The gateway invokes the **send_reply** (or **send_mo_error**, **send_subtree_error** if an error occurred) method against the **OSIMgmt::LinkedReplyHandler** object that was passed to the original **cmis_create** call, passing the response data as an argument.

Synchronous create operation

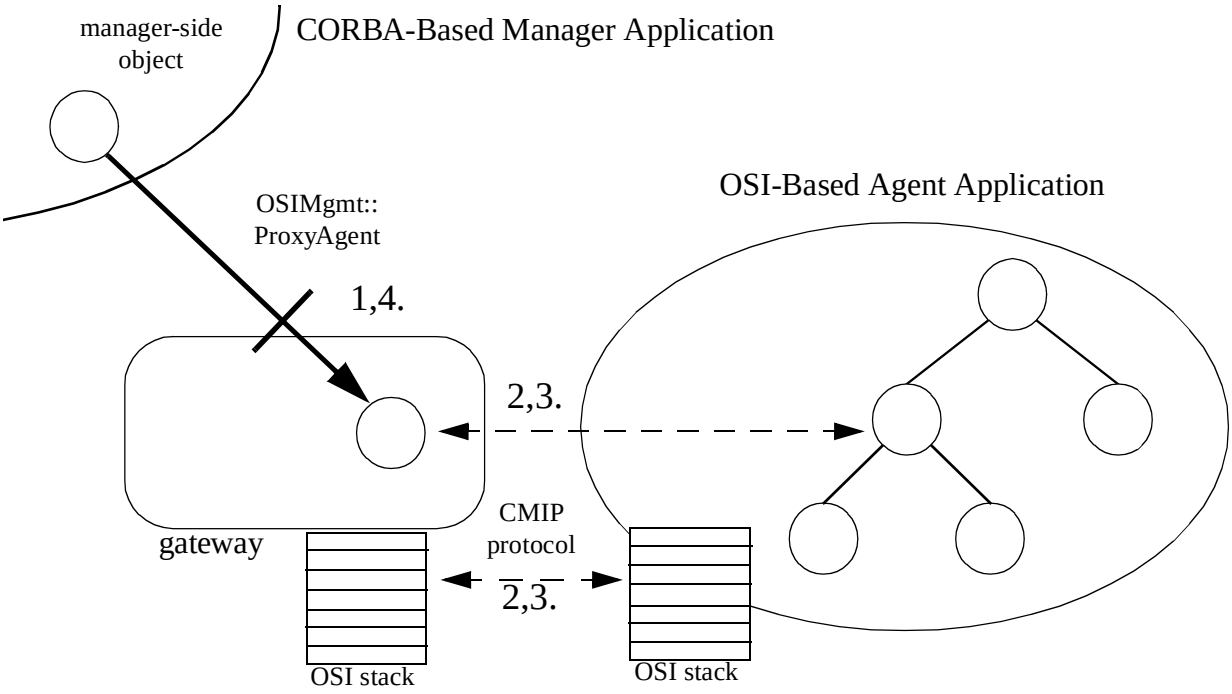

Figure 3-16 Synchronous creation of a managed object through a CORBA/CMIP gateway

The following steps are taken each time a synchronous M-CREATE request is sent through the CORBA/CMIP Gateway.

1. The CORBA manager object invokes the **cmis_create_sync** operation against the **OSIMgmt::ProxyAgent** object. The input arguments are the X711CMI CMIS data arguments, the creation method (**creation_kind**), the **object_name** (distinguished name), and the **interface_name** of the object being created.

2. The **OSIMgmt::ProxyAgent** object implementation transforms the IDL data into a CMIP PDU and sends it over the OSI stack interface to the OSI agent. Once the PDU has been sent, the **cmis_create_sync** call blocks, waiting for the response.

3. At some later point, the response comes back to the gateway from the OSI agent over the OSI stack. The gateway implementation internally converts the response to its IDL equivalent.

4. The **cmis_create_sync** method returns the results of the create operation through the out-arguments of the **cmis_create_sync** call. These include the **created_interface_name**, the **created_object_name** (distinguished name of the object actually created), the **creation_time** (time when the managed object was created), and the **created_attribute_values** (values of attributes of the new object).

Scenario 2 - Generic operations on managed objects

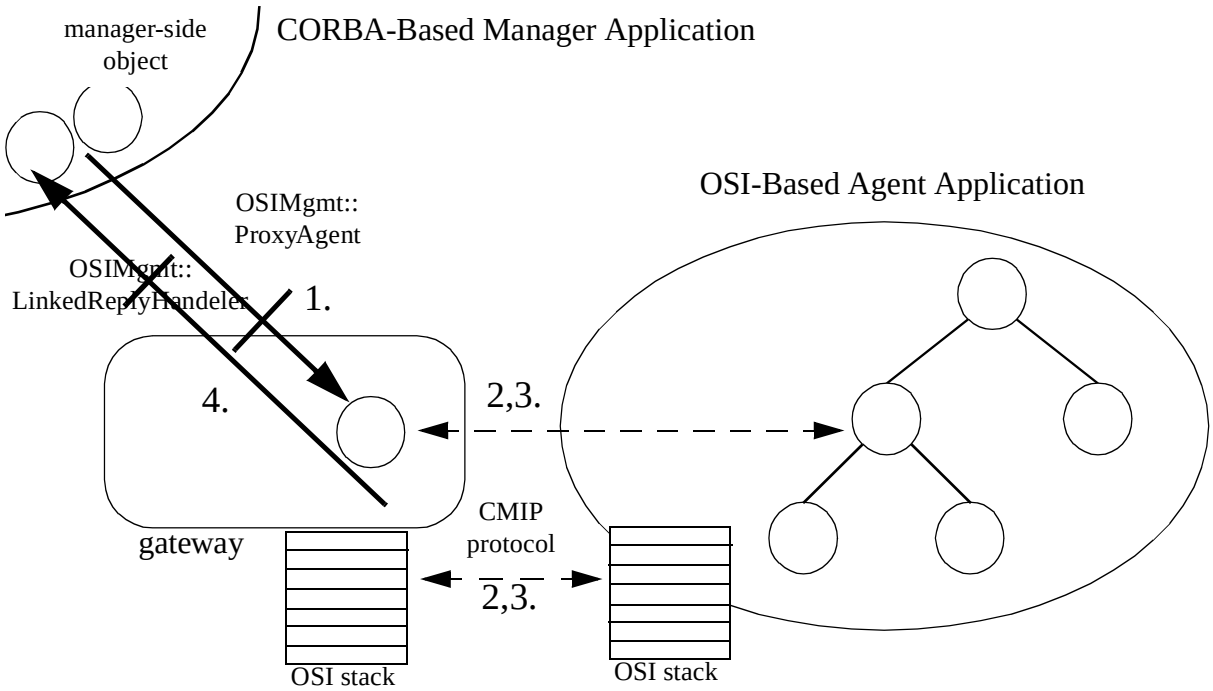

Figure 3-17 Operations against a managed object through a CORBA/CMIP gateway

The following steps are taken each time an M-GET, M-SET, M-ACTION or M-DELETE request is sent through the CORBA/CMIP Gateway.

1. The CORBA manager object invokes the desired operation against the **OSIMgmt::ProxyAgent** object. All method invocations that will result in one or more responses take the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** object references - against which the separate response methods will be later invoked. For unconfirmed operations, both object references should be nil. The other input arguments are the X711CMI CMIS scope, filter, synchronization and access control arguments, the **object_name** (distinguished name), and the **interface_name**.

2. The **OSIMgmt::ProxyAgent** object implementation transforms the IDL data into a CMIP PDU and sends it over the OSI stack to the OSI agent. Once the PDU has been sent, operation invocation returns.

3. At some later point, a response comes back to the gateway from the OSI agent over the OSI stack. The gateway implementation internally converts the response to its IDL equivalent.

4. The gateway invokes the **send_reply** (or **send_mo_error**, **send_subtree_error** if an error occurred) method against the **OSIMgmt::LinkedReplyHandler** object which was passed to the original operation invocation, passing the response data as an argument. If the current reply is the last one, this will be immediately followed by an invocation of the **end_of_replies** method against the **OSIMgmt::EndOfRepliesHandler** object reference which was passed in to the initial request operation.

Scenario 3 - Cancelling a get operation

This scenario describes how the M-CANCEL-GET operation may be sent from a CORBA manager to an OSI agent, assuming that an M-GET request is outstanding.

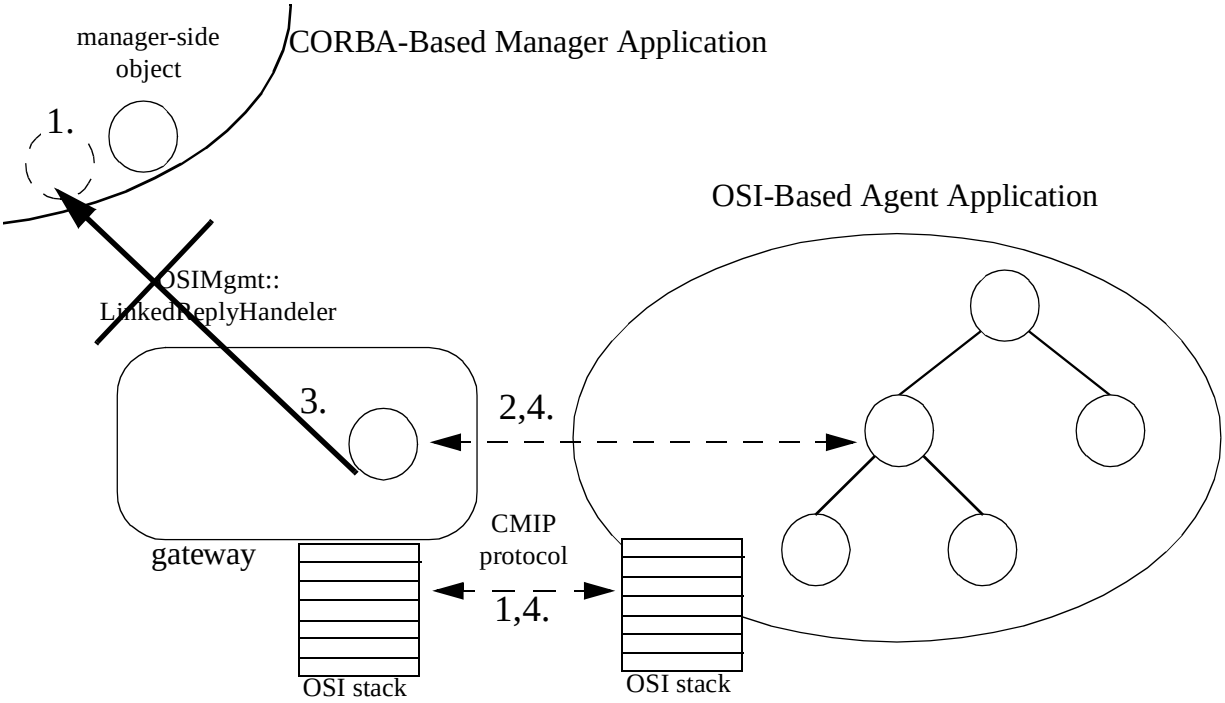

Figure 3-18 Cancelling an outstanding M-GET operation

The following steps must be taken in order to get the CORBA/CMIP gateway to send an M-CANCEL-GET request to a pending M-GET.

1. At some point after the initial M-GET request is issued, but before all responses have been received, the CORBA manager deletes the **OSIMgmt::LinkedReplyHandler** object, which was associated with the original request.

2. A response comes back to the gateway from the OSI agent over the OSI stack. The gateway implementation internally converts the response to its IDL equivalent.

3. The gateway attempts to invoke the **send_reply** (or **send_mo_error**, **send_subtree_error** if an error occurred) method against the **OSIMgmt::LinkedReplyHandler** object that was passed to the original operation invocation, passing the response data as an argument. Since this object no longer exists, a standard CORBA exception OBJECT_NOT_EXIST will be thrown.

4. The gateway catches this exception, which tells it to synthesize an M-CANCEL-GET PDU and send it down the OSI stack to the OSI agent.

3.3.2 Agent Side Gateways

3.3.2.1 Overview

CORBA/CMIP gateways must be used by any CORBA Agent Application needing to offer a management interface based on CMIP. A CORBA/CMIP gateway runs in one CORBA server. However, one or several JIDM gateways can coexist in the same CORBA server. Programs in this server have access to both ORB services and services encapsulating access to management-specific protocols provided by JIDM gateways at the server. Besides, there can be several CORBA servers containing JIDM gateways in the same CORBA Agent Application.

Any CORBA/CMIP gateway at a CORBA Agent Application has several objects associated with it (see Figure 3-19 on page 3-73):

- A **JIDM::EventPortFinder** CORBA object that enables CORBA managed objects at the agent application to establish connections to **JIDM::EventPort** objects at remote Manager Applications that are accessible through the gateway.
- A **JIDM::DomainPort** object that serves requests issued from remote Manager Applications that want to get access to managed objects at the local managed object domain.

These objects are created during start-up of the CORBA server where the CORBA/CMIP gateway is going to run.

Several JIDM gateways can exist in a CORBA Agent and a **JIDM::EventPortFinder** object is associated with each of them. All of them would be registered in a root **JIDM::EventPortFinder** object at the CORBA Agent (see Figure 3-19).

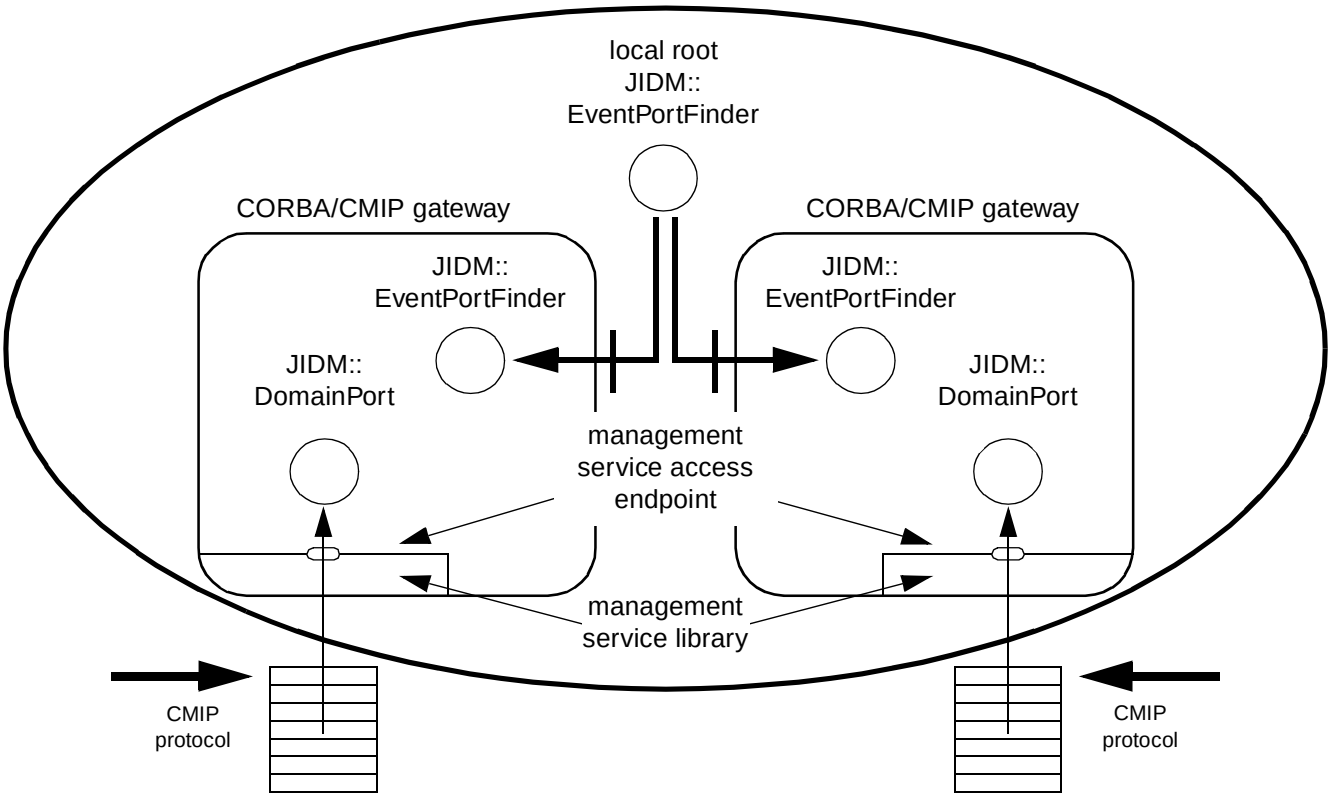

Figure 3-19 Structure of CORBA/CMIP gateways (agent side)

A root **CosNaming::NamingContext** object and a root **CosLifeCycle::FactoryFinder** object exist at any CORBA managed object domain. Whether these two interfaces are exported by the same CORBA object or different CORBA objects is an implementation issue. In addition, an **OSIMgmt::LocalRoot** object exists in order to deal with local orphan managed objects. References to these CORBA objects can be obtained from a CORBA/CMIP gateway by using the standard Initialization Services and are passed to the **JIDM::DomainPort** object at creation time.

3.3.2.2 Handling access to managed objects

A **JIDM::DomainPort** object resides in the CORBA/CMIP gateway in order to handle access to the managed object domain and serve association requests issued from remote Manager Applications.

Every **JIDM::DomainPort** object has an AE-title associated with it. This AE-title is used by remote Manager Applications to identify the OSI managed object domain accessible through the **JIDM::DomainPort** object.

When a new association request is received by the **JIDM::DomainPort** object that is in a gateway, the **JIDM::DomainPort** object creates a new **OSIMgmt::ProxyAgent** object. This object handles CMIS requests received through the newly established association.

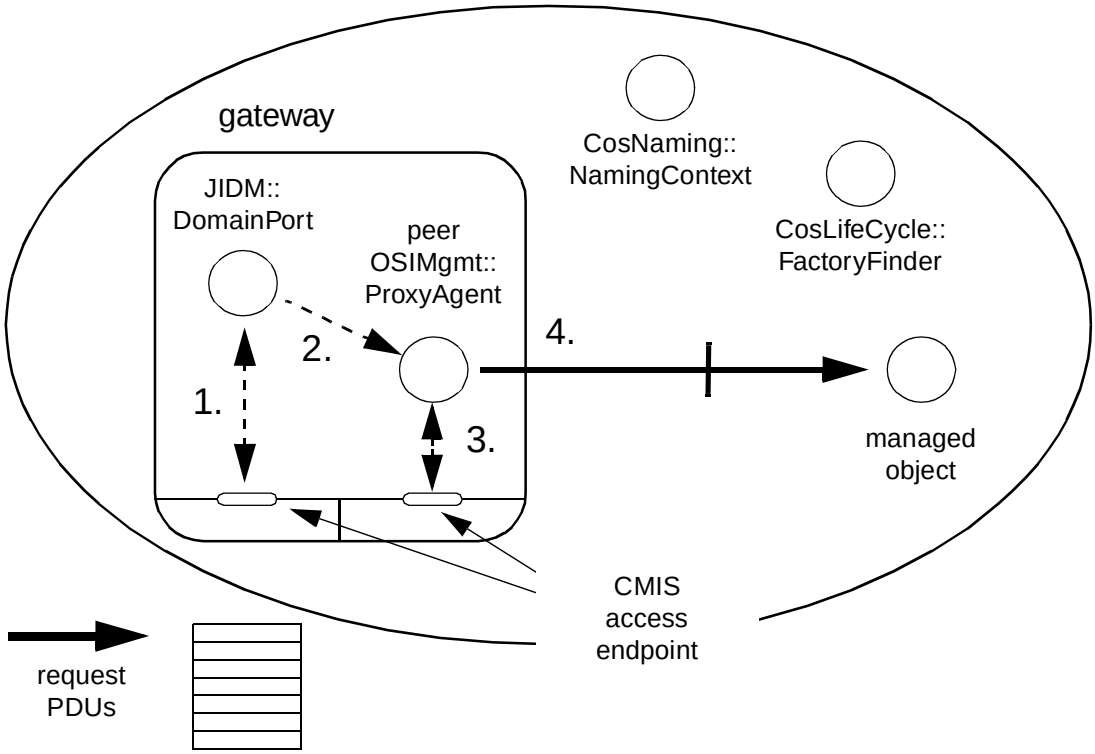

Figure 3-20 Handling access to local managed objects from a JIDM gateway

A **JIDM::DomainPort** object in a JIDM gateway holds references to the initial **CosNaming::NamingContext** and **CosLifeCycle::FactoryFinder** and **OSIMgmt::LocalRoot** objects in the managed object domain where the JIDM gateway is located. The **JIDM::DomainPort** object passes copies of these references to each **OSIMgmt::ProxyAgent** object it creates.

3.3.2.3 Creation of managed objects

In CORBA managed object domains, **OSIMgmt::ProxyAgent** objects receive PDU indications, perform the appropriate operations, and return the appropriate PDU responses.

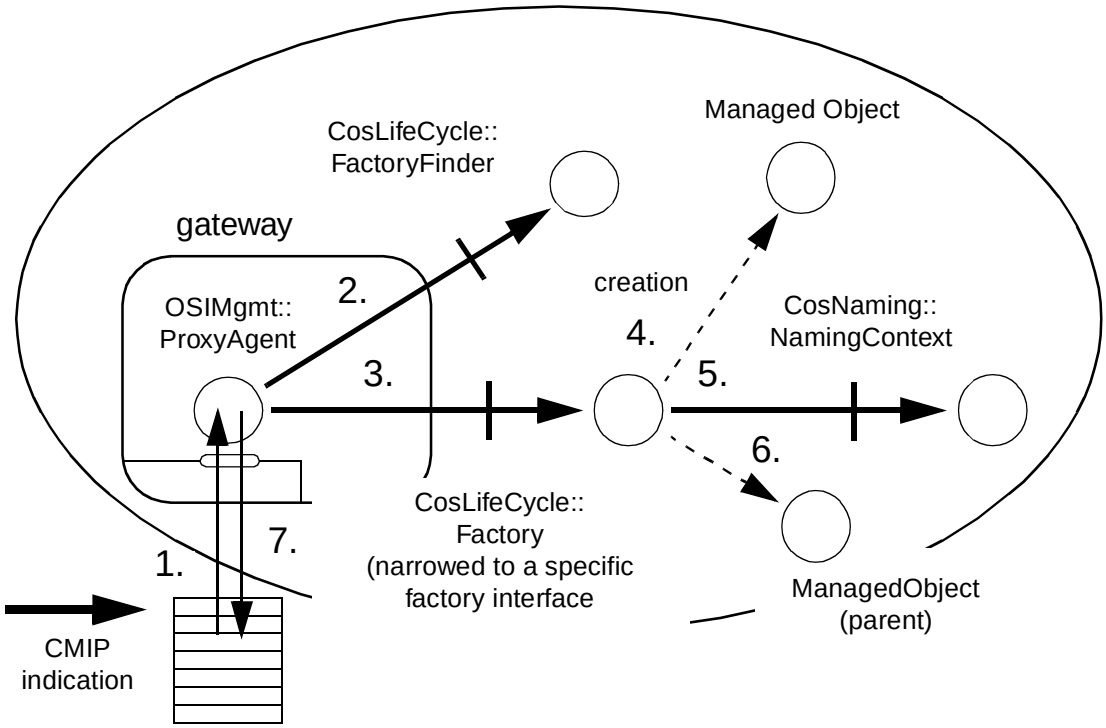

Figure 3-21 Handling management create PDU indications

The following steps are followed each time a create PDU indication is received by a CORBA/CMIP gateway:

1. An **OSIMgmt::ProxyAgent** object receives an m-create indication through the CMIS connection endpoint it holds.
2. The **OSIMgmt::ProxyAgent** object finds an appropriate factory by invoking the **find_factories** operation provided by the initial **CosLifeCycle::FactoryFinder** object at the managed object domain.
3. The **OSIMgmt::ProxyAgent** object narrows the obtained Factory object reference to a new object reference associated with a specific factory interface. Next, it invokes the operation for creating managed objects exported by the factory being referenced.
4. The Factory object creates a new CORBA managed object, an instance of the managed object type specified in the CMIP create indication (using the value of the Managed object class field in the CMIP PDU).
5. The Factory object binds an OSI name (the one passed as of the Managed object instance field in the CMIP PDU, but in IDL form) to the new CORBA managed object.
6. The Factory object informs the container (the naming context) of the CORBA managed object that a new subordinate object has been created.

7. When the operation invoked by the **OSIMgmt::ProxyAgent** object returns (or when an exception is raised), the **OSIMgmt::ProxyAgent** object constructs and sends an appropriate CMIP response to the remote OSI Manager Application.

3.3.2.4 Invocation of operations on single managed objects

OSIMgmt::ProxyAgent objects receive CMIP m-set/m-get/m-action indications on single objects, perform the appropriate operations and return the appropriate CMIP responses.

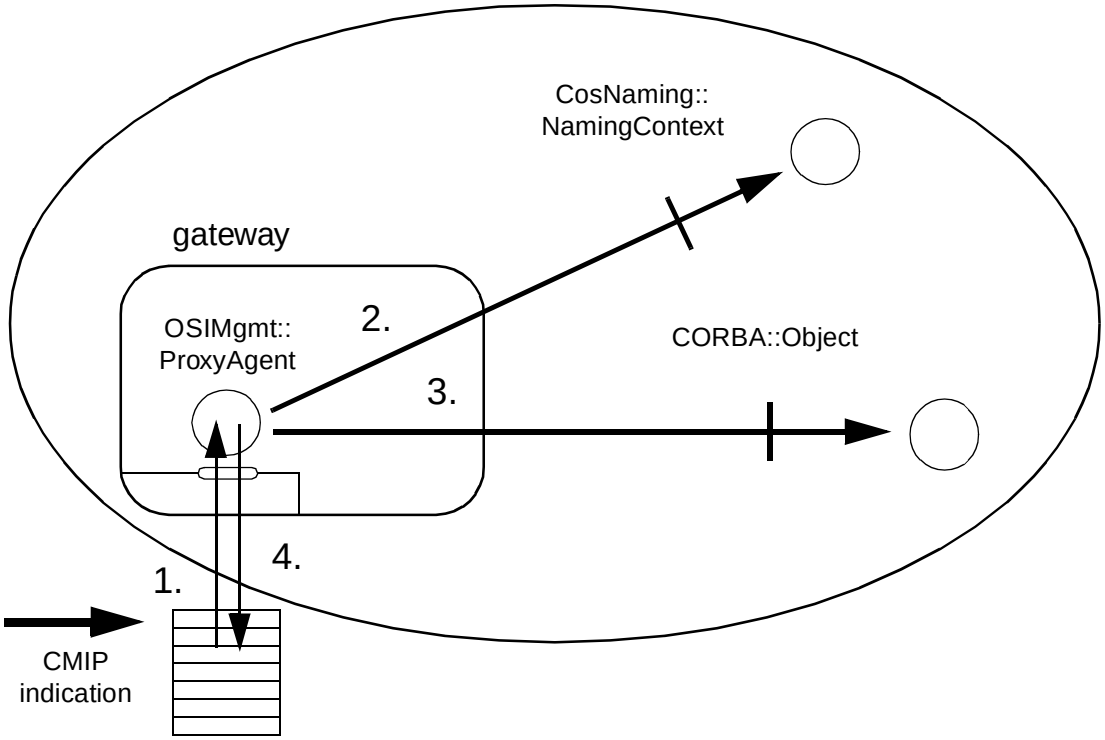

Figure 3-22 Invocation of operations on single managed objects

The following steps are used each time an m-set/m-get/m-action PDU indication on a single object is received by a CORBA/CMIP gateway:

1. An **OSIMgmt::ProxyAgent** object receives an m-set, m-get, or m-action indication, referred to a single object, through the CMIS connection endpoint it holds.

2. The **OSIMgmt::ProxyAgent** object finds a reference to the target managed object by invoking the **resolve** operation exported by the initial **NamingContext** object. The name of the target managed object (base object instance field in the CMIP indication) is passed in the invocation, once it is translated to IDL form (see OSI naming facilities).

3. The **OSIMgmt::ProxyAgent** object invokes the appropriate operation on the managed object. In a generic CORBA/CMIP gateway, this may be accomplished by using the Dynamic Invocation API provided by the local ORB.

4. When the management operation invoked by the **OSIMgmt::ProxyAgent** object returns (or when an exception is raised), the **OSIMgmt::ProxyAgent** object constructs and sends an appropriate CMIP response to the remote OSI Manager Application.

A reference to the **CosNaming::NamingContext** acting as the local naming root in the OSI Agent Application is passed to the **OSIMgmt::ProxyAgent** object at creation time. If the target managed object is going to send multiple replies as a result of invoking an action, an exception is triggered (see Specification Translation).

3.3.2.5 Handling CMIP indications with scope and filtering

In CORBA-based OSI Agent Applications, **OSIMgmt::ProxyAgent** objects receive CMIP m-set/m-get/m-action indications with scope and filtering, perform the appropriate operations and return all the appropriate CMIP responses associated with the generated replies.

The following steps are used each time a scoped m-set/m-get/m-action PDU indication is received by a CORBA/CMIP gateway:

1. An **OSIMgmt::ProxyAgent** object receives a CMIP m-set, m-get or m-action indication with scope and filtering, through the CMIS connection endpoint it holds.

2. The **OSIMgmt::ProxyAgent** object finds a reference to the base managed object by means of invoking the resolve operation exported by a **NamingContext** object. The name of the base managed object (base object instance field in the CMIP indication) is passed in the invocation, once it is translated to idl form (see OSI naming facilities).

3. The **OSIMgmt::ProxyAgent** object locally creates a CORBA object which is responsible for handling the different replies and which holds the same CMIS connection endpoint. This object exports two interfaces:
   - **OSIMgmt::LinkedReplyHandler**, for receiving individual linked replies.
   - **OSIMgmt::EndOfRepliesHandler**, for detecting the end of replies.

4. The **OSIMgmt::ProxyAgent** object narrows the CORBA object reference pointing to the base managed object instance to a new CORBA object reference associated with the **OSIMgmt::ManagedObject** interface. Next, it invokes the corresponding scoped operation, now visible through that narrowed reference.

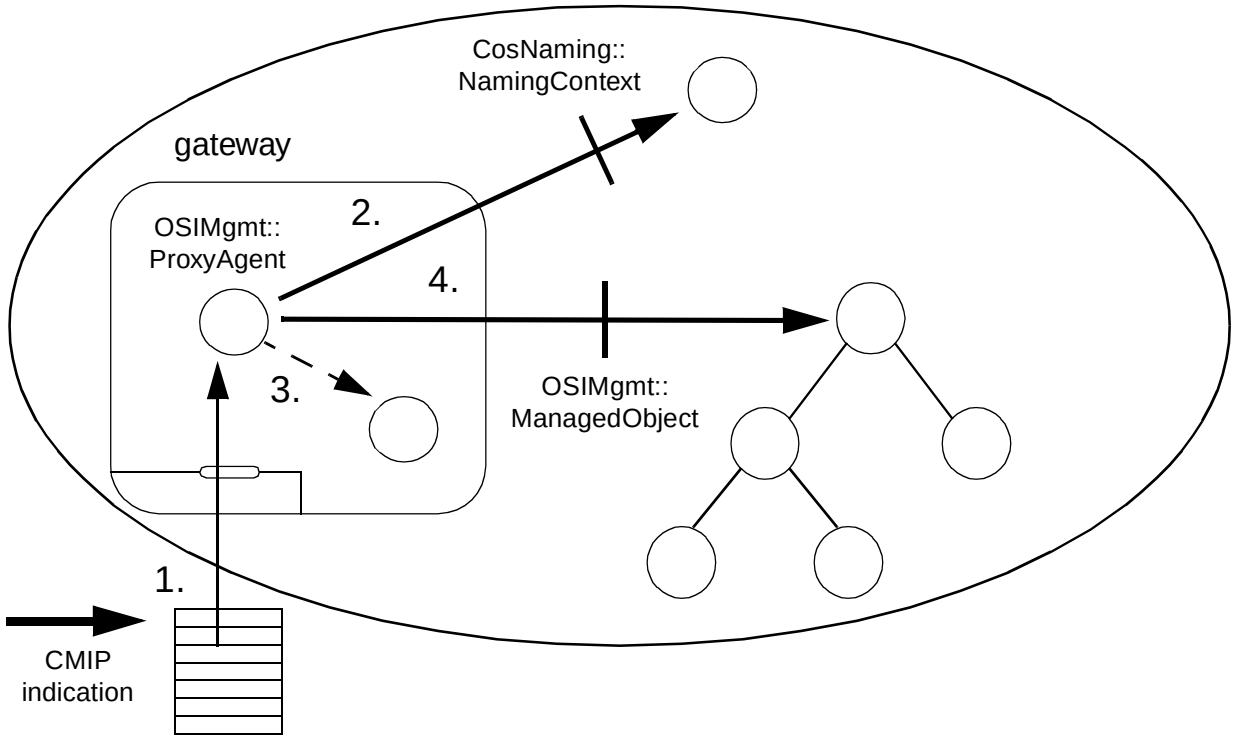

Figure 3-23 Handling CMIP indications with scope and filtering

Next, the following steps take place:

1. The base managed object (root of the subtree to which scope and filtering is going to be applied) propagates the operation to all descendants within the scope of the operation and waits until all descendants satisfying the filter have replied.
2. Every object within the scope that satisfies the filter invokes the **send_reply** operation exported by the **OSIMgmt::LinkedReplyHandler** object, in the gateway, is invoked.
3. The **OSIMgmt::LinkedReplyHandler** object constructs an appropriate CMIP response and sends it back through the CMIS endpoint connection it holds.
4. The base managed object is informed by its subordinates that there are no pending replies.
5. The base managed object informs the gateway that it has finished by invoking the **end_of_replies** operation exported by the **OSIMgmt::EndOfRepliesHandler** object in the gateway.
6. The **OSIMgmt::EndOfRepliesHandler** object constructs the final CMIP response and sends it to the remote OSI Manager Application.

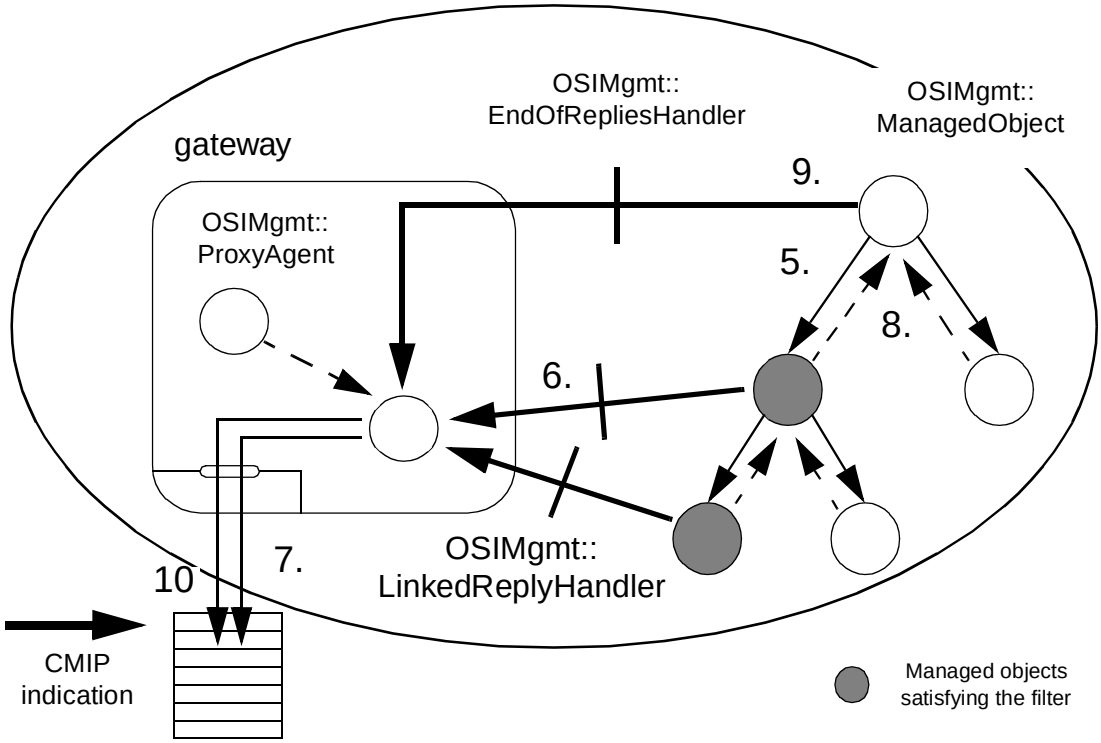

Figure 3-24 Handling CMIP indications with scope and filtering (cont.)

If the base managed object is not part of the OSI Management Application that received the CMIP indication, then the operation is propagated to the list of all local orphan managed objects that are descendants of the base managed object. This list is obtained by invoking the **list_orphan_descendants** operation exported by the local root object.

It is important to point out that the base managed object is informed that there are no pending replies **after** all descendants satisfying the filter have invoked the **send_reply** operation. This way, race conditions are avoided.

Returning from the **send_reply** operation doesn't imply that a CMIP response has already been sent. All responses may be sent together just before the last CMIP response.

The gateway is much simpler to program. Actually, it neither needs to maintain a local copy of the naming tree nor to test which managed objects satisfy the filtering. It only needs to:

- create an object in the CORBA/CMIP gateway process that exports the **LinkedReplyHandler** and the **EndOfRepliesHandler** interfaces.
- invoke the scoped operation exported by the base managed object through the **OSIMgmt::ManagedObject** interface.

Different strategies can be implemented to resolve how each object forwards operations to its descendants and detects when its descendants have no pending replies, but these strategies are transparent to the gateway.

Interactions between managed objects and the CORBA/CMIP gateway are minimized.

3.3.2.6 *Handling m-delete indications*

In CORBA-based OSI managed object domains, **OSIMgmt::ProxyAgent** objects receive CMIP **m-delete** indications, perform the appropriate operations, and return the appropriate CMIP responses.

CMIP **m-delete** indications are handled in a similar way as CMIP **m-set**/**m-get**/**m-action** indications with scoping and filtering. We must take into account that deletion of a single managed object may cause deletion of several objects (all its descendants). Every time a managed object is deleted, its superior object is notified.

Basic algorithm:

1. An **OSIMgmt::ProxyAgent** object receives a CMIP m-delete indication through the CMIS connection endpoint it holds.
2. The **OSIMgmt::ProxyAgent** object finds a reference to the base managed object by invoking the **resolve** operation exported by a **NamingContext** object. The name of the target managed object (base object instance field in the CMIP indication) is passed in the invocation, once it is translated to IDL form (see OSI naming facilities).
3. The **OSIMgmt::ProxyAgent** object locally creates a CORBA object which is responsible for handling the different replies. This object holds the same CMIS connection endpoint as the **OSIMgmt::ProxyAgent** and exports two interfaces:
   - **OSIMgmt::LinkedDeletionHandler**, for handling each deletion.
   - **OSIMgmt::EndOfDeletionsHandler**, for detecting the completion of the deletion.
4. The **OSIMgmt::ProxyAgent** object narrows the CORBA managed object reference to a reference associated with the **OSIMgmt::ManagedObject** interface. Next, it invokes the **scoped_delete** operation visible through that reference.

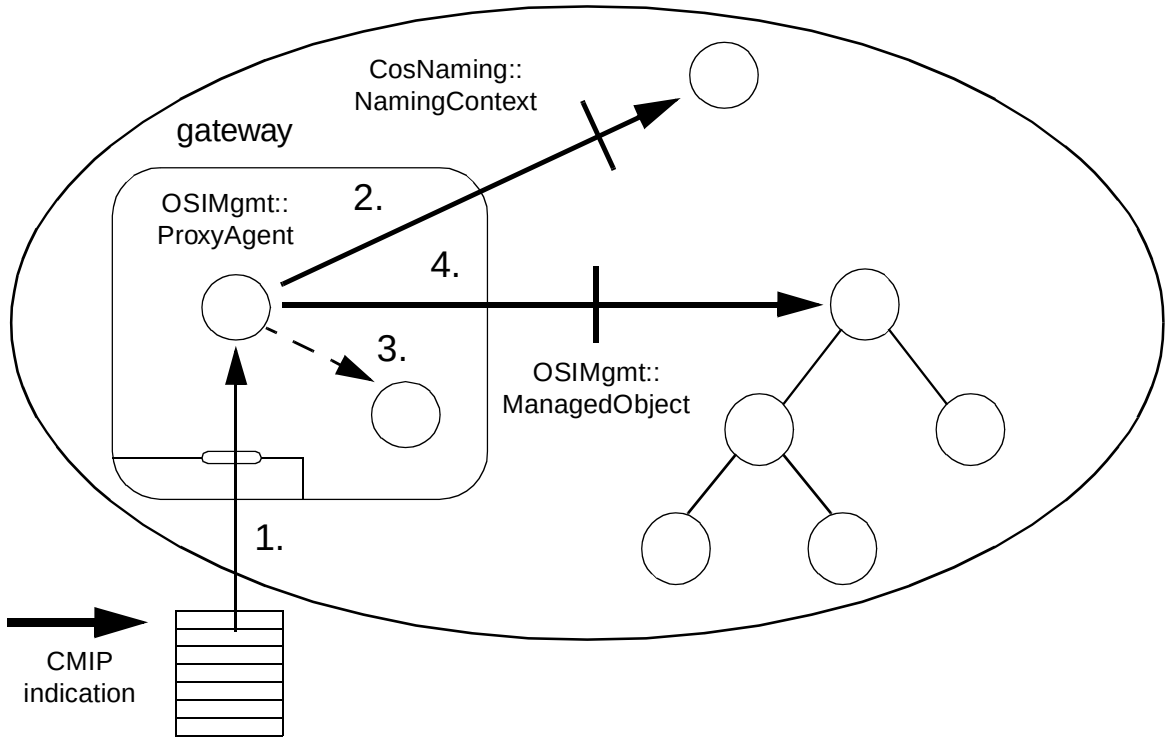

Figure 3-25 Handling CMIP deletion indications

Next, the following steps take place:

1. In some situations, the base managed object propagates the delete operation to part/all of its descendants (the **delete** operation was requested with scope and filtering or deletion of the base managed object implies deletion of all its descendants).

2. For every descendant that is deleted, the **confirm_deletion** operation exported by the **OSIMgmt::LinkedDeletionHandler** object, in the gateway, is invoked. As it was already mentioned before, every managed object is notified when one of its subordinates has been deleted.

3. The **OSIMgmt::LinkedDeletionHandler** object constructs an appropriate CMIP response and sends it back through the CMIS endpoint connection it holds.

4. The base managed object is informed by its subordinates about the completion of the deletion. If appropriate, the base managed object is deleted and a confirmation is sent to the **OSIMgmt::LinkedDeletionHandler** object.

5. The base managed object informs the gateway that the deletion has finished by invoking the **end_of_deletions** operation exported by the **OSIMgmt::EndOfDeletionsHandler** object in the gateway.

6. The **OSIMgmt::EndOfDeletionsHandler** object constructs the final CMIP response and sends it to the remote OSI Manager Application.

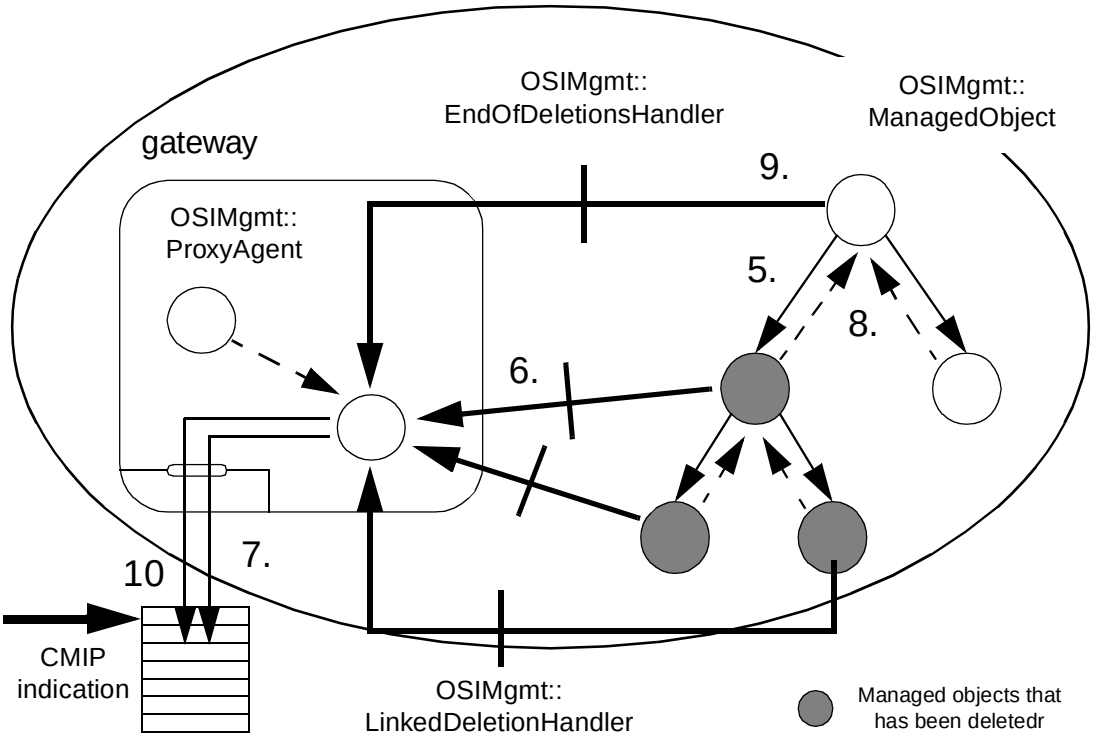

Figure 3-26 Handling CMIP deletion indications (cont.)

In some situations, the base managed object is not part of the OSI Management Application, which received the CMIP m-delete indication but deletion must be applied to some descendants that are part of the application.

This situation is resolved by propagating the **delete** operation to the list of all local orphan managed objects that are descendants of the base managed object. This list is obtained by invoking the **list_orphan_descendants** operation exported by the local root object.

The **scoped_delete** operation may be invoked in either a non-blocking (asynchronous) or blocking (synchronous) mode. A reference to an **OSIMgmt::EndOfDeletionsHandler** object reference (for asynchronous request) or a nil object reference (for synchronous request) should be passed as the last argument in the invocation.

3.3.2.7 *Sending m-event-report requests*

Event Forwarding Discriminators (EFDs) are the managed objects that receive event notifications, emitted by other managed objects within the same OSI managed object domains, and determine which ones are going to be forwarded, as CMIP m-event-reports requests, to specific OSI Manager.

At creation time, an EFD tries to find references to **CosEventChannel::SupplierAdmin** interfaces associated with remote **OSIMgmt::EventPort** objects. It obtains these references by invoking the **find_event_port** operation exported by a **JIDM::EventPortFinder** object, located in a CORBA/CMIP gateway. It can try to find references for:

- various **JIDM::EventPorts**, each of which is bound to one AE_title contained in the list of destinations defined for the **PushEFD**.
- a single **JIDM::EventPort** bound to a wildcard address (only valid if automatic event forwarding - recipient manager resolution is supported).

Note that an EFD may register itself as a **CosEventComm::PushConsumer** or a **CosEventComm::PullConsumer** in the **JIDM::EventPort** associated with each of its assigned destinations.

Different alternatives for EFDs

Definition C-1

```
module X734 {
.........
     interface PushEFD : CosEventComm::PushConsumer,
                                 X711::eventForwardingDiscriminator ...
     {};
     interface PullEFD : CosEventComm::PullConsumer,
                                  X711::eventForwardingDiscriminator ...
     {};
};
```

Different implementations are possible but, in general, managed objects report events by pushing them into local **CosEventChannelAdmin::EventChannels**. EFDs register themselves as event consumers in these event channels.

Multiple implementation strategies can be adopted but they are transparent to remote OSI Manager Applications:

- EFDs may register themselves as **CosEventComm::PushConsumers** or as **CosEventComm::PullConsumers** in **CosEventChannelAdmin::EventChannels**.
- Several **CosEventChannelAdmin::EventChannels** may be employed (connected in cascades, etc). EFDs must know which event channels they must connect to.

Note that only part of the interfaces exported by an EFD are visible across OSI Management boundaries. From external OSI Manager Applications a **PushEFD** and **PullEFD** is just an **X711::eventForwardingDiscriminator**.

3.3.2.8 Sending m-event-report requests (push model)

Basic algorithm:

1. A **PushEFD** registers itself as a **CosEventComm::PushConsumer** in every local event channel that is necessary.
2. CORBA managed objects report events by using the standard event notification services. Each event notification being generated is finally received by some event channel, connected to the **PushEFD** object.
3. The event channel forwards event notifications to the **PushEFD** object by invoking the **push** operation exported by it, through the standard **CosEventComm::PushConsumer** interface.
4. The **PushEFD** object supplies the event to **JIDM::EventPort** objects corresponding to the different destinations.
5. The proxy of an **OSIMgmt::EventPort** in the CORBA/CMIP gateway constructs a CMIP m-event-report request PDU and sends it through the CMIS communication endpoint it holds.

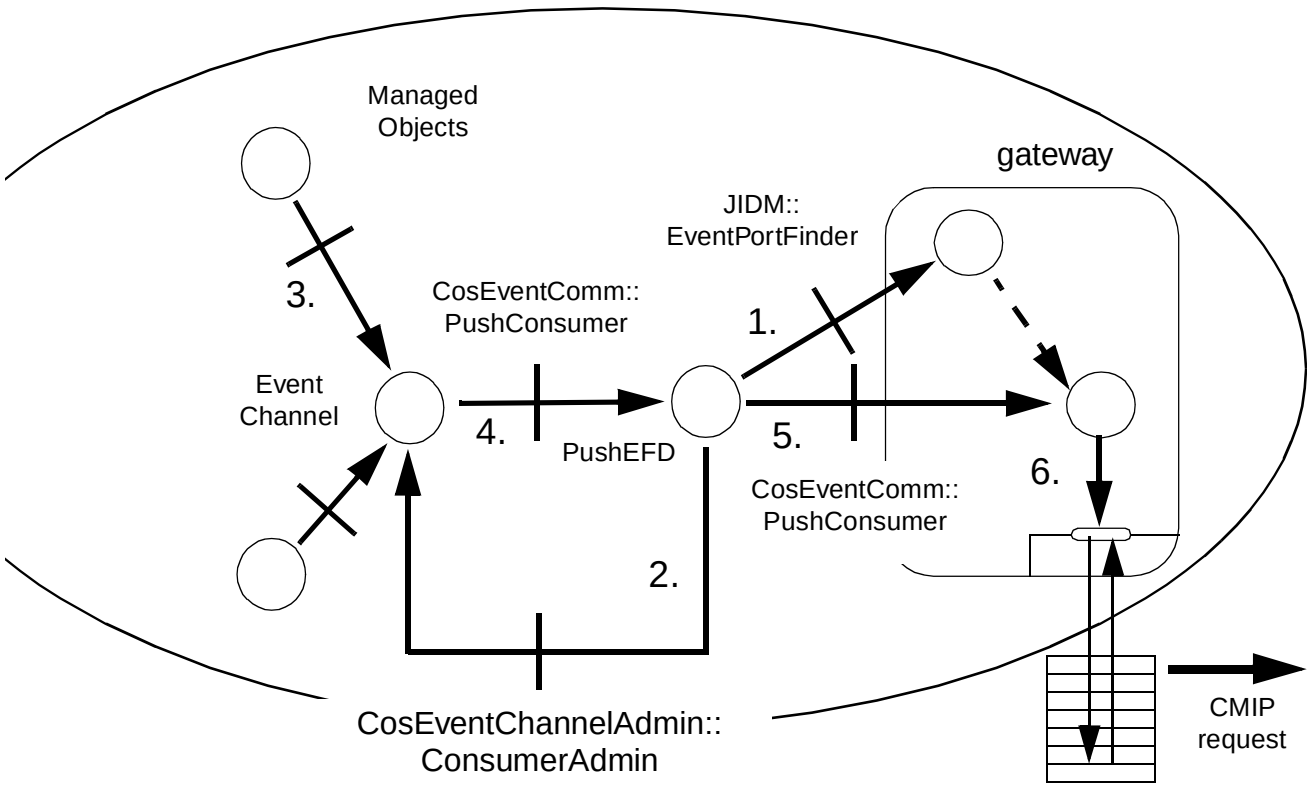

Figure 3-27 Sending m-event-reports (push model)

3.3.2.9 *Sending m-event-report requests (pull model)*

At creation time, each PullEFD tries to find references to **CosEventChannel::SupplierAdmin** interfaces associated with remote **JIDM::EventPort** objects. It obtains these references by invoking the **find_event_port** operation exported by a **JIDM::EventPortFinder** object, located in a CORBA/CMIP gateway. It can try to find references for:

- various **JIDM::EventPorts**, each of which is bound to one AE_title contained in the list of destinations defined for the **PushEFD**.
- a single **JIDM::EventPort** bound to a wildcard address (only valid if automatic event forwarding - recipient manager resolution is supported).

A **PushEFD** registers itself as a **Push** or **Pull** Supplier for each destination.

Basic algorithm:

1. A **PullEFD** registers itself as a **CosEventComm::PullConsumer** in every event channel that is necessary.
2. CORBA managed objects report events by using the standard event notification services. Each event notification being generated is finally received by some event channel, connected to the **PullEFD** object.
3. The **PullEFD** object pulls event notifications received by all event channels where it is registered by means of invoking the pull operation exported by them, through the standard **CosEventComm::PullSupplier** interface.
4. The **PushEFD** object supplies the event to **OSIMgmt::EventPort** objects corresponding to the different destinations.
5. The proxy of an **OSIMgmt::EventPort** in the CORBA/CMIP gateway constructs a CMIP m-event-report request PDU and sends it through the CMIS communication endpoint it holds.

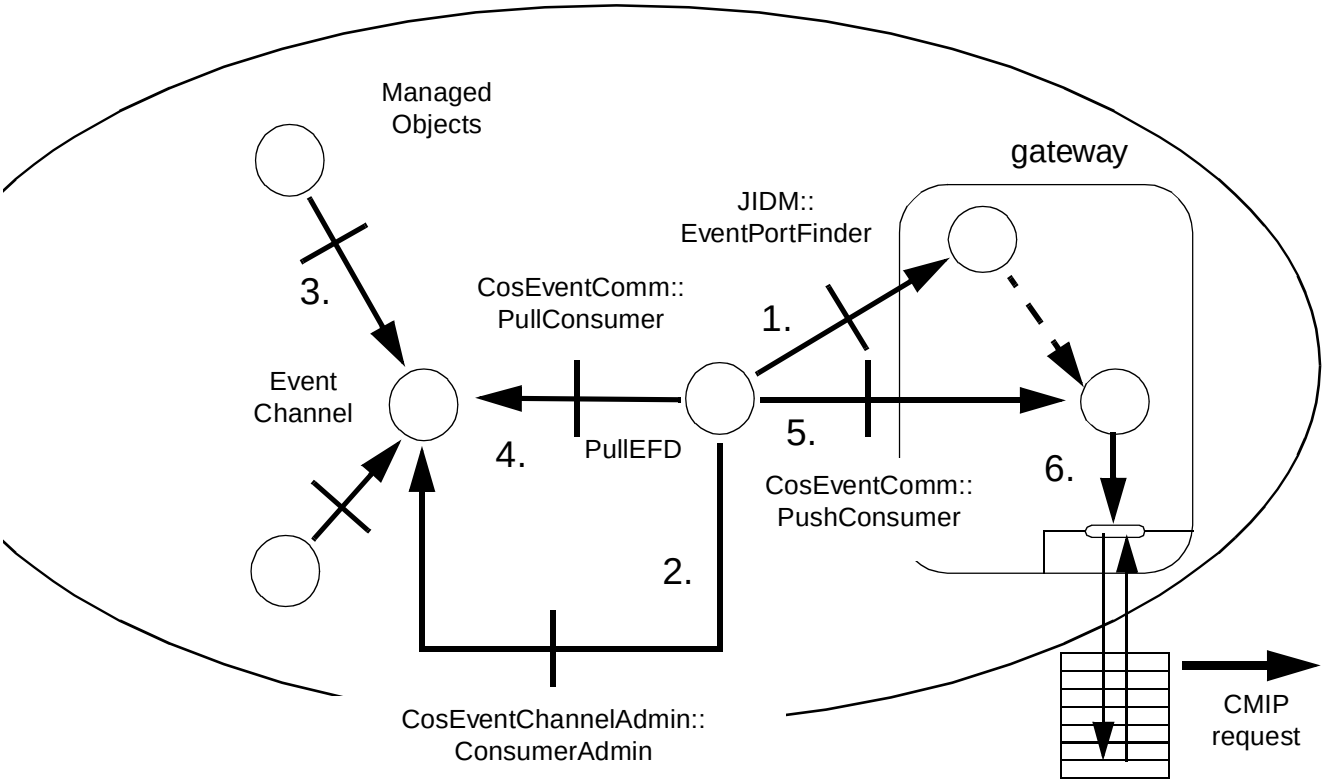

Figure 3-28 Sending m-event-reports (pull model)

OSI Support Services 4

Contents

This chapter contains the following sections.

4.1 OSI Caching and Tracking Services

To provide client applications with fast and efficient access to the values of an attribute of a managed object, it is desirable under certain circumstances to configure a CORBA **ManagedObject** object with the ability to cache information. If so configured, the CORBA **ManagedObject** object can maintain a local store of attribute values, thus eliminating the need to contact the real underlying managed object when this information is requested. This mechanism is referred to in this specification as *caching*. Caching is an optional mechanism that permits applications to avail of improved performance, at the cost of additional resource usage. Caching may be configured on individual managed objects, or on all managed objects of a class, or on all managed objects within a proxy agent.

Any managed object that has been configured with the ability to cache may also be optionally configured with the ability to dynamically update its cached attribute values in response to change notifications received from the underlying managed object. This ability is known as *tracking*. If a managed object is a tracking object, it will update its

cached values in response to the notifications defined by the OSI Systems Management Functions (ObjectCreation, ObjectDeletion, AttributeValueChange, StateChange, and RelationshipChange).

The caching and tracking functionality is intended to provide improved performance. The goals of providing this functionality are:

- transparency: a client application executing normal management operations on a managed object (get, set, action, etc.) must see no difference in behavior (other than performance) whether the operations are executed against a non-caching (regular) managed object, a caching but non-tracking managed object, or a tracking managed object.
- flexibility of configuration: configuration of caching and tracking is available at multiple levels: per proxy agent (all managed objects managed by a given proxy agent), per managed object class (all managed objects of a given object class), and per specific managed object.

4.1.1 The OSICaching Module

```
#ifndef _OSICACHING_IDL_
#define _OSICACHING_IDL_

#include <OSIMgmt.idl>

#pragma prefix “jidm.org”

module OSICaching {
    typedef unsigned long ExpirationInterval; // in seconds
    typedef ASN1_ObjectIdentifier ManagedObjectClass;
    typedef sequence <ManagedObjectClass> ManagedObjectClassSeq;
    typedef ASN1_ObjectIdentifier AttrId;
    typedef sequence < ASN1_ObjectIdentifier > AttrIdSeq;

    // NoSuchAttributes is raised when any specified attribute identifiers
    // are either unknown or invalid.
    exception NoSuchAttributes {
        AttrIdSeq unknown_attributes;
    };

    // AttributesNotCached is raised when any specified attribute identifiers
    // to relevant caching operations are not being cached.
    exception AttributesNotCached {
        AttrIdSeq attr_id_list;
    };

    // NoSuchObjectClasses is raised when any specified object classes are
    // either unknown or invalid.
    exception NoSuchObjectClasses {
        ManagedObjectClassSeq unknown_mocs;
    };


    // ObjectClassesNotCached is raised when any specified object classes
```

```
// to relevant caching operations are not being cached.
exception ObjectClassesNotCached {
      ManagedObjectClassSeq moc_list;
};

// InvalidObjectClassAttributesPairs is raised when any specified attribute
// identifiers do not belong to the specified managed object class.
struct ObjectClassAttributesPair {
      ManagedObjectClass moc;
      AttrIdSeq attr_id_list;
};
typedef sequence<ObjectClassAttributesPair> ObjectClass AttributesPairSeq;
exception InvalidObjectClassAttributesPairs {
      ObjectClassAttributesPairSeq invalid_pairs;
};

/* There may be situations when more than one type of error may occur
* because of a single invocation of an operation. To accurately convey
* the different types of error information, CacheConfigException is used
* by some operations. If any of the members of the following exception
* are not relevant, then such members shall be empty sequences, i.e.,
* sequences of zero length. For example, when passing an argument of
* AttrIdSeq to remove cached attributes , the client may pass some invalid
* or unknown attribute identifiers, and some valid attribute identifiers
* that are not cached. In such situations, CacheConfigException is raised
* with the invalid or unknown attribute identifiers specified in the
* no_such_attributes member, the valid but not cached attribute
* identifiers specified in the attrs_not_cached member, and the rest of
* the members set to zero length sequences.
*/

      exception CacheConfigException {
      AttrIdSeq no_such_attributes;
      ManagedObjectClassSeq no_such_classes;
      AttrIdSeq attrs_not_cached;
      ManagedObjectClassSeq mocs_not_cached;
      ObjectClassAttributesPairSeq invalid_moc_attrs_pairs;
};


// abstract interface for configuring all caches
interface CacheConfigurator {
      void set_default_expiration_interval (
            in ExpirationInterval expiration_interval,
            in boolean override_specific_settings
      );
      ExpirationInterval get_default_expiration_interval ();

      void set_caching_enabled (
            in boolean enabled,
            in boolean override_specific_settings
      );
      boolean is_caching_enabled ();
};
```

```
// cached attribute information
struct CachedAttribute {
      AttrId attr_id;
      ExpirationInterval expiration_interval;
};
typedef sequence < CachedAttribute > CachedAttributeSeq;

// abstract interface to configure per-attribute cache
interface PerAttributeCacheConfigurator {
      void add_cached_attributes (
            in CachedAttributeSeq attr_list,
            in boolean override_specific_settings
      ) raises ( NoSuchAttributes );

      void remove_cached_attributes (
            in AttrIdSeq attr_id_list,
            in boolean override_specific_settings
      ) raises ( CacheConfigException );

      CachedAttributeSeq get_cached_attributes ();

      ExpirationInterval get_expiration_interval (
            in AttrId attr_id
      ) raises ( CacheConfigException );

      void set_expiration_interval(
            in AttrIdSeq attr_id_list,
            in ExpirationInterval interval
      ) raises ( CacheConfigException );
};

// managed object class with indicated attributes cached
struct CachedObjectClass {
      ManagedObjectClass moc;
      CachedAttributeSeq cached_attributes_list;
};
typedef sequence < CachedObjectClass > CachedObjectClassSeq;

// abstract interface to configure per-class cache
interface PerClassCacheConfigurator {
      void add_cached_classes (
            in CachedObjectClassSeq class_list,
            in boolean override_specific_settings
      ) raises ( CacheConfigException );

      void remove_cached_classes (
            in ManagedObjectClassSeq moc_list,
            in boolean override_specific_settings
      ) raises ( CacheConfigException );

      void remove_cached_attributes_from_class_cache(
            in ManagedObjectClass moc,
            in AttrIdSeq attr_id_list,
            in boolean override_specific_settings
      ) raises ( CacheConfigException );
```

```
        CachedObjectClassSeq get_cached_classes ();

        CachedAttributeSeq get_cached_attributes_for_class (
            in ManagedObjectClass moc
        ) raises ( OSIMgmt::NoSuchObjectClass );

        void set_expiration_interval_for_class (
            in ManagedObjectClass moc,
            in AttrIdSeq attr_list,
            in ExpirationInterval extension_duration
        ) raises ( CacheConfigException );
    };

    interface ProxyAgent : OSIMgmt::ProxyAgent,
                           CacheConfigurator,
                           PerAttributeCacheConfigurator,
                           PerClassCacheConfigurator {};


    interface ManagedObject : OSIMgmt::ManagedObject,
                              CacheConfigurator,
                              PerAttributeCacheConfigurator {

        void refresh_cached_values (
            in AttrIdSeq attr_list
        ) raises ( CacheConfigException );

        void invalidate_cached_values (
            in AttrIdSeq attr_list
        ) raises ( CacheConfigException );
    };

};

#endif /* _OSICACHING_IDL_ */
```

4.1.1.1 Description of OSICaching module

The interfaces and methods of the **OSICaching** module permit the configuration of attribute caching at multiple levels. The validity of the cached attribute values may also be controlled by setting appropriate expiration intervals, which can also be controlled at multiple levels.

The **OSICaching::CacheConfigurator**, **OSICaching::PerAttributeCacheConfigurator**, and **OSICaching::PerClassCacheConfigurator** interfaces are all abstract and should not be instantiated directly. Only the **OSICaching::ProxyAgent** and **OSICaching::ManagedObject** interfaces may be instantiated as concrete objects.

The attribute cache of any caching managed object is loaded with attribute values when the cache is initialized. This might happen at different times, such as after successful creation or before first access to the real managed object. Not all attributes of a managed object need to be cached. The list of attributes that are cached for any given

managed object may be configured at multiple levels, including the proxy agent level, the managed object class level, and on the individual managed object itself. The actual list of attributes that are in any given managed object's cache is the union of all the attributes requested to be cached at all these levels. The ability to selectively cache only certain attributes of interest affords the flexibility of not caching attributes that may be changing rapidly and dynamically in the underlying managed object, such as PDU counters, gauges, and clocks.

It is important to note that the cache is a read-only cache, and that there is no access to directly writing the cached attribute value. The managed object implementation automatically updates the cached attribute value when the appropriate attribute-changing or attribute-retrieving operations occur.

If the cached value of an attribute is valid, any request to read the value (such as in a **get** operation) will result in the request being serviced from the cache, thus precluding the need to contact the underlying managed object. If the cached value of an attribute is no longer valid (because its expiration interval has passed since the last update), any request to read the value will trigger an *attribute fault*, and will result in the request being serviced from the actual attribute value in the underlying managed object. A request to read the value of an attribute that is not in the cache is always serviced from the underlying managed object, as is the case for any regular (non-caching) CORBA managed object.

The attribute cache of a managed object is updated with the latest attribute values when a CMIS operation is successfully performed. At this time, the cache reflects the same attribute values that are reported back to the manager in the response, if those attributes are in the cache. Other cached attributes that are not affected by the operation will continue to retain their prior values. As soon as an attribute value in the cache is updated as a result of a CMIS operation, the expiration clock for this attribute is reset, and the duration of validity of this updated value will now be the full duration of the applicable expiration interval; this applies regardless of whether the previous cached value of the attribute had already expired or not. Specifically, the expiration timer for a cached attribute value is reset back to its full applicable duration when any of the following occurs:

- A **get** operation has triggered an attribute fault and has been serviced from the underlying managed object, and the cached attribute value is updated to the same value reported back in the **get** response.
- A **set** operation has been successfully serviced, and the cached attribute value is updated to the same value reported back in the **set** response.
- An explicit request is made to the managed object to refresh the cached attribute value from the underlying managed object.

The duration of validity of any cached attribute value may be configured by the application. An application may cause a cached attribute value to remain permanently valid by setting an expiration interval of 0. This means that the cached attribute value never expires and an attribute fault to read the value from the underlying managed object is never necessary.

The cached value of an attribute is considered to be invalid when any of the following occurs:

- The applicable time interval for the duration of the validity of the cached value has expired.
- An explicit method call is made to invalidate cached attribute values.

In these cases, any attempt to read an attribute value always triggers an attribute fault.

An application may explicitly invalidate the cached values of an **OSICaching::ManagedObject** by invoking the method **invalidate_cached_values()**. This invalidates all its cached values immediately, even if their expiration intervals haven't expired. An application may explicitly cause the cached values of an **OSICaching::ManagedObject** to be refreshed using the current values from the underlying GDMO managed object by invoking the method **refresh_cached_values()**. This also resets the expiration interval of the cached values back to their full duration.

A managed object's cache may be configured at multiple levels. In general, the list of attributes to be cached, and the expiration intervals for each, may be provided at the level of a proxy agent, a managed object class, or an individual managed object. The methods to apply these cache configuration policies are available on **OSICaching::ProxyAgent** and **OSICaching::ManagedObject**. Note that the cache configuration interfaces on **OSICaching::ProxyAgent** imply that the caching policies apply to managed objects controlled by that **ProxyAgent**; they do not imply that the **ProxyAgent** object itself has any caching capabilities.

The levels of cache configuration available are prioritized. From the most general to the most specific, the levels are:

- Proxy Agent: applies to all managed objects controlled by the proxy agent.
- Managed Object Class: applies to all managed objects of that managed object class.
- Managed Object: applies to that individual managed object.

The cache configuration policy specified at a more specific level overrides the cache configuration policy specified at the less specific level. In particular, the following should be noted:

- The list of attributes to be cached for a given managed object is the union of the list of attributes requested to be cached at all applicable levels without the list of attributes that have been specifically removed either using the **override_specific_settings** parameter or using the operations at the managed object level.
- If the same attribute is requested to be cached at multiple levels, the expiration interval for an attribute that is specified at the most specific level applies.

The caching policy in **OSICaching::ProxyAgent** established by the methods inherited from **PerAttributeCacheConfigurator** applies to all indicated attributes wherever they occur in any managed object, regardless of class. This permits the establishment of a caching policy for all occurrences of the same attribute in a proxy agent, even if there is no caching policy established at the managed object class level

or managed object level, in that proxy agent. The caching policy established by the methods in **OSICaching::ProxyAgent** that are inherited from the **PerClassCacheConfigurator** interface apply to all indicated attributes only if they occur in the managed objects of the indicated managed object class.

The methods on **OSICaching::ManagedObject**, including those inherited from **PerAttributeCacheConfigurator**, apply to the actual attributes in the managed object itself. These methods permit the addition or removal of attributes in the cache, changing expiration timeouts, refreshing cached values, or invalidating cached values.

While configuring the caching policy on a **ProxyAgent** or **ManagedObject**, a list of **CachedAttribute** structs (an attribute identifier coupled with an expiration interval), may be supplied. While configuring the caching policy on a **ProxyAgent**, a list of **CachedObjectClass** structs (a class name coupled with a list of **CachedAttribute** structs) may be supplied. If any of these lists is an empty sequence, it is interpreted to mean “all.” In addition, when empty lists are used, default expiration intervals are associated with all relevant attributes. Thus, a caching policy that applies to all attributes in a **ProxyAgent**, a **ManagedObjectClass** or a **ManagedObject**, or a caching policy that applies to all attributes of all managed object classes in a **ProxyAgent**, can be easily enforced by invoking the appropriate methods with an empty list argument.

The methods **CacheConfigurator::set_default_expiration_interval()** and **CacheConfigurator::set_caching_enabled()** permit the caching policy to be changed on a level-wide basis, at either the **ProxyAgent** level or the **ManagedObject** level. If **set_default_expiration_interval()** has never been called, the default expiration interval is considered to be zero (i.e., the cache never expires).

The methods **CacheConfigurator::get_default_expiration_interval()** and **CacheConfigurator::is_caching_enabled()** indicate the state of the level-wide caching policy, at either the **ProxyAgent** level or the **ManagedObject** level.

The **set_default_expiration_interval()** will affect only those attributes that are cache-enabled. The **override_specific_settings** parameter allows the flexibility to either retain any existing custom expiration intervals for some attributes or change the interval to the specified value for all cache-enabled attributes including those that have custom expiration intervals.

This specification permits multiple **ProxyAgent** objects to represent the same underlying TMN agent. It also permits these multiple **ProxyAgent** objects to have different caching and tracking characteristics. For example, one **ProxyAgent** representing a particular TMN agent may have caching and tracking capabilities, while another **ProxyAgent** representing the same TMN agent may not. This implies that a **ManagedObject** accessed via the first ProxyAgent may be configured for caching and tracking, whereas a **ManagedObject** accessed via the second cannot be so configured.

The caching and tracking configuration applied to a **ProxyAgent** object applies to all managed object references obtained from it. It should be noted that an IOR for a **ManagedObject** may be obtained from a **ProxyAgent** in several different ways:

- by doing a **resolve()** on the **NamingService** in that **ProxyAgent**

- by invoking a **create()** operation on the **ManagedObjectFactory** in that **ProxyAgent**

A client application that obtains a **ManagedObject** IOR for a particular underlying GDMO managed object from a caching and tracking **ProxyAgent** will be able to take advantage of caching and tracking, whereas the same client application obtaining another **ManagedObject** IOR for the same underlying GDMO managed object from a non-caching and non-tracking **ProxyAgent** will not be able to take any advantage of caching and tracking. Therefore, caching and tracking capabilities are properties of the IOR of a **ManagedObject** and not necessarily of the underlying managed object itself.

The following rules apply when changes are made to the caching policy after an existing caching policy has already been applied to various attributes.

- Any change to the caching policy (list of cached attributes, attribute expiration intervals, etc.) made on a **ProxyAgent** applies only to **ManagedObject** references *subsequently* obtained from that **ProxyAgent**, unless the parameter **override_specific_settings** is set to TRUE. That is, existing **ManagedObject** object references are not affected. If the parameter **override_specific_settings** is set to TRUE, then the caching policy for all existing **ManagedObject** object references affected by this change in the **ProxyAgent** is updated to the one specified.
- Any change to the caching policy (list of cached attributes, attribute expiration intervals, etc.) made on a **ManagedObjectClass** applies only to *future* **ManagedObject** object references of that class obtained from that **ProxyAgent**, unless the parameter **override_specific_settings** is set to TRUE. That is, existing **ManagedObject** object references of that class are not affected. If the parameter **override_specific_settings** is set to TRUE, then the caching policy for all existing **ManagedObject** object references of that class affected by this change in the **ProxyAgent** is updated to the one specified.
- Any change to the caching policy made on an individual **ManagedObject** object reference is applied immediately to the **ManagedObject** object reference. The parameter **override_specific_settings** is ignored.

Any method that changes the caching and tracking policy may throw exceptions such as **NoSuchAttributes**, **NoSuchClasses**, etc. The semantics of the exception are that all known OIDs are treated as requested, and the unknown OIDs are returned in the exception.

4.1.2 The OSITracking module

```
#ifndef _OSITRACKING_IDL_
#define _OSITRACKING_IDL_

#include <OSICaching.idl>

#pragma prefix “jidm.org”

module OSITracking {
```

```
typedef OSICaching::ManagedObjectClassSeq ManagedObjectClassSeq;
typedef OSICaching::AttrIdSeq AttrIdSeq;
// abstract interface to configure all tracking
interface TrackConfigurator {
      void set_tracking_enabled (

            in boolean enabled,
            in boolean override_specific_settings
      );

boolean is_tracking_enabled ();
};

// abstract interface to configure per-attribute tracking
interface PerAttributeTrackConfigurator {
      void add_tracked_attributes (
            in AttrIdSeq attr_list,
            in boolean override_specific_settings
      ) raises ( OSICaching::NoSuchAttributes );

// If the attr_id_list contains an attribute identifier that is not
// being tracked, then that attribute identifier is ignored
// by remove_tracked_attributes.
void remove_tracked_attributes (
      in AttrIdSeq attr_id_list,
      in boolean override_specific_settings
) raises ( OSICaching::NoSuchAttributes );

AttrIdSeq get_tracked_attributes ();
};

// managed object class with indicated attributes tracked
struct TrackedObjectClass {
      OSICaching::ManagedObjectClass moc;
      AttrIdSeq list_of_tracked_attributes;
};

typedef sequence < TrackedObjectClass > TrackedObjectClassSeq;

// TrackConfigException is similar in purpose to
// OSICaching::CacheConfigException
exception TrackConfigException {
      ManagedObjectClassSeq no_such_mocs;
      AttrIdSeq no_such_attr_ids;
      OSICaching::ObjectClassAttributesPairSeq invalid_moc_attrs_pairs;
};

// abstract interface to configure per-class tracking
interface PerClassTrackConfigurator {
      void add_tracked_classes (
            in TrackedObjectClassSeq class_list,
            in boolean override_specific_settings
      ) raises ( TrackConfigException );
```

```
        void remove_tracked_classes (
            in ManagedObjectClassSeq moc_list,
            in boolean override_specific_settings
        ) raises ( OSICaching::NoSuchObjectClasses );

        TrackedObjectClassSeq get_tracked_classes ();

        AttrIdSeq get_tracked_attributes_for_class (
            in OSICaching::ManagedObjectClass class_name
        ) raises ( OSIMgmt::NoSuchObjectClass );
        };

        interface ProxyAgent : OSICaching::ProxyAgent,
                    TrackConfigurator,
                    PerAttributeTrackConfigurator,
                    PerClassTrackConfigurator {};

        interface ManagedObject : OSICaching::ManagedObject,
                        TrackConfigurator,
                        PerAttributeTrackConfigurator {};

};

#endif /* _OSITRACKING_IDL_ */
```

4.1.2.1 Description of the OSITracking module

The tracking mechanism is an extension to the caching mechanism specified above to permit the dynamic update of critical information, without the need for any application intervention.

A tracked attribute in a managed object is an attribute whose value in the cache is dynamically updated by way of notifications received from the underlying managed object. The **ProxyAgent** implementation dynamically updates its cache based on any information made available to it in notifications emitted by the underlying managed object, including at least the standard Systems Management notifications of object creation, object deletion, attribute value change, state change, and relationship change.

When a cached attribute value is dynamically updated as a result of notification tracking, the expiration timer for that cached attribute value is reset back to its full applicable duration of validity, regardless of whether the prior cached value was still valid or had expired.

As with caching, tracking may be configured on a per proxy agent basis, per managed object class basis, and per individual managed object basis. These have the same meaning and same levels of overriding as caching.

An attribute value in a managed object may only be tracked if it is also cached. In particular, the list of attributes configured to be tracked must have been configured to also be in the cache at a prior time. Any attributes that are requested to be tracked at any given level, but are not in the cache configuration at that particular level, are ignored.

4.1.3 Mechanism to obtain Cached/Tracked services

The mechanism to obtain cached/tracked services is an extension to the one used to obtain access to any managed domain. That is, using the **ProxyAgentFinder::access_domain** operation.

An additional set of criteria are specified to gain access to these value added services, if available. These criteria are specified in the following tables. Table 4-1 repeats the basic **OSIMgmt** criteria needed to create an **OSIMgmt::ProxyAgent**.

For use with caching and tracking, the criteria specified in Table 4-2 on page 4-13 may replace these, by specifying an already existing **OSIMgmt::ProxyAgent**, that represents the domain to be cached (and maybe tracked). Destruction of the original **OSIMgmt::ProxyAgent** is the responsibility of the application that created it.

Table 4-1 Basic OSIMgmt ProxyAgentFinder Criteria

criterion name	type of value	meaning
“domain title”	**X227ACS::AE_titleType**	AE-title associated to the managed object domain for which access is requested. The wildcard address is allowed.
“controller object”	**JIDM::ProxyAgentController**	reference associated to a **JIDM::ProxyAgentController** object registered by the manager (OPTIONAL)
“access control”	**X711CMI::AccessControlType**	Information to be used as input to access control functions in establishing default access privileges for all exchanges on the association (OPTIONAL)
“requestor title”	**X227ACS::AE_titleType**	Title used to denote the Manager which requested access to the OSI managed object domain (OPTIONAL)

Table 4-2 Alternative for caching/tracking OSIMgmt ProxyAgentFinder Criteria

criterion name	type of value	meaning
“proxy agent”	**OSIMgmt::ProxyAgent**	Already existing ProxyAgent to which caching is to be applied.

criterion name	type of value	meaning
“controller object”	**JIDM::ProxyAgentController**	Reference associated to a **JIDM::ProxyAgentController** object registered by the manager, that controls destruction of the cached/tracked **ProxyAgent** (OPTIONAL).

In addition to one of the above, the criteria specified in Table 4-3 must also be provided for caching (and tracking) to be activated.

The values passed with the criteria names (“caching” and “tracking”) are implementation-defined. Each implementation should specify what the appropriate values for these criteria should be.

If caching is specified for a **ProxyAgent** using the criteria specified in Table 4-3, then it is an implementation issue whether there are really two separate **ProxyAgent**s, one that does not support caching and another that does.

Table 4-3 Additional criteria needed for caching/tracking OSIMgmt ProxyAgentFinder

criterion name	type of value	meaning
“caching”	any	Caching is enabled; the value is implementation dependent.
“tracking”	any	Tracking is enabled; the value is implementation dependent (OPTIONAL).

4.2 Collection Service

4.2.1 Overview

The OSI Collection service specification provides facilities to manipulate arbitrary sets of **OSIMgmt::ManagedObjects**.

This service is patterned after the CORBA Collection Service specification and re-uses some of the concepts defined there. However, no direct interface re-use is attempted to be able to provide highly typed interfaces, specific to OSI management environments, and that takes into account the distributed nature of the collections being manipulated.

It defines an **OSICollection::Iterator** interface, to be able to navigate, traverse, and manipulate the collections, and two different types of OSI collections:

- *Enumerated collection*, in which the collection membership is explicitly controlled by the client; any arbitrary set of managed objects may be added to the collection, according to any grouping criterion convenient to the application.

- *Rule collections*, in which the collection membership is defined by some rule (in this case, a scope and filter specification). All managed objects that satisfy the rule are the members of the rule collection.

4.2.2 The OSICollection Module

```
#ifndef _OSICOLLECTION_IDL_
#define _OSICOLLECTION_IDL_

#include <OSIMgmt.idl>

#pragma prefix "jidm.org"

module OSICollection {
    typedef OSIMgmt::ManagedObject ManagedObject;
    typedef sequence < ManagedObject > ManagedObjectSeq;
    exception IteratorInvalid { };
    exception IteratorInBetween { };
    exception CollectionInvalid { };
    exception NotFound { };
    exception InvalidName { };

    interface Iterator {
        // retrieving elements
        boolean  get_element (
            out ManagedObject mo
        ) raises ( IteratorInvalid, IteratorInBetween );
        boolean  get_n_elements (
            in unsigned long how_many,
            out ManagedObjectSeq mo_list
        ) raises ( IteratorInvalid );

        // moving iterator
        void restart () raises ( IteratorInvalid );
        void set_to_next_element () raises ( IteratorInvalid );
        void set_to_next_nth_element (
            in unsigned long how_many
        ) raises ( IteratorInvalid );

        // iterator state
        void invalidate ();
        boolean is_valid ();
        boolean is_in_between ();
        boolean is_equal ( in Iterator other ) raises ( IteratorInvalid );
        // cloning, assigning and destroying
        Iterator clone ();
        void assign ( in Iterator from_where ) raises ( IteratorInvalid );
        void destroy ();
    };

    typedef OSIMgmt::LinkedReplyHandler LinkedReplyHandler;
    typedef OSIMgmt::EndOfRepliesHandler EndOfRepliesHandler;

    // abstract base interface
```

```
interface BaseCollection {
    // operations to perform on all elements in the collection
    void perform_get (
        in OSIMgmt::ASN1_ObjectIdentifierSeq attr_id_list,
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );
    void perform_set (
        in OSIMgmt::SetOperationArgument modif_list,
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );
    void perform_action (
        in ASN1_ObjectIdentifier action_id,
        in ASN1_DefinedAny action_info,
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );
    void perform_delete (
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );

    // statistics
    boolean is_empty ();


    // creating iterators
    Iterator create_iterator (
        in boolean read_only
    ) raises ( CollectionInvalid );

    // destruction
    void destroy ();
};

interface EnumCollection : BaseCollection {
    // adding elements
    void add_element ( in ManagedObject element );
    void add_elements ( in ManagedObjectSeq elem_list );
    void add_all_from ( in BaseCollection collection );

    // removing elements
    void remove_element_at (
        in Iterator where
    ) raises ( IteratorInvalid, IteratorInBetween );
    void remove_all ();
};

interface RuleCollection : BaseCollection {
    ManagedObject get_base_object () raises ( CollectionInvalid );
    X711CMI::ScopeType get_scope () raises ( CollectionInvalid );
    X711CMI::CMISFilterType get_filter () raises ( CollectionInvalid );
    X711CMI::CMISSyncType get_synchronization () raises ( CollectionInvalid );
};
```

```
    interface CollectionFactory {
        EnumCollection create_enum_collection ();

        EnumCollection create_enum_collection_from_collection (
           in BaseCollection collection
        );

        RuleCollection create_rule_collection (
            in OSIMgmt::ManagedObject base_managed_object,
            in X711CMI::ScopeType scope,
            in X711CMI::CMISFilterType filter,
            in X711CMI::CMISSyncType sync
        );


        RuleCollection create_rule_collection_by_name (
            in OSIMgmt::ProxyAgent proxy_agent,
            in CORBA::ScopedName base_mo_interface,
            in CosNaming::Name base_mo_name,
            in X711CMI::ScopeType scope,
            in X711CMI::CMISFilterType filter,
            in X711CMI::CMISSyncType sync
        );
    };
};

#endif /* _OSICOLLECTION_IDL_ */
```

4.2.2.1 *Descriptions of OSICollection types and operations*

The Iterator interface

The **OSICollection::Iterator** interface is similar to **OSIMgmt::RepliesIterator** (see Chapter 3) in terms of navigation operations (**get_element**, **get_n_elements**). The semantics and behavior of these operations are the same as the equivalent operations specified in the **OSIMgmt::RepliesIterator** interface (**get_reply**, **get_n_replies**).

This interface also adds a number of operations for different purposes:

- to manipulate the iterator position:
  - **restart**, resets the iterator to point to the first element in the collection.
  - **set_to_next_element**, moves the iterator to the next element in the collection.
  - **set_to_next_nth_element**, moves the iterator n times.
- to control and monitor the state of the iterator:
  - **invalidate**, sets the state of the iterator to invalid.
  - **is_valid**, checks validity of the iterator.
  - **is_in_between**, checks whether the iterator points to an element, or is in between elements.
  - **is_equal**, checks if both iterators belong to the same collection and point to the same element.

- to manipulate the lifecycle of iterators:
  - **clone**, makes an exact copy of the iterator.
  - **assign**, changes the state of the iterator to match the one assigned to it.
  - **destroy**, destroys the iterator.

Note that **OSICollection::Iterators** are "*managed iterators*," as explained in the CORBA Collection Service specification. This means that the status of the iterator is always known, never undefined, specifically in the event the collection contents change.

The possible iterator states are:

- **valid**, pointing to an element of the collection.
- **invalid**, pointing to nothing (for example, past the end of the collection).
- **in-between**, not pointing to an element but knowing what its position in the collection is.

A valid iterator remains valid as long as the element it points to remains in the collection. If the element is removed, the iterator goes to the in-between state. Certain operations require the iterator to be in a valid state (like all the ***_at** operations), while others work even when the iterator is in an in-between state (like **get_n_elements**). An iterator becomes invalid when it points to nothing (for example, it has been moved past the last element).

The BaseCollection interface

The **OSICollection::BaseCollection** is an abstract interface specification (i.e., that may not be instantiated by itself) that provides the functionality that is common to all collections (enumerated or rule-based).

Specifically, it provides operations to allow:

- manipulation of the collection and its elements
  - **is_empty**, identifies whether the collection has any elements
  - **create_iterator**, creates an iterator on the collection
  - **destroy**, destroys of the collection
- specific operations to be performed on all objects in the collection
  - **perform_get**, **perform_set**, **perform_action**, **perform_delete**

  All these operations perform the corresponding CMIS operations on all the objects in the collection (one per object in enumerated collections, or one for a whole rule collection), returning the results via the **RepliesHandlers** passed in as parameters. For a more detailed discussion of both the different CMIS operations and the handler callback objects, see the corresponding OSI Mgmt sections in this specification. Also, the semantics assigned to the presence or absence of the handler object references are those specified in the OSIMgmt sections, as well.

If an object in a collection returns the standard CORBA exception OBJECT_NOT_EXIST when performing one of these global operations, the collection will immediately remove the object from the collection.

The EnumCollection interface

The **OSICollection::EnumCollection** interface provides a general mechanism to group (otherwise not necessarily related) managed objects. Specifically, all the managed objects in a single collection are *not* required to belong to the same managed object domain (i.e., their references might have been obtained through different **OSIMgmt::ProxyAgents**).

The member managed objects of an enumerated collection are added and removed from the collection on an individual basis (**add_element**, **remove_element_at**), or in batch mode (**add_elements**, **add_all_from**, **remove_all**). It is important to note that an **OSICollection::Collection** object does not maintain any specified order among its members. No assumption should be made regarding the retrieval order, as compared to the insertion order. However, ordering within the collection is implementation-defined; this means that any two traversals of an iterator over the same collection will return managed objects in exactly the same order if no membership manipulation has occurred in between the traversals.

The RuleCollection interface

The **OSICollection::RuleCollection** interface has its membership defined by its governing rule. Specifically, the rule is a scope and filter specification, specified to the collection factory at the time the rule collection is created. The governing rule of a rule collection is defined by:

- a base object, that specifies the base of the scoping specification for this collection,
- a scope specification, as defined in [X720] and in Chapter 4 OSIMgmt, and
- a filter specification, as defined in [X720] and in Chapter 4 OSIMgmt.

Once created, the governing rule of a rule collection cannot be changed, because doing so may invalidate the existing collection membership. When a CMIS operation is invoked on a rule collection, the indicated base managed object, scope, and filter that form the collection's governing rule are the parameters used to invoke the scoped operation on the actual agent or agents that are spanned by that rule.

A CMIS synchronization (atomic or best effort) may also be specified for a rule collection. This synchronization parameter is used in the scoped request that is sent to the agent(s) when a CMIS operation is invoked on the rule collection.

When an iterator is requested from a rule collection (via the **create_iterator()** method), then a snapshot of the collection's membership is taken to serve the iterator. This snapshot must reflect the best available knowledge of the managed objects that meet the collection's defining rule at that time.

The CollectionFactory Interface

The **OSICollection::CollectionFactory** interface provides methods to create **EnumCollection** and **RuleCollection** objects.

In particular, the **CollectionFactory** interface allows:

- Creation of an empty **EnumCollection** (**create_enum_collection()**).

- Creation of an **EnumCollection** as a copy of another **Collection**. If the copied collection is a **RuleCollection**, a snapshot of its membership is taken at the time of creation of the **EnumCollection** (**create_enum_collection_from_collection()**).
- Creation of a **RuleCollection** specifying its base managed object and scoping/filtering rule (**create_rule_collection()**).
- Creation of a **RuleCollection** specifying its base managed object by reference to an **OSIMgmt::ProxyAgent** and the class and instance names of the base object, plus the scoping/filtering rule (**create_rule_collection_by_name()**).

4.3 Dynamic Management of ASN.1 Any Values

4.3.1 Overview

The ***Dynamic Management of CORBA::Any values*** facility, introduced in the *CORBA 2.2* specification, enables the manipulation of **CORBA::Any** values at runtime, without having any static information (generated by an IDL compiler) about the type being carried inside the **Any**.

This facility extends the one above to support **CORBA::Any** values that originate from an ASN.1 specification that has been translated into IDL via the JIDM Specification Translation algorithm (see [XOJIDM]) in a way that is closer to the original ASN.1 type.

Note that all operations that may be performed through this interface may also be performed directly using the basic **CORBA::DynAny** interface. Additional functionality provided by this interface is the access to ASN.1 specific information, that might have been lost in the translation from ASN.1 to IDL (specifically, type constraints), and the ability to use the original ASN.1 names, instead of the translated IDL names. Also, mechanisms are provided to deal with common ASN.1 constructs such as OPTIONAL, DEFAULT, and anonymous elements in an easier manner.

The behavior of DynAny objects has been defined in such a way as to enable efficient implementations in terms of allocated memory space and speed of access.

In order for this interface to be fully operational and provide the above mentioned advantages, some mechanism (not specified here) to access cross-domain information is needed. The most likely scenario for this is the use of an OSI MIR (see Section 4.4, “The OSI Management Information Repository,” on page 4-27), if available. However, this facility does not require the use of an OSI MIR (other implementation mechanisms are possible alternatives).

The **ASN1::DynAny** IDL is patterned after the **CORBA::DynAny** IDL, with the following differences:

- Defined within an ASN1 IDL module, rather than within the CORBA module.
- The factory for these objects is explicitly defined as a CORBA object, rather than using the ORB pseudo interface. The way to get a reference to such factory is implementation specific.

- Names of **ASN1::DynAny** subtypes resemble the names of the ASN.1 construct, not those of IDL.
- It inherits from the **CORBA::DynAny** interface, and redefines some of the behaviors.
- While the **CORBA::DynAny** interface relies on the **CORBA::TypeCode** of the value being processed, the **ASN1::DynAny** reuses that information, plus that provided by a simpler **ASN1::Kind** type.

4.3.2 The ASN1 Module

```
#ifndef _ASN1_IDL_
#define _ASN1_IDL_

#include <orb.idl>
#include <ASN1Types.idl>

#pragma prefix “jidm.org”
module ASN1 {

  typedef CORBA::Identifier Identifier;

  enum Kind {
    ak_none, // used when value is not ASN.1 based
    ak_null, ak_boolean,
    ak_integer, ak_real,
    ak_numericstring, ak_printablestring,
    ak_visiblestring, ak_iso646string,
    ak_graphicstring, ak_objectdescriptor,
    ak_teletexstring, ak_t61string,
    ak_generalizedtime, ak_utctime,
    ak_octetstring, ak_generalstring,
    ak_ia5string, ak_videotexstring,
    ak_bmpstring, ak_universalstring,
    ak_objectidentifier,
    ak_bitstring,
    ak_any, ak_definedany,
    ak_external,
    ak_enum,
    ak_sequence, ak_set,
    ak_sequenceof, ak_setof,
    ak_choice
  };


interface DynAny : CORBA::DynAny {
    Kind asn1_kind() raises (Invalid);
    Identifier asn1_type_name () raises (Invalid);
    Identifier asn1_module_name() raises (Invalid);
    Identifier asn1_module_nickname() raises (Invalid);
    ASN1_ObjectIdentifier asn1_module_oid() raises (Invalid);

    void asn1_assign (in DynAny asn1_dyn_any) raises (Invalid);
```

```
    void from_dyn_any (in CORBA::DynAny dyn_any) raises (Invalid);
    CORBA::DynAny to_dyn_any() raises (Invalid);
    DynAny asn1_copy();

  interface DynAny : CORBA::DynAny {
    Kind asn1_kind() raises (Invalid);
    Identifier asn1_type_name () raises (Invalid);
    Identifier asn1_module_name() raises (Invalid);
    Identifier asn1_module_nickname() raises (Invalid);
    ASN1_ObjectIdentifier asn1_module_oid() raises (Invalid);
     void asn1_assign (in DynAny asn1_dyn_any) raises (Invalid);
    void from_dyn_any (in CORBA::DynAny dyn_any) raises (Invalid);

    CORBA::DynAny to_dyn_any() raises (Invalid);
    DynAny asn1_copy();

    void insert_asn1_null(in ASN1_Null value) raises(InvalidValue);
    void insert_asn1_boolean(in ASN1_Boolean value)
            raises(InvalidValue);
    void insert_asn1_unsigned16(in ASN1_Unsigned16 value)
            raises(InvalidValue);
     void insert_asn1_unsigned(in ASN1_Unsigned value)
            raises(InvalidValue);
     void insert_asn1_unsigned64(in ASN1_Unsigned64 value)
            raises(InvalidValue);
     void insert_asn1_integer16(in ASN1_Integer16 value)
            raises(InvalidValue);
      void insert_asn1_integer(in ASN1_Integer value)
            raises(InvalidValue);
     void insert_asn1_integer64(in ASN1_Integer64 value)
            raises(InvalidValue);
    void insert_asn1_real(in ASN1_Real value) raises(InvalidValue);
    void insert_asn1_numericstring(in ASN1_NumericString value)
            raises(InvalidValue);


void insert_asn1_printablestring(in ASN1_PrintableString value)
            raises(InvalidValue);
     void insert_asn1_visiblestring(in ASN1_VisibleString value)
            raises(InvalidValue);
     void insert_asn1_iso646string(in ASN1_ISO646String value)
            raises(InvalidValue);
     void insert_asn1_graphicstring(in ASN1_GraphicString value)
            raises(InvalidValue);
     void insert_asn1_objectdescriptor(in ASN1_ObjectDescriptor value)
            raises(InvalidValue);
     void insert_asn1_teletexstring(in ASN1_TeletexString value)
            raises(InvalidValue);
     void insert_asn1_t61string(in ASN1_T61String value)
            raises(InvalidValue);

     void insert_asn1_generalizedtime(in ASN1_GeneralizedTime value)
            raises(InvalidValue);
     void insert_asn1_utctime(in ASN1_UTCTime value)
            raises(InvalidValue);
```

```
void insert_asn1_octetstring(in ASN1_OctetString value)
        raises(InvalidValue);
void insert_asn1_generalstring(in ASN1_GeneralString value)
        raises(InvalidValue);
 void insert_asn1_ia5string(in ASN1_IA5String value)
        raises(InvalidValue);
void insert_asn1_videotexstring(in ASN1_VideotexString value)
        raises(InvalidValue);

void insert_asn1_bmpstring(in ASN1_BMPString value)
        raises(InvalidValue);
void insert_asn1_universalstring(in ASN1_UniversalString value)
        raises(InvalidValue);

void insert_asn1_objectidentifier(in ASN1_ObjectIdentifier value)
        raises(InvalidValue);

void insert_asn1_bitstring(in ASN1_BitString value)
        raises(InvalidValue);

void insert_asn1_any(in ASN1_Any value) raises(InvalidValue);
void insert_asn1_definedany(in ASN1_DefinedAny value)
        raises(InvalidValue);

void insert_asn1_external(in ASN1_External value)
        raises(InvalidValue);


ASN1_Null get_asn1_null() raises(TypeMismatch);
ASN1_Boolean get_asn1_boolean() raises(TypeMismatch);

ASN1_Unsigned16 get_asn1_unsigned16() raises(TypeMismatch);
ASN1_Unsigned get_asn1_unsigned() raises(TypeMismatch);
ASN1_Unsigned64 get_asn1_unsigned64() raises(TypeMismatch);
ASN1_Integer16 get_asn1_integer16() raises(TypeMismatch);
ASN1_Integer get_asn1_integer() raises(TypeMismatch);
ASN1_Integer64 get_asn1_integer64() raises(TypeMismatch);

ASN1_Real get_asn1_real() raises(TypeMismatch);

ASN1_NumericString get_asn1_numericstring()
        raises(TypeMismatch);
ASN1_PrintableString get_asn1_printablestring()
            raises(TypeMismatch);
ASN1_VisibleString get_asn1_visiblestring() raises(TypeMismatch);
ASN1_ISO646String get_asn1_iso646string() raises(TypeMismatch);
ASN1_GraphicString get_asn1_graphicstring()
        raises(TypeMismatch);
ASN1_ObjectDescriptor get_asn1_objectdescriptor()
        raises(TypeMismatch);
ASN1_TeletexString get_asn1_teletexstring() raises(TypeMismatch);
ASN1_T61String get_asn1_t61string() raises(TypeMismatch);
ASN1_GeneralizedTime get_asn1_generalizedtime()
            raises(TypeMismatch);
```

```
ASN1_UTCTime get_asn1_utctime() raises(TypeMismatch);

ASN1_OctetString get_asn1_octetstring() raises(TypeMismatch);
ASN1_GeneralString get_asn1_generalstring()
        raises(TypeMismatch);
ASN1_IA5String get_asn1_ia5string() raises(TypeMismatch);
ASN1_VideotexString get_asn1_videotexstring()
        raises(TypeMismatch);

ASN1_BMPString get_asn1_bmpstring() raises(TypeMismatch);
ASN1_UniversalString get_asn1_universalstring()
        raises(TypeMismatch);

ASN1_ObjectIdentifier get_asn1_objectidentifier()
        raises(TypeMismatch);

ASN1_BitString get_asn1_bitstring() raises(TypeMismatch);


ASN1_Any get_asn1_any() raises(TypeMismatch);
ASN1_DefinedAny get_asn1_definedany() raises(TypeMismatch);

ASN1_External get_asn1_external() raises(TypeMismatch);

ASN1_Any current_asn1_component () raises(Invalid);
};

interface DynEnum: DynAny, CORBA::DynEnum {
    attribute string value_as_asn1_identifier;
    attribute long value_as_asn1_value;
};

interface DynNamedNumber: DynAny {
    attribute string value_as_asn1_identifier;
};

typedef CORBA::FieldName FieldName;
typedef CORBA::NameValuePairSeq NameValuePairSeq;

interface DynSetSeq: DynAny, CORBA::DynStruct {
    FieldName current_asn1_elem_name ();
    Kind current_asn1_elem_kind ();
    NameValuePairSeq get_asn1_elems() raises(Invalid);
    void set_asn1_elems(in NameValuePairSeq value)
        raises (InvalidSeq);
    void insert_optional_absent() raises (InvalidValue);
    DynAny insert_optional_present() raises (InvalidValue);
    void insert_default_absent() raises (InvalidValue);
    DynAny insert_default_present() raises (InvalidValue);
    boolean get_optional_presence() raises (TypeMismatch);
    DynAny get_optional_present() raises (TypeMismatch);
    boolean get_default_presence() raises (TypeMismatch);
    DynAny get_default_present() raises (TypeMismatch);
};
```

```
    interface DynChoice: DynAny, CORBA::DynUnion {
        DynAny asn1_elem ();
        attribute FieldName asn1_elem_name;
        Kind asn1_elem_kind ();
    };

    interface DynAnyFactory {
        exception InconsistentKind {};
        exception InconsistentTypeCode {};


        typedef CORBA::Identifier Identifier;

        DynAny create_asn1_dyn_any(in any value);

        DynAny create_basic_dyn_any(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynStruct create_dyn_struct(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynSequence create_dyn_sequence
                    (in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
    CORBA::DynUnion create_dyn_union(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
    CORBA::DynEnum create_dyn_enum(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
    CORBA::DynArray create_dyn_array(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
    CORBA::DynFixed create_dyn_fixed(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
            DynAny create_asn1_dyn_primitive(in Identifier asn1_nickname,
                    in Identifier asn1_name)
                raises(InconsistentKind);
            DynEnum create_asn1_dyn_enum(in Identifier asn1_nickname,
                    in Identifier asn1_name)
                raises(InconsistentKind);
            DynSetSeq create_asn1_dyn_setseq(in Identifier asn1_nickname,
                    in Identifier asn1_name)
                raises(InconsistentKind);
            DynSetSeqOf create_asn1_dyn_setseqof(in Identifier
                        asn1_nickname,
                    in Identifier asn1_name)
                raises(InconsistentKind);
    DynChoice create_asn1_dyn_choice(in Identifier asn1_nickname,
                    in Identifier asn1_name)
                raises(InconsistentKind);
  };

};

#endif /* _ASN1_IDL_ */
```

Note that all types, derived from ASN.1 or not, can be manipulated through this interface. In case the type comes from ASN.1, then extra operations might be available (if needed), that could help to manipulate the value.

However, note that ALL operations are possible through the unextended **CORBA::DynAny** interface. The added value provided by the **ASN1::DynAny** extension is the fact that ASN.1 constraints might be checked by the implementation, if possible (this is not a mandatory conformance point for this facility, but a quality of implementation issue).

4.3.2.1 *Description of ASN1 types and operations*

The Kind type

The **ASN1::Kind** type identifies the ASN.1 type, which is held by a **DynAny** object. The specification provides functions in all modules to access the kind(s) at each level.

If the **CORBA::TypeCode** does not correspond to an ASN.1 type, then the special kind of **ak_none** is used. In this case, none of the extended interfaces may be used (they will all return an appropriate exception: Invalid, InvalidValue of TypeMismatch).

The Exceptions

The following inherited exceptions are used:

- Invalid means either the **ASN1::DynAny** is not initialized (for read operations) or it is incompatible with the operation being performed.
- InvalidValue means trying to insert the wrong type/value. This exception would also be raised in case an ASN.1 constraint is not satisfied when inserting a value.
- TypeMismatch means trying to extract the wrong type.
- InvalidSeq means the sequence used does not have the appropriate structure or types.

The Type identification

Besides getting the CORBA TypeCode, the **ASN.1::DynAny** interfaces provide methods to get the ASN.1 kind, type name, module name, OID, and module nickname. If this is not an ASN.1 type, these would raise Invalid (except the **ASN1::Kind** would be **ak_none**); if unknown, they would be empty.

The Lifecycle

Equivalent functions to the ones provided in **CORBA::DynAny**, with the same semantics.

The Insertion operations

The inherited operations will work based on the **CORBA::TypeCode** and for all types (ASN.1 or otherwise).

There are different insertion operations per primitive ASN.1 type, even if they map to the same IDL type. In this way, the interface is more type safe (in the ASN.1 sense).

For example, if we have **MyType::=INTEGER(1..500)**, then either **insert_ushort** or **insert_asn1_unsigned16** would work. And both could check for the bounds, and return InvalidValue if the constraint is violated.

Note that there is nothing to insert an **ASN1_Recursive**. The appropriate type to be inserted (whatever would go in the any) must be used if using the **ASN1::DynAny** interface, and **insert_any** if using the **CORBA::DynAny** interface. Again, if the wrong type is to be inserted in the **any**, then the exception could be raised.

The Extraction operations

The inherited operations will work based on the **CORBA::TypeCode** and for all types (ASN.1 or otherwise).

There are different extraction operations per primitive ASN.1 type, even if they map to the same IDL type. In this way, the interface is more type safe (in the ASN.1 sense).

For example, in the case introduced above, either **get_ushort** or **get_asn1_unsigned16** would work.

Note that there is nothing to extract an **ASN1_Recursive:** you have to extract the appropriate type (whatever is in the any) if using the **ASN1::DynAny** interface, and **get_any** if using the **CORBA::DynAny** interface. If the wrong type is to be extracted from the **any**, then the TypeMismatch exception should be raised.

The Navigation operations

In addition to the methods inherited from **CORBA::DynAny**, there is an extra method to get the current component as an **ASN1::DynAny**, rather than as a **CORBA::DynAny** (that would require narrowing for some operations).

The Enumerated interface

Provides operations to access names/values as specified in ASN.1. If these operations are not going to be available (because the type does not correspond to an ASN.1 enumerated value), then narrowing to this interface should fail.

The NamedNumber interface

This is a special case, as it is a primitive type, but has a subtype specification. It provides the ability to read or write values by their ASN.1 names. If these operations are not going to be available, then narrowing to this interface should fail.

The SetSeq interface

Instead of providing two exact interfaces, just one is provided for both SET and SEQUENCE types.

In addition to the operations to navigate the components, getting names, and inserting/extracting sequences (inherited from the **CORBA::DynStruct** interface); equivalent operations are provided with the ASN.1 counterparts. Specifically, field names and insert/extract sequence would use ASN.1 type names, instead of the IDL

equivalents inherited from the **CORBA::DynStruct** interface. Another difference is that in the ASN.1 sequences, fields that have the OPTIONAL or DEFAULT clauses might be omitted. Additionally, there are methods to insert absent/present optional (and defaulted) and also to check for the presence and value of such fields.

The Choice interface

As the inherited interface was almost complete, this interface only specifies duplicates of some operations to provide ASN.1 names/types.

The SetSeqOf interface

The same as with SetSeq, only one interface is specified for both SET OF and SEQUENCE OF types. The only added operation is the one to get the ASN.1 item kind.

The DynAnyFactory

This factory is capable of creating **DynAny**s for both normal IDL types and for ASN.1 types. Also, the creation methods are compatible and consistent with those provided by the ORB interface for IDL types.

- **The IDL Factory methods**

  Create the **CORBA::DynAny** subclasses appropriate for the provided typecode. The returned objects might be narrowable to one of the **ASN1::DynAny** interfaces, if the IDL type was indeed coming from an ASN.1 type.

- **The ASN1 Factory methods**

  In the absence of ASN.1 typecodes, the ASN.1 module nickname and type name are used as the mechanism to identify the type being created. In this case, the returned object already exports the appropriate interface. In case the specified type does not match the factory operation being used, the InconsistentKind exception is raised.

4.4 The OSI Management Information Repository

An *OSI Management Information Repository* (OSI MIR) contains the description and structure of the information models used within the TMN Management model.

This specification does not provide any standard interface for an OSI MIR, therefore allowing implementations of CORBA/TMN systems to provide this functionality using any appropriate mechanism.

An OSI MIR provides two services:

- *Model description*: The translation from GDMO and ASN.1 to IDL leaves some information elements untranslated. For example, some constraints on ASN.1 types, or some inheritance relationships in GDMO class hierarchies, cannot be translated into IDL. An OSI MIR can fill in the missing information for applications that need it, by providing a full description of the original GDMO and ASN.1 syntax.

- *Translation description*: Some implementations, in particular those dealing with both OSI and CORBA domains, may need to know the correspondence between GDMO/ASN.1 and the IDL representations of a model. For example, an implementation might need to know that the IDL enumerated value selecting the **globalForm** choice of X.711 ASN.1 **AttributeId** syntax is named **globalFormChoice_1**. An OSI MIR can provide information on this mapping.

An OSI MIR allows managers and agents to dynamically access all information on a certain model, both as an original GDMO/ASN.1 model and as the equivalent IDL model. Among other things, this makes it feasible to dynamically process management requests without necessarily having any compile-time knowledge of which GDMO/ASN.1 documents and modules were processed and what mapping rules were applied for translation into IDL. With the assistance of an OSI MIR available at run time, an application may:

- Check the validity of values with the constraints applied to their syntax.
- Translate management requests from one domain (CORBA or OSI) to another.

This specification allows implementations of CORBA/TMN systems to choose any approach to building an OSI MIR for providing dynamic, run-time metadata support to their applications, including, but not limited to, the following:

- An OSI MIR may be provided as a stand-alone proprietary CORBA facility with its own exposed IDL interface.
- An OSI MIR may be provided as a repository, which is an extension of the CORBA Interface Repository.
- An OSI MIR may be an internal repository within a CORBA/TMN system, with no exposed interface and hence no CORBA visibility to applications. It may be reserved only for internal use by the CORBA/TMN system implementation.

4.5 SNMP Management Facilities Specification

4.5.1 Overview

The SNMP Management Facilities section addresses the bidirectional mapping of names, messages, and events in SNMP domain to names, operation invocation, and events in CORBA domain by providing a set of SNMP-specific CORBA object services and extensions.

This is done by extending the JIDM Facilities specification and by providing SNMP-specific extensions to the generic JIDM manager-agent framework. Those facilities support functionality that is specific to SNMP Management, as follows:

- The ability to name MIB entries according to SNMP Management principles.
- The ability to create and delete MIB entries (or table rows) according to SNMP Management principles.
- The ability to communicate traps and notifications.

To support the interoperability between CORBA and SNMP domains, we have to develop a set of service interfaces, called SNMP Management facilities as an extension of some of the CORBA Object services specification. In addition, we take advantage of the generic manager-agent framework provided by the JIDM facilities, to initialize and find the services defined for the SNMP Management facilities.

The main purpose of the SNMP management facilities is to extend the CORBA object services to support SNMP protocol-specific behavior of MIB entries/objects. For example, the SNMP MIB entries have to be named whenever they are created, whereas CORBA objects need not be named when they are created. So, we have to standardize the way objects are created and named.

The goal of the service interfaces is to provide a uniform way to find MIB entries, retrieve information from MIB entries, and handle events from CORBA and SNMP domains.

In SNMP domains the MIB entries are named based on a set of well-defined policies about the order and the types of the index variables associated with the MIB entries. The SNMP Naming service interface encapsulates SNMP-specific naming behavior. The SNMP-specific Lifecycle services interface extends the COSS generic factory to support the behavior of the MIB module specific factories generated during SMI->IDL translation. The SNMP specific Lifecycle would also take care of the naming of the MIB entries when an MIB entry is created.

There is some information loss during the mapping of SNMP MIB to IDL and we have to retrieve it through some mechanism. For example, OIDs of the table-entries/groups and variables are not mapped to IDL but they are needed at the gateway. So we need to extend the CORBA interface repository to provide OID information.

The SNMP Management Information Repository (also known as SMI repository service) [OPTIONAL Service] defines a set of interfaces that provide information about the SNMP specific IDL modules and interfaces in the InterfaceRepository (IR) in an SNMP specific way. For example, one can obtain information about SNMP specific IDL module names, SNMP specific IDL interfaces, and SNMP specific variables among all IDL interfaces in the IFR. The view one gets from the SMI repository is an SNMP specific view based on the information in the IFR. One can use the SMI repository to query about OID of a variable.

SNMP CORBA Facilities 5

Contents

This chapter contains the following sections.

5.1 Overview

This chapter addresses the bidirectional mapping of names, messages, and events in SNMP domain to names, operation invocation, and events in CORBA domain by providing a set of SNMP-specific CORBA object services and extensions.

This is done by extending the JIDM Facilities specification and by providing SNMP-specific extensions to the generic JIDM manager-agent framework. Those facilities support functionality that is specific to SNMP Management, as follows:

- the ability to name MIB entries according to SNMP Management principles.
- the ability to create and delete MIB entries (or table rows) according to SNMP Management principles.
- the ability to communicate traps and notifications.

To support the interoperability between CORBA and SNMP domains, we have to develop a set of service interfaces, called SNMP Management facilities as an extension of some of the CORBA Object services specification. In addition, we take advantage of the generic manager-agent framework provided by the JIDM facilities, to initialize and find the services defined for the SNMP Management facilities.

The main purpose of the SNMP management facilities is to extend the CORBA object services to support SNMP protocol-specific behavior of MIB entries/objects. For example, the SNMP MIB entries have to be named whenever they are created, whereas CORBA objects need not be named when they are created. So, we have to standardize the way objects are created and named.

The goal of the service interfaces is to provide a uniform way to find MIB entries, retrieve information from MIB entries, and handle events from CORBA and SNMP domains.

In SNMP domains the MIB entries are named based on a set of well-defined policies about the order and the types of the index variables associated with the MIB entries. The SNMP Naming service interface encapsulates SNMP-specific naming behavior. The SNMP-specific Lifecycle services interface extends the COSS generic factory to support the behavior of the MIB module specific factories generated during SMI->IDL translation. The SNMP specific Lifecycle would also take care of the naming of the MIB entries when an MIB entry is created.

There is some information loss during the mapping of SNMP MIB to IDL and we have to retrieve it through some mechanism. For example, OIDs of the table-entries/groups and variables are not mapped to IDL but they are needed at the gateway. So we need to extend the CORBA interface repository to provide OID information.

The SNMP Management Information Repository (also known as SMI repository service) [OPTIONAL Service] defines a set of interfaces that provide information about the SNMP specific IDL modules and interfaces in the InterfaceRepository (IR) in a SNMP specific way. For example, one can obtain information about SNMP specific IDL module names, SNMP specific IDL interfaces and SNMP specific variables among all IDL interfaces in the IFR. The view one gets from the SMI repository is an SNMP specific view based on the information in the IFR. One can use the SMI repository to query about OID of a variable.

5.2 The SNMPMgmt Module

The SNMPMgmt module comprises a collection of interfaces that together define a basic set of services for developing SNMP Management Applications based on CORBA. This module contains the following interfaces:

- The **ProxyAgent** interface
- The **SmiEntry** interface
- The **GenericFactory** interfaces
- The **NamingContext** interface
- The **SmiTableIterator** and **GetNextEntryIterator** interfaces

```
#ifndef _SNMPMGMT_IDL_
#define _SNMPMGMT_IDL_

#include <orb.idl>
#include <CosPropertyService.idl>
```

```
#include <ASN1Types.idl>
#include <JIDM.idl>

#pragma prefix “jidm.org”

module SNMPMgmt {
    const string ManagementDomainKeyId = “Internet Management”;
    const string ManagementDomainKeyKind = “XSM environment”;
    const string ProtocolVer = “Protocol Version”;
    const string TransportProtocol = “Transport Protocol”;
    const string DomainTitle = “Domain Title”;
    const string TransportAddress = “Transport Address”;
    const string TransportPort = “Transport Port”;
    const string CommunityName = “Community Name”;
    const string ContextEngineID = “Context EngineID”;
    const string ContextName = “Context Name”;

    // Redefinition of types
    typedef CORBA::ScopedName ScopedName;
    typedef CosLifeCycle::Criteria Criteria;
    typedef CosPropertyService::PropertyName VarName;
    typedef CosPropertyService::PropertyNames VarNameList;
    typedef CosPropertyService::Property NameValuePair;
    typedef CosPropertyService::Properties NVPairList;

    typedef ASN1_ObjectIdentifier EntryIndex;
    typedef sequence < EntryIndex > EntryIndexList;

    typedef string TAddress; // Transport address of an agent

    enum ProtocolVersion { snmpV1, snmpV2c, snmpV3 };

    // SNMP Protocol specific exceptions
    exception ProtocolError {
        ASN1_Integer error_status;
        ASN1_Integer error_index;
    };
    exception MultVarProtocolError {
        ASN1_Integer error_status;
        VarNameList  error_var_list;
        NVPairList   result_var_list;
    };

    // SMI information module specific exceptions.
    exception NoSuchSmiModule { };
    exception NoSuchSmiEntry { };
    exception NoSuchVariable { };

    // MIB entry specific exceptions
    exception NoSuchHost { };
    exception NoSuchObject { };
    exception EndOfMibView { };
    exception AlreadyExists { };
```

```
interface SmiEntry : CosLifeCycle::LifeCycleObject,
        CosPropertyService::PropertySet {
    // the value of entry_name is always “0” for the groups.
    readonly attribute ASN1_ObjectIdentifier entry_name;
};
typedef sequence < SmiEntry > SmiEntryList;

interface SmiTableIterator {
    boolean next_one_entry( out SmiEntry smi_entry );
    boolean next_n_entries (
        in unsigned long how_many,
        out SmiEntryList smi_entry_list
    );
    void destroy();
};

 interface GenericFactory : CosLifeCycle::GenericFactory {
    SmiEntry create_mib_entry (
      in ScopedName t_entry_type,
      in ASN1_ObjectIdentifier entry_index,
      in Criteria create_criteria
    ) raises ( NoSuchSmiEntry, AlreadyExists );

    SmiEntry create_mib_entry_with_auto_name (
      in ScopedName t_entry_type,
      in Criteria create_criteria
    ) raises ( NoSuchSmiEntry, AlreadyExists );
};

interface GetNextEntryIterator {
    // Get the next entry index according to lexical ordering rule
    // of SNMP OIDs -- follows SNMP get-next traversal rule
    boolean next_one_entry ( out EntryIndex entry_index );
    boolean next_n_entries (
        in unsigned long how_many,
        out EntryIndexList entry_index_list
    );
    void destroy();
};

// NamingContext extends CosNaming::NamingContext to provide
// navigating the SNMP name space in the lexicographic order
// and SNMP specific name and context resolution
interface NamingContext : CosNaming::NamingContext {
    string get_next_entry(
        in string entry_name
    ) raises ( InvalidName, NotFound, CannotProceed );

    GetNextEntryIterator get_next_entry_iterator(
        in string initial_entry_name
    ) raises ( InvalidName, NotFound );
};

interface NamingDirectory : NamingContext {
    NamingContext resolve_domain_context(
```

```
            in TAddress p_host_name
        ) raises ( NoSuchHost, CannotProceed, InvalidName,
            NotFound );

        NamingContext resolve_smi_module(
           in TAddress p_host_name,
            in string p_smi_module_name
        ) raises ( NoSuchHost, NoSuchSmiModule, InvalidName,
            NotFound );

        NamingContext resolve_smi_entry(
            in TAddress p_host_name,
            in ScopedName p_entry_type
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed,
            InvalidName, NotFound );

        SmiEntry resolve_mib_entry(
            in TAddress p_host_name,
            in ScopedName p_entry_type,
            in string p_entry_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed,
            InvalidName, NotFound );

        void list_smi_entries(
            in TAddress p_host_name,
            in ScopedName p_entry_type,
            in unsigned long how_many,
            out SmiEntryList out_list,
            out SmiTableIterator table_iterator
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed,
            InvalidName, NotFound );
};

// ProxyAgent

interface ProxyAgent : JIDM::ProxyAgent {

        readonly attribute TAddress host_name;

        ASN1_Any get_a_variable (
            in TAddress p_host_name,
            in ScopedName p_var_scoped_name,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchVariable, NoSuchObject,
            ProtocolError );

        NVPairList get_variables (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in VarNameList p_var_name_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
            MultVarProtocolError );

        void set_a_variable (
```

```
            in TAddress p_host_name,
            in ScopedName p_var_scoped_name,
            in EntryIndex p_var_index,
            in ASN1_Any p_var_new_value
        ) raises ( NoSuchHost, NoSuchVariable, NoSuchObject,
            ProtocolError );

        void set_variables (
            in TAddress p_host_name,
        in ScopedName p_entry_scoped_name,
        in NVPairList p_var_nvp_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
            MultVarProtocolError );

            void list_mib_entries(
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in long p_how_many,
            out EntryIndexList p_entry_index_list,
            out GetNextEntryIterator p_entry_name_list_itr
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
            ProtocolError );
                    boolean mib_entry_exists (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name
        ) raises ( NoSuchHost, NoSuchSmiEntry, ProtocolError );

        boolean is_mib_module_supported (
            in TAddress p_host_name,
            in string p_smi_module_name
        ) raises ( NoSuchHost, NoSuchSmiModule, ProtocolError );

};

        struct EntryVarBind {
        ScopedName entry_name; // IDL scoped name of the interface
            for table-entry
        string entry_index; // row index of an entry in the form of
            ObjectId string
        CosPropertyService::Properties nvp_list;
};
typedef sequence<EntryVarBind> EntryVarBindList;
typedef EntryVarBindList NotificationVariableList;
typedef EntryVarBindList InformVariableList;

struct NotificationInfo { // to be sent when using untyped event
            channel
        CosNaming::Name src_entry_name;
        ScopedName event_type;
        ASN1_GeneralizedTime event_time;
        any notification_info;
};
struct InformInfo { // to be sent when using untyped event channel
        CosNaming::Name src_obj_name;
```

```
        InformVariableList inform_info;
};

interface Notifications {
        void snmp_notification (
                in CosNaming::Name src_entry_name,
                in ScopedName event_type,
                in ASN1_GeneralizedTime event_time,
                in any notification_info
        );
        void snmp_inform (
                in CosNaming::Name src_entry_name,
                in InformVariableList inform_variables
        );

        void snmp_report (
                in CosNaming::Name src_entry_name,
                        in InformVariableList report_variables
        );
};

interface PullNotifications {
        boolean try_snmp_notification (
                out CosNaming::Name src_entry_name,
                out ScopedName event_type,
                out ASN1_GeneralizedTime event_time,
                out any notification_info
        );

        void pull_snmp_notification (
                out CosNaming::Name src_entry_name,
                out ScopedName event_type,
                out ASN1_GeneralizedTime event_time,
                out any notification_info
        );

        boolean try_snmp_inform (
                out CosNaming::Name src_entry_name,
                out InformVariableList inform_variables
        );

        void pull_snmp_inform (
                out CosNaming::Name src_entry_name,
                out InformVariableList inform_variables
        );

        boolean try_snmp_report (
                out CosNaming::Name src_entry_name,
                out InformVariableList report_variables
        );

        void pull_snmp_report (
                out CosNaming::Name src_entry_name,
                out InformVariableList report_variables
        );
```

```
    };

};

#endif /* _SNMPMGMT_IDL_ */
```

5.2.1 The SNMPMgmt::ProxyAgent Interface

CORBA manager objects that require access to managed objects that are members of some SNMP managed object domain must establish a connection with that domain.

As a result of establishing the connection, an **SNMPMgmt::ProxyAgent** object is created. **SNMP::ProxyAgent** objects export the **JIDM::ProxyAgent** interface and support additional operations that are specific to SNMP Management.

The **SNMPMgmt::ProxyAgent** provides a generic and version independent SNMP MIB based query interface, that encapsulates the SNMP protocol specific behavior.

```
// ProxyAgent

    interface ProxyAgent : JIDM::ProxyAgent {
        readonly attribute TAddress host_name;

        ASN1_Any get_a_variable (
            in TAddress p_host_name,
            in ScopedName p_var_scoped_name,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchVariable, NoSuchObject,
                ProtocolError );

        NVPairList get_variables (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in VarNameList p_var_name_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
                MultVarProtocolError );

        void set_a_variable (
            in TAddress p_host_name,
            in ScopedName p_var_scoped_name,
            in EntryIndex p_var_index,
            in ASN1_Any p_var_new_value
        ) raises ( NoSuchHost, NoSuchVariable, NoSuchObject,
                ProtocolError );

        void set_variables (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in NVPairList p_var_nvp_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
                MultVarProtocolError );
```

```
.       void set_variables (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in NVPairList p_var_nvp_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
                   MultVarProtocolError );

        void list_mib_entries(
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in long p_how_many,
            out EntryIndexList p_entry_index_list,
            out GetNextEntryIterator p_entry_name_list_itr
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
                   ProtocolError );

        boolean is_mib_entry_exist (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name
        ) raises ( NoSuchHost, NoSuchSmiEntry, ProtocolError );

        boolean is_mib_module_supported (
            in TAddress p_host_name,
            in string p_smi_module_name
        ) raises ( NoSuchHost, NoSuchSmiModule, ProtocolError );

};
```

Connections are established by means of invoking the **access_domain** operation exposed by a root **JIDM::ProxyAgentFinder** object as explained in Section 2.1.4, “The JIDM::ProxyAgentFinder Interface,” on page 2-11. The value associated with the “XSM environment” Key parameter passed to the **access_domain** operation is **Internet Management**. Note that the **access_domain** operation returns a reference to a **JIDM::ProxyAgent** interface. If the client wants to get visibility of the specific operations defined for the **SNMPMgmt::ProxyAgent** interface, this reference must be narrowed.

Table 5-1 presents the names and meaning for criteria that can be passed in the invocation to the **access_domain** operation when trying to access an SNMP managed domain. While the domain title criterion is mandatory, the rest of criteria components are optional.

Table 5-1 SNMPMgmt conventions for proxy agent finding criteria

criterion name	type of value	meaning
“domain title”	TAddress	Transport Address associated to the managed object domain for which access is requested. The wildcard address (“*”) is allowed.

criterion name	type of value	meaning
“transport protocol”	string	Transport protocol used to access the managed domain. Possible values are “UDP”, “IPX”, etc. If not present, “UDP” is used as default. The value of this criterion determines the format of the TAddress used for the “domain title” criterion.
“transport port”	any	Transport protocol dependent access point. If not present, (unsigned short)161 is used as default (corresponding to the default UDP port for SNMP).
“protocol version”	SNMPMgmt::ProtocolVersion	SNMP protocol version to be used to access the managed domain. If not specified, snmpV1 is used as default.
“controller object”	JIDM::ProxyAgentController	reference associated to a JIDM::ProxyAgentController object registered by the manager (OPTIONAL).
“community name”	string	Information to be used as the community name to be sent in the SNMP PDUs. Only valid for snmpV1 and snmpV2c. If not present, “public” is used (OPTIONAL).
“context engine id”	string	Only valid for snmpV3. Consult SNMP v3 documentation for more information (OPTIONAL).
“context name”	string	Only valid for snmpV3. Consult SNMP v3 documentation for more information (OPTIONAL).

The **TAddress** type used as domain title follows the **TAddress** Textual-Convention for transport service addresses defined in the SNMPv2-TC module. For the UDP domain (default case), the **TAddress** follows the 4+2 octets format. The format of the **TAddress** string for UDP domain is as follows: **<IP-Address>[:<Udp-port>]**. The **<IP-Address>** represent the stringified value of first 4 octets in DNS format or dotted number format. The **<Udp-port>** represents the integer value of last 2 bytes.

Semantics of the domain title and controller object parameters were specified in Section 2.1.4, “The JIDM::ProxyAgentFinder Interface,” on page 2-11. If the domain title specified is the wildcard (“*”), then a generic **SNMPMgmt::ProxyAgent** object is returned, that would be able to interact with multiple SNMP agents.

The criteria, in the case of SNMP Systems Management Reference model, may include additional parameters, namely:

- Transport and protocol specifications, carrying information on the type of transport protocol, the protocol access point (port), and the version of SNMP required to communicate with the managed domain.
- Protocol dependent security related criteria.

Since **SNMPMgmt::ProxyAgent** objects are **JIDM::ProxyAgent** objects, they provide the means by which CORBA manager objects are able to obtain references to:

- An initial **CosLifeCycle::FactoryFinder** object located at the OSI managed object domain.
- An initial **CosNaming::NamingContext** object located at the OSI managed object domain.

Invoking the **find_factories** operation exposed by the initial **CosLifeCycle::FactoryFinder** object, CORBA manager objects may find factories that enable creation of new table entries in the SNMP managed object domain.

Invoking the **resolve** operation exposed by the initial **CosNaming::NamingContext** object, CORBA manager objects may obtain CORBA object references to existing members of the SNMPI managed object domain.

Once a CORBA manager object obtains a CORBA object reference associated to an SNMP managed object, it can invoke operations exposed by the object. It will do so by means of using the standard ORB services defined in *CORBA The Common Object Request Broker: Architecture and Specification:*

- the Dynamic Invocation Interface (DII); or
- IDL stubs generated from definitions in OMG IDL of interfaces exported by the object (which might have been generated from SNMP definitions according to XoJIDM (see "[XoJIDM] Inter Domain Management: Specification Translation" mentioned in Appendix A).

5.2.1.1 Description of the ProxyAgent operations

The get_domain_factory_finder operation

The **get_domain_factory_finder** operation obtains a reference to the initial **CosLifeCycle::FactoryFinder** object located at the domain being accessed through an **SNMPMgmt::ProxyAgent** object. As already explained in Section 2.1.2, "The JIDM::ProxyAgent Interface," on page 2-4, CORBA manager objects can locate appropriate managed object factories by means of invoking the **find_factories** operation exposed by this initial **CosLifeCycle::FactoryFinder** object.

The space of keys established for SNMP Management environments is described in Table 5-2.

Table 5-2 SNMPMgmt conventions for factory finder keys

id field	kind field	meaning
fully scoped name of object interface	“object interface”	Find factories that create objects supporting the named interface.
fully scoped name of factory interface	“factory interface”	Find factories supporting the named factory interface.

CORBA Managers can create managed objects either using operations exposed by specific factories whose interfaces are derived from specific SMI modules or by using operations exposed by generic factories.

In respect to generic factories, one (or several) of the three following scenarios may be supported:

1. The standard **CosLifeCycle::GenericFactory** interface is used.
2. The **SNMPMgmt::GenericFactory** interface is used.
3. One of the standard factory interfaces defined in SYSMANfacilties (see “[SYSMANfacilities] Systems Management: Common Management Facilities, Volume I.” mentioned in Appendix A) is used.

In any case, the factory object would be responsible for checking if the new managed object can be contained in the designated domain.

With these considerations in mind, the alternatives for finding factories in SNMP Systems Management environments are more precisely described as follows:

- Only the name of the object interface is specified.

  Here, it is implicitly assumed that there is a specific factory interface associated with the managed object interface. CORBA managers know the name and operations associated with the factory in advance so that they can properly narrow and use the reference returned by the **find_factories** operation.

- Only the name of the object factory interface is specified.

  Here, references returned by the **find_factories** operation can be narrowed to the IDL interface whose name has been specified. The CORBA manager object who invoked the operation knows the signature and semantics of operations supported by the designated object factory interface.

In case objects are created through **CosLifeCycle::GenericFactory** objects, the **Key** value passed in the invocation to the **create_object** operation would be the name of the interface exported by the new MIB table entry. The **Criteria** value would be a sequence of <name, value> pairs, which would correspond to the rest of the arguments needed for creation of the SMI entry as specified in Table 5-3 (name of the SMI entry in string or name-value-pair list format, initial attribute list, etc).

Table 5-3 SNMPMgmt conventions for managed object creation criteria

criterion name	type of value	interpretation
“managed object interface”	CORBA::ScopedName	Name of interface exported by the new SMI entry.
“managed object name”	string	Naming parameter based on index part of objectID of a variable, given as a string. Concatenated index values in dotted number form. The information in “name” and “index variables” are interchangeable.
“index variables”	CosPropertyService::Properties	Naming parameter based on index variables, given as a sequence of name-value pairs. Alternative to “managed object name” criterion.
“initialization”	CosPropertyService::Properties	When this parameter is supplied, it contains a set of attribute identifiers and values to be assigned to the new SMI entry.

The get_domain_naming_context operation

The **get_domain_naming_context** operation obtains a reference to the initial **CosNaming::NamingContext** object located at the domain being accessed through an **SNMPMgmt::ProxyAgent** object. The returned **CosNaming::NamingContext** reference can be narrowed to the **SNMPMgmt::NamingContext** interface, or even to the **SNMPMgmt::NamingDirectory** interface for wildcard **SNMPMgmt::ProxyAgent** objects.

As already explained in Section 2.1.2, “The JIDM::ProxyAgent Interface,” on page 2-4, CORBA manager objects can obtain CORBA object references to members of a managed object domain as a result of invoking the **resolve** operation exposed by the initial **CosNaming::NamingContext** object located at the domain. The **resolve** operation may also be used to obtain reference to **CosNaming::NamingContext** objects subordinated to the initial **CosNaming::NamingContext** object.

Managed objects will be named according to the SNMP Naming Principles.

The type of **SNMPMgmt::ProxyAgent** created depends on the criteria used when access to the domain was solicited. In particular, if access to a “wildcard” ProxyAgent was granted, then the initial **CosNaming::NamingContext** object in the managed domain must support resolution using the **TAddress host_name** parameters. However, if the **ProxyAgent** was not generic, then the use of this parameter with a value other than the empty string (““) will raise the standard NO_PERMISSION exception.

The destroy operation

Any **SNMPMgmt::ProxyAgent** object exposes the **destroy** operation, which disposes of the object. Disposing an **SNMPMgmt::ProxyAgent** object means freeing resources used to maintain the associated connection.

Destruction of an **SNMPMgmt::ProxyAgent** object can take place either gracefully or non-gracefully, as described in Section 2.1.2, “The JIDM::ProxyAgent Interface,” on page 2-4. A reference to a **JIDM::ProxyAgentController** object may be passed at the manager side, as described in Section 2.1.3, “The JIDM::ProxyAgentController Interface,” on page 2-9.

SNMP operations

The SNMP protocol version independent **SNMPMgmt::ProxyAgent** provides a set of operations that are convenient to use in CORBA domain to browse SNMP MIB in variable-specific as well as table-oriented ways. Besides, it extends the **JIDM::ProxyAgent** interface to support SNMP-specific management operations.

All SNMP operations carry a **p_host_name** parameter of type **TAddress**. This parameter may only be used in “wildcard” **ProxyAgent**s, and will contain the transport specific address of the agent to be contacted for this particular management operation. In case of “non-wildcard” **ProxyAgent**s, the agent is specified at the time of creation of the **ProxyAgent** object (specified in the criteria passed to the **access_domain** call), and therefore this parameter must be the empty string (““); if a different value is used, then the standard NO_PERMISSION exception is raised.

Type definitions and Exceptions

The **VarNameList** and **NameValuePair** types redefine the **CosPropertyService::PropertyNames** and **CosPropertyService::Property** respectively such that type name reflects variable centric approach of SNMP.

The **EntryIndex** type represents the string for instance information of a conceptual row of a table. The **EntryIndex** string represents the stringfied (in dotted number form) version of sequence of oids that represents the instance information. EntryIndexList type represent a set of entry indexes.

The ProtocolError exception is raised to inform the client application about the SNMP protocol related errors. SNMP errors are indicated by the **ProtocolError.error_status** field. An application will map the SNMP related error to the corresponding CORBA Exception as shown in Table 5-4.

Table 5-4 Mapping of SNMP Errors to IDL Exceptions

SNMP ERROR	IDL Exception
noError	NO_EXCEPTION
tooBig	IMP_LIMIT
noSuchName	NO_IMPLEMENT
badValue	BAD_PARAM

SNMP ERROR	IDL Exception
readonly	BAD_OPERATION
genErr	INTERNAL
noAccess	NO_PERMISSION
wrongType	BAD_TYPECODE
wrongLength	MARSHAL
wrongEncoding	MARSHAL
wrongValue	BAD_PARAM
noCreation	CosLifeCycle::InvalidCriteria
inconsistentValue	BAD_PARAM
resourceUnavilable	NO_RESOURCE
commitFailed	INTERNAL
undoFailed	INTERNAL

The MultVarProtocolError exception is raised during multiple value get-set method. The **error_status** field follows the same mapping used for **ProtocolError**. The **result_var_list** field contains the returned variable list. The **error_var_list** field contains the names of the variables that are not part of the **result_var_list**.

The NoSuchSmiModule exception is raised if an SNMP agent does not support the associated MIB module. The NoSuchSmiEntry exception is raised if an SNMP agent does not support a specific MIB table or a group. The NoSuchVariable exception is raised if the SNMP agent does not support the specific SMI variable. The NoSuchHost exception is raised if the host cannot be found in the host database. The NoSuchObject exception is raised if the **noSuchObject** element is selected in the value part of the **VarBind** returned result. The NoSuchInstance exception is raised if the **noSuchInstance** element is selected in the value part of the **VarBind** returned result.

The get_a_variable operation

The **ProxyAgent::get_a_variable()** operation returns the value of an SNMP variable (tabular or non-tabular) given the name of the variable in IDL scoped format (**M::I::A**, where **M** is the SMI information module, **I** is the interface identifier of the table-entry/group for the variable, and **A** is the identifier of the attribute for the variable), and the index information in ASN.1 ObjectIdentifier format.

The **p_var_scoped_name** is the IDL scoped name of the variable in the form of **M::I::A**, where **M** is the SMI information module, **I** is the interface identifier of the table-entry/group for the variable, and **A** is the identifier of the attribute for the variable.

The **p_var_index** is the row index of the variable in the string form.

The returned value is of type **ASN1_Any** (typedef of **CORBA::Any**) and the **TypeCode** of the returned value is set according to the Specification Translation mapped IDL type of the attribute. In other words, the **TypeCode** of the returned value is equal to the value returned by the **type()** operation of the **AttributeDef for M::I::A**.

The operation raises NoSuchVariable exception if the variable name in **p_var_scoped_name** does not exist. The operation raises NoSuchObject exception if the variable with given instance information does not exist at the remote host. For all other cases the operation raises SNMP-specific protocol error using ProtocolError exception.

The get_variables operation

The **ProxyAgent::get_variables()** operation returns a list of values of an SNMP variable (tabular or non-tabular) given the IDL scoped name of the SMI table-entry/group (in the form of **M::I**, where **M** is the SMI information module, **I** is the interface identifier of the table-entry/group), IDL identifier of the variables, and the index information of a specific entry in ASN.1 ObjectIdentifier format.

The returned value is in the form of Name-Value pair list where names are the identifiers of the variables and values are of type **any**. The **TypeCode** of the values of the variables are set according to the mapped IDL type of the corresponding attributes.

The operation raises NoSuchSmiEntry exception if the interface name in **p_entry_scoped_name** does not exist in the interface repository. For all other cases the operation raises MultVarProtocolError exception by assigning the SNMP-specific protocol error to the **error_status** field.

The set_a_variable and set_variables operations

The **ProxyAgent::set_a_variable()** and **ProxyAgent::set_variables()** operations are defined to modify the values of variables within MIB entries. The parameters are similar to the corresponding **get** operations except that the new value is also provided as a parameter.

The list_mib_entries operation

The **ProxyAgent::list_mib_entries** interface provides access to the instance names of the entries of a certain table. This operation is designed to handle very large tables through the use of an iterator interface, called **GetNextEntryIterator** (see Section 5.2.8, “The SNMPMgmt::GetNextEntryIterator Interface,” on page 5-26).

The **GetNextEntryIterator** interface is defined to provide information about the indexes (names) of each row of a table. The **ProxyAgentt::list_mib_entries()** operation may be implemented using GET-BULK (GET-NEXT for SNMPv1). The **GetNextEntryIterator** interface provides the indexes of the entries of the MIB tables in lexicographical order.

The **p_entry_scoped_name** is the IDL scoped name of the variable in the form of **M::I**, where **M** is the SMI information module, and **I** is the interface identifier of the table-entry/group. **p_how_many** specifies the maximum number of entry indexes to be returned in **p_entry_index_list**.

If there are more entry indexes to be returned, then a reference to a **GetNextEntryIterator** object is returned; otherwise, a null reference is returned. The operation raises NoSuchSmiEntry exception if the interface name in **p_entry_scoped_name** does not exist. For all other cases the operation raises ProtocolError exception.

The mib_entry_exists operation

The **mib_entry_exists()** operation checks if there exists any entry for a specific (**p_entry_scoped_name**) group/table at the remote agent. It returns TRUE if there is at least one entry of **p_entry_scoped_name** table-entry/group and returns FALSE if there is none. The **p_entry_scoped_name** is the IDL scoped name of the table-entry/group in the form **M::I**.

The is_mib_module_supported operation

The **is_mib_module_supported()** operation returns TRUE if any group or table in the module specified by **p_smi_module_name** exists in the remote host. The **p_smi_module_name** provides the identifier of the SMI module in IDL.

5.2.2 The SNMPMgmt::SmiEntry interface

The **SNMPMgmt::SmiEntry** interface is the base IDL interface for all IDL interfaces of SMI groups and table-entries (possibly generated via the XoJIDM Specification Translation algorithm for SNMP to IDL translation)..

```
interface SmiEntry : CosLifeCycle::LifeCycleObject, CosPropertyService::PropertySet {
    // the value of entry_name is always “0” for the groups.
    readonly attribute ASN1_ObjectIdentifier entry_name;
};
typedef sequence < SmiEntry > SmiEntryList;
```

The **SmiEntry** interface inherits from **CosLifeCycle::LifeCycleObject** and **CosPropertyService::PropertySet** interfaces. The **SNMPMgmt::SmiEntry** interface has a read-only attribute called **entry_name** that contains the value of the index(es) (instance information part of the object-id of the group or table entry) of the corresponding variable or table entry. The **entry_name** attribute always contains “0” for SMI entries related to groups. The **entry_name** of a table-entry is known during the create time and set by the factory objects.

LifeCycle operations

The **SNMPMgmt::SmiEntry** interface inherits from the standard **CosLifeCycle::LifeCycleObject** interface. This means that every SMI entry exposes the operations defined in the **CosLifeCycle::LifeCycleObject** interface. Specifically, the following semantics for the operations is specified:

- The **copy** operation is not appropriate for SNMP management environments, so if invoked it should raise the NotCopyable exception.
- The **move** operation is not appropriate for SNMP management environments, so if invoked it should raise the NotMovable exception.
- The **remove** operation deletes the SmiEntry from the SNMP managed domain. Note that deletion of an SMI entry might be forbidden (as is the case for SNMP groups, and certain table entries). If, for whatever reason, the object could not be destroyed, the NotRemovable exception will be raised.

PropertySet operations

For SNMP management, each SNMP variable within a group or a table entry is always represented as an IDL attribute. In order to manipulate these attributes in groups, the CORBA Property Service is used.

The **CosPropertyService::PropertySet** interface is used to get the values of one or more variables in a group or in a row of a table with a single method invocation.

The property names to be used when invoking **PropertySet** operations are the simple unscoped names of the attributes that are to be accessed. The behavior of all operations is the same as specified in the Property Service Specification.

Specifically, it is possible to:

- Get the value of one, several, or all SNMP variables in an SNMP group or table entry by using the **get_property_value()**, **get_properties()**, and **get_all_properties()** operations, respectively.
- Set the value of one or several SNMP variables in an SNMP group or table entry by using the **define_property()** and **define_properties()** operations, respectively.

Note that this allows manipulation of multiple properties within a single SNMP group or table entry. It is not possible to access multiple table entries and/or groups with a single method invocation.

Since the properties based on SNMP variables are statically defined, the dynamic deletion of property is not allowed, so all the delete operations of the **PropertySet** interface (**delete_property()**, **delete_properties()**, and **delete_all_properties()**) should return the standard NO_PERMISSION exception.

5.2.3 The SNMPMgmt::SmiTableIterator Interface

For each SMI table within an SMI group, there is an operation to retrieve the information contained in the table. These operations return an **SNMPMgmt::SmiTableIterator** object reference, to allow the traversal of the information within the table.

```
interface SmiTableIterator {
    boolean next_one_entry( out SmiEntry smi_entry );
    boolean next_n_entries (
        in unsigned long how_many,
```

```
            out SmiEntryList smi_entry_list
        );
        void destroy();
};
```

The **SmiTableIterator** interface allows a client application to traverse a MIB table in the lexicographic order of the names (as defined in SNMP GET-NEXT) of its entries and returns the reference to each table entry. A reference to an **SmiTableIterator** is obtained as a result of the invocation of the **<M>::<G>::get_<T>()** operation, where **<M>::<G>** represents the IDL scoped name of a group and **<T>** represents the identifier of the table.

The **next_one_entry()** operation retrieves the object reference to the next entry of a specific table following the SNMP get-next traversal order. The returned reference is put in the **smi_entry** output parameter. If there are no more entries, then the operation returns FALSE; otherwise, it returns TRUE.

The **next_n_entries()** operation retrieves the references to a set of entries of a specific table following the SNMP get-next traversal order. The number of entries to be retrieved is specified by the **how_many** parameter and the returned entries are placed in the **smi_entry_list** output parameter. If there are no more entries, then the operation returns FALSE; otherwise, it returns TRUE.

The **destroy()** operation destroys the iterator object associated with the reference.

5.2.4 The SNMPMgmt::GenericFactory Interface

In SNMPv2, there is a data type (Textual Convention) called **RowStatus**, for creation of a row whose value defines various stages of life-cycle of an entry in a table. A table that supports entry creation by managers must include a variable of type **RowStatus**. When a manager wants to create a table entry, it must pass the proper value of **RowStatus** variable in addition to all the variables with read-create access and the index variables in the SNMPv2 SET message. (Please see the section 7.1.12.1. in RFC1902).

These operations are mapped to create operations in a factory interface in CORBA domain. A factory interface can be defined for each module and there will be one **create_<smi_entry_type>()** operation per IDL interface generated for group/table-entry from the SNMP MIB module.

```
interface GenericFactory : CosLifeCycle::GenericFactory {
    SmiEntry create_mib_entry (
        in ScopedName t_entry_type,
        in ASN1_ObjectIdentifier entry_index,
        in Criteria create_criteria
    ) raises ( NoSuchSmiEntry, AlreadyExists );

    SmiEntry create_mib_entry_with_auto_name (
        in ScopedName t_entry_type,
        in Criteria create_criteria
    ) raises ( NoSuchSmiEntry, AlreadyExists );
};
```

The **GenericFactory** is an extension of the **CosLifeCycle::GenericFactory** interface defined in the CORBA Life Cycle Service specification that provides SNMP SMI specific generic life-cycle operations.

The **create_mib_entry()** operation creates a MIB entry that supports the interface specified in the **t_entry_type** parameter and binds the name (given by the **entry_ins_name** parameter) with the reference to the object within the scope of the naming-context for the given interface type (**t_entry_type**). The operation returns the object reference of the newly created object. The **t_entry_type** parameter specifies the scoped name (in the form of **M::I**) of the IDL interface for an SMI based group or table-entry.

The **entry_ins_name** parameter specifies the instance information of the given MIB entry. The instance information is the stringified form of concatenated values of the index variables of the given entry. The **create_criteria** parameter specifies the values of a set of criterion in the form of a name-value-pair list. The allowed criteria are as defined in Table 5-5 and only the “initialization” criterion is used with this operation.

Table 5-5 Criteria for SNMP Specific Life Cycle Service

criterion name	type of value	meaning
“entry name”	string	Naming parameter based on index part of objectID of a variable, given as a string. Concatenated index values in dotted number form. Alternative to “index variables” criterion.
“index variables”	CosPropertyService::Properties	Naming parameter based on index variables, given as a sequence of name-value pairs. Alternative to “entry name” criterion.
“initialization”	CosPropertyService::Properties	When this parameter is supplied, it contains a set of attribute identifiers and values to be assigned to the new SMI entry.
“domain title”	SNMPMgmt::TAddress	Agent location where new entry is to be created (cannot be wildcard).

The **create_mib_entry()** operation raises the NoFactory exception if a type-specific factory cannot be found. The operation raises InvalidCriteria if the any of the criterion in **create_criteria** parameter is not a valid one. The operation raises the CannotMeetCriteria exception if any one of the criteria cannot be met. The operation raises the CosNaming::NamingContext::AlreadyBound exception if the given name in **entry_ins_name** is already bound within the scope of the naming-context for interface type in **t_entry_type**.

The **create_mib_entry_with_auto_name ()** is similar to **create_mib_entry()** but the name of the newly created object is assigned by the factory object.

5.2.5 The SNMPMgmt::NamingContext Interface

The main goal of mapping names in SNMP domain to names in CORBA domain is to standardize the SNMP naming hierarchy (host, variable, index) based on the **NamingContext** interfaces of the CORBA naming service. This goal is achieved in two ways: by standardizing the MIB tree hierarchy and by extending the **NamingContext** interface to list its entries in the lexicographic order of the names.

```
// NamingContext extends CosNaming::NamingContext to provide
// navigating the SNMP name space in the lexicographic order
// and SNMP specific name and context resolution

    interface NamingContext : CosNaming::NamingContext {
    string get_next_entry(
        in string entry_name
    ) raises ( InvalidName, NotFound, CannotProceed );

    GetNextEntryIterator get_next_entry_iterator(
        in string initial_entry_name
    ) raises ( InvalidName, NotFound );
};
```

The **SNMPMgmt::NamingContext** extends **CosNaming::NamingContext** to provide the navigation capability of the SNMP name space in the lexicographic order, as expected by the GET-NEXT command.

The **get_next_entry()** returns the name of the lexicographically next entry of that given by **entry_name**. If **entry_name** is a zero-length string, then the first entry is returned. If there are no more entries after the **entry_name**, then the CannotProceed exception is raised. The InvalidName and NotFound exceptions are raised based on the response from the **resolve()** operation using **entry_name**.

The **get_next_entry_iterator()** returns an Iterator (of type **SNMPMgmt::GetNextEntryIterator**) that conforms to the SNMP get-next lexicographic ordering. The returned iterator is set to point to the entry given by **initial_entry_name**. A zero-length string (““) for **initial_entry_name** will set the returned iterator at the beginning of list of entries in the naming context.

5.2.6 Naming MIB Entries Using SNMP Names in CORBA Domain

This section describes how to name entries of SNMP MIB in CORBA domain using the INDEX variables and access the MIB entries based on their corresponding SNMP names and finally retrieve the values of SNMP variables.

5.2.6.1 Overview of Naming of Variables in SNMP Domains

In SNMP domain the names are associated with the instances of variables; and the instances of tabular/non-tabular variables are uniquely addressable within the scope of a host IP address. The notion of table-entries are conceptualized by assigning the same index information to all the instances of variables of the same table entry. From a manager’s perspective, entries of the SNMP MIBs are implicitly arranged in the

following naming hierarchy: Host, variable OID, and row index. Since SNMP MIBs are agent location dependent the SNMP names are also location dependent. For non-tabular variables of group, the name is always zero (“0”) and for rows of tabular variable, the instance information depends on the INDEX clause of the corresponding table.

5.2.6.2 *Building of Global Name Tree of SNMP MIBs using CORBA Naming Service*

To map names of variable instances in SNMP domain to attributes of MIB entries CORBA domain, we need to map the names of variable within the scope of a host to hierarchical name tree based on CORBA Naming Service specification.

The hierarchy of the nodes of the name tree is as follows: SNMP-MIB-ROOT (global root of all SNMP MIBs), host name, MIB information module name, IDL interface names for table-entries/group, and finally the row indices (leaf nodes).

The nodes for root, host, module, and the table support the **SNMPMgmt::NamingContext** interface. The references to MIB entries are bound with the nodes for table-entries/groups using their names (row indices).

The SNMP names in string form are always mapped to the **id** part of a **CosNaming::NameComponent**. The **kind** part of a **NameComponent** is always initialized with a zero-length string.

The root of the global MIB tree is registered with a well-defined name, “SNMP-MIB-ROOT,” within the scope of some naming-context in the CORBA name space.

The nodes for the hosts (domains) of SNMP MIBs are represented by a naming-context (within the scope of the root MIB node, “SNMP-MIB-ROOT”). Since a host name can be represented in many ways (ip-address, DNS name, many aliases of host name), we need to define one name that can be used to unambiguously identify the nodes for the host under MIB. The default scheme for naming a node for host is DNS based name.

If the DNS based naming is not supported, then IP address (e.g., 135.180.160.16) can be used. Since the CORBA naming service allows name aliasing, a compliant implementation can support both DNS based naming and Ip-address based naming by registering the reference of nodes for a host naming-context using both names. Since the CORBA name-component has two parts: id and kind, the id of part is initialized by the host name and the kind part is initialized with zero-length string.

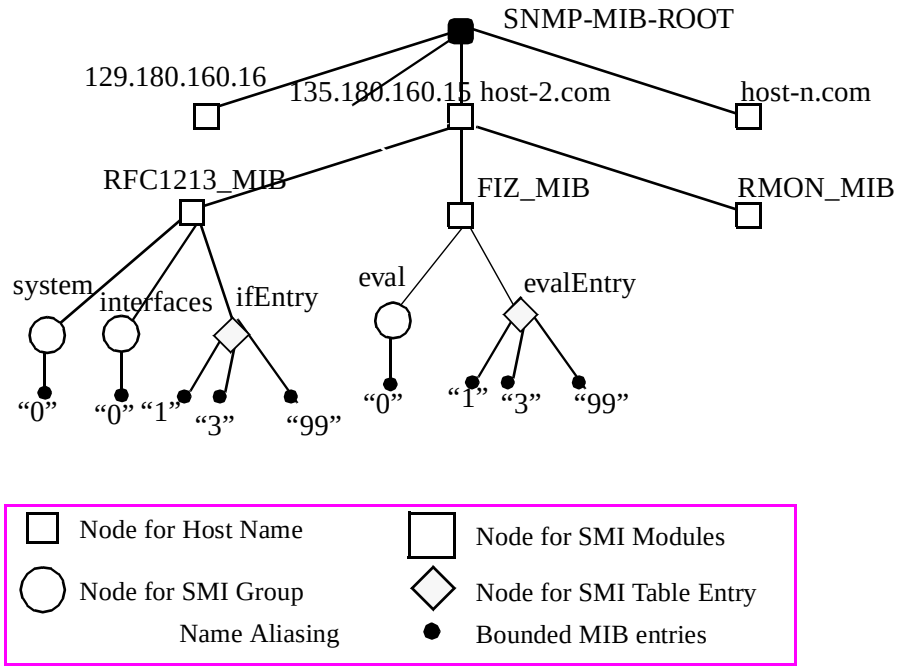

Figure 5-1 SNMP Naming hierarchy

A compliant implementation of the SNMP Name-Tree can add more nodes between the root ("SNMP-MIB-ROOT") node and hosts based on the DNS domain hierarchy or IP subnet as long as name resolution based on full DNS name or IP address returns the reference to the same naming-context node.

The **JIDM::ProxyAgent::get_domain_naming_context()** operation will return a reference to the naming-context at the host-name level for "non-wildcard" **ProxyAgent**s, while it will return a reference to the root **NamingContext** for "wildcard" **ProxyAgent**s.

Within the scope of each host (domain), there will be a naming-context for each of the MIB information modules that are implemented at the host. The naming context associated with the node for the MIB module is named using the IDL SMI module identifier.

Within the scope of each MIB information module, there will be a naming-context based node for each of the IDL interfaces for group, and table entry defined within the scope of the IDL module. The naming context associated with the nodes for groups/table-entries are named using the identifier of the corresponding IDL interface.

The entries (instances that support the IDL interfaces obtained by mapping SNMP MIB) of the MIB implementation are bound within the naming-context for the corresponding IDL interface. The names of the bounded MIB entries are the instance information needed to identify all the variables of the row of the conceptual table.

For non-tabular variables of group, the name is always zero(“0”) and for rows of tabular variable, the instance information depends on the INDEX clause of the corresponding table. The bounded objects are the leaf-object of the SNMP name tree. To support get-next traversal, the naming-contexts SNMP name is extended to retrieve the objects in the SNMP lexicography order.

5.2.6.3 Resolving SNMP names to obtain Object References to Table-entries/Groups and Support for SNMP GET-NEXT message

The following example describes how to map the SNMP name of an instance of non-tabular variable of a group to name of the corresponding MIB entry and IDL attribute in the CORBA domain. Given a non-tabular name in OID form, first we have to separate the instance information from the variable OID using the largest variable OID prefix match, as shown in the second line. Next, we derive the OID of the group by dropping the right-most identifier for the variable (as shown in third line).

evalSlot.0 (=> 1.3.6.1.3.555.2.1.0)
=> 1.3.6.1.3.555.2.1, 0
=> 1.3.6.1.3.555.2, 0, 1.3.6.1.3.555.2.1
=> FIZ_MIB::eval, 0, FIZ_MIB::eval::evalSlot
=> FIZ_MIB, eval, 0, FIZ_MIB::eval::evalSlot
=> <FIZ_MIB, eval, 0> , evalSlot
=> host_nc->resolve(<FIZ_MIB, eval, 0>)->get_property(“evalSlot”)

The OIDs of the group and the variable are then converted to corresponding IDL scoped names (possibly using the **SNMPMIR::Repository** interface). Then the IDL scoped name for the group is split into its module and interface name. Finally, we use the ordered sequence of module name, interface name, and instance information to derive a compound name of the MIB entry for the group within the scope of a naming context for a host. If we know the name of the host in DNS or IP address form, we know the complete path name of the group within the scope of the well defined root node, called MIB.

In the following example, we show how to access the value an instance of **evalSlot.0** (actually represented by `1.3.6.1.3.555.2.1.0`) to the IDL scoped name of the corresponding group interface (**FIZ_MIB::eval**) and attribute (**FIZ_MIB::eval::evalSlot**) of the interface.

evalStatus.2 (=> 1.3.6.1.3.555.2.2.1.4.2)
=> 1.3.6.1.3.555.2.2.1.4, 2
=> 1.3.6.1.3.555.2.2.1, 2, 1.3.6.1.3.555.2.2.1.4
=> FIZ_MIB::evalEntry, 2, FIZ_MIB::evalEntry::evalStatus
=> <FIZ_MIB, evalEntry, 2>, evalStatus
=> host_nc->resolve(<FIZ_MIB, evalEntry, 2>)->get_property(“evalStatus”)

The compound name of the object (within the scope of a host) that supports **eval** is derived by mapping the fully scoped name of the interface and the instance information (`“0”`) to the **id** part of name-components of the compound name.

Given the compound name, we can get the reference to the MIB entry that represents the **eval** group by using the **resolve()** operation of the host naming context. Then we can use the resolved object reference to retrieve the value of the variable by invoking the get operation associated with the attribute, **evalSlot**.

The example describes how to map the SNMP name of an instance of tabular variable of a conceptual table to naming service based names of an MIB entry and the attribute of the corresponding interface. The example splits the object-id (`1.3.6.1.3.555.2.2.1.4.2`) into the OID of the variable (`1.3.6.1.3.555.2.2.1.4`) and its instance information (2) by using the largest prefix match.

Then we derive the OID (`1.3.6.1.3.555.2.2.1`) of the corresponding table-entry by dropping the last number of the variable OID. Then we obtain the OIDs of table entry and the variable to corresponding IDL Scoped name of the interface (**FIZ_MIB::evalEntry**) and attribute (**FIZ_MIB::evalEntry::evalStatus**), respectively. Then we split the IDL scoped name of the table-entry into module name and interface name. Then we use the ordered sequence of the module name, interface name and the instance information to derive the compound name (**<FIZ_MIB, evalEntry, 2>**), within the scope of a host context. This compound name represent the name of the MIB entry representing the row in the table.

Given the compound name, we can get the reference to the MIB entry by using the **resolve()** operation of the host naming context. Then we use the resolved object reference to retrieve the value of the variable by invoking the **get** operation associated with the attribute, **evalStatus**.

5.2.7 The SNMPMgmt::NamingDirectory Interface

The **SNMPMgmt::NamingDirectory** interface extends **SNMPMgmt::NamingContext** to provide the global navigating capability of the SNMP name space.

```
interface NamingDirectory : NamingContext {
    NamingContext resolve_domain_context(
        in TAddress p_host_name
    ) raises ( NoSuchHost, CannotProceed, InvalidName, NotFound );

    NamingContext resolve_smi_module(
        in TAddress p_host_name,
        in string p_smi_module_name
    ) raises ( NoSuchHost, NoSuchSmiModule, InvalidName, NotFound );

    NamingContext resolve_smi_entry(
        in TAddress p_host_name,
        in ScopedName p_entry_type
    ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed, InvalidName,
            NotFound );

    SmiEntry resolve_mib_entry(
        in TAddress p_host_name,
        in ScopedName p_entry_type,
```

```
        in string p_entry_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed, InvalidName,
                NotFound );

        void list_smi_entries(
            in TAddress p_host_name,
            in ScopedName p_entry_type,
            in unsigned long how_many,
            out SmiEntryList out_list,
            out SmiTableIterator table_iterator
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed, InvalidName,
                NotFound );
};
```

The **resolve_domain_context()** returns the reference to the naming context for the domain specified by the **p_host_name**.

The **resolve_smi_module()** returns the reference to the naming context for **p_smi_module_name** within the scope of naming-context for the domain specified by **p_host_name**. The **p_host_name** follows the format defined in the **TAddress** type. The **p_smi_module_name** is the MIB module name.

The **resolve_smi_entry_context()** returns the reference to the naming context for **p_entry_type** (which has to be in scoped-name format) within the scope of the naming-context for **p_host_name**. **p_host_name** follows the format defined in the **TAddress** type. **p_entry_type** is in the form of **M::I** scoped name.

The **resolve_mib_entry()** operation returns the reference to the MIB entry for the row with index specified by **p_entry_name** with the scope of naming contexts for **p_host_name** and **p_entry_type**. **p_host_name** follows the format defined in the **TAddress** type. The **p_entry_type** parameter is in the form of **M::I** scoped name; **p_entry_index** is the instance part of the variables of a conceptual row of a table. For groups, **p_entry_index** is always “0.” The returned reference is a reference to the base SNMP interface, **SNMPMgmt::SmiEntry**, and may be narrow casted to the specific IDL interface for the table entry.

The **list_mib_entries()** operation returns the reference to a list of MIB entries of a table specified by **p_entry_name** with the scope of naming contexts for **p_host_name**. The **p_host_name** follows the format defined in the **TAddress** type. The **p_entry_type** parameter is in the form of **M::I** scoped name. The number of entries to be retrieved is specified by the **how_many** parameter and the returned entries are placed in the **ol**. If there are more entries in the table than specified by **how_many** parameter, then an **SNMPMgmt::SmiTableIterator** reference is placed in the **table_iterator** parameter.

5.2.8 The SNMPMgmt::GetNextEntryIterator Interface

The **GetNextEntryIterator** interface lets a client application traverse an MIB table in the lexicographic order followed by SNMP GET-NEXT message requirements and returns the index information of each entry. A reference to **GetNextEntryIterator** is

obtained as a result of the invocation of the **SNMPMgmt::ProxyAgent::list_mib_entries()** and **SNMPMgmt::NamingContext::get_next_entry_iterator()** operations.

```
interface GetNextEntryIterator {
// Get the next entry index according to lexical ordering rule
// of SNMP OIDs -- follows SNMP get-next traversal rule
    boolean next_one_entry ( out EntryIndex entry_index );
    boolean next_n_entries (
        in unsigned long how_many,
        out EntryIndexList entry_index_list
    );
    void destroy();
};
```

The **next_one_entry()** operation retrieves the instance information of the next entry of a specific table following the SNMP get-next traversal rule. If there are no more entries, then the operation returns FALSE; otherwise, it returns TRUE.

The **next_n_entries()** operation retrieves the instance information of a set of entries of a specific table following the SNMP get-next traversal rule. The number of entries to be retrieved is specified by the **how_many** parameter and the returned entries are placed in the **entry_index_list**. If there are no more entries, the operation returns FALSE; otherwise, it returns TRUE.

The **destroy()** operation destroys the object associated with the iterator reference.

5.2.9 Event Communication

5.2.9.1 Data Types for Untyped Event Communication

The following describes the data types for untyped event communications between MIB objects and the manager objects.

```
struct EntryVarBind {
    ScopedName entry_name; // IDL scoped name of the interface for table-entry
    string entry_index; // row index of an entry in the form of ObjectId string
    CosPropertyService::Properties nvp_list;
    };
    typedef sequence<EntryVarBind> EntryVarBindList;
    typedef EntryVarBindList NotificationVariableList;
    typedef EntryVarBindList InformVariableList;

    struct NotificationInfo { // to be sent when using untyped event channel
    CosNaming::Name src_entry_name;
    ScopedName event_type;
    ASN1_GeneralizedTime event_time;
    any notification_info;
    };
    struct InformInfo { // to be sent when using untyped event channel
    CosNaming::Name src_obj_name;
    InformV
```

The **SNMPMgmt::EntryVarBind** type is defined to map variables in **VarBindList** of SNMP PDU to IDL in a convenient form for objects in CORBA domain. The variables of each row of same table are grouped together by their index values and mapped to **EntryVarBind**. The name of the table-entry is mapped to **entry_name**, the common row index is mapped to **entry_index**. The textual names and the values of the variable in each **VarBind** are mapped to **CosPropertyService::PropertyType** based **nvp_list** field. Since the variables in **VarBindList** may span multiple rows of different tables, a single **VarBindList** may result in multiple **EntryVarBind** instances. The **EntryVarBindList** type is defined to handle such cases.

The **NotificationVariableList** and the **InformVariableList** types redefines the **EntryVarBindList** for SNMP message specific information.

The **NotificationInfo** type is defined as the data type to be sent as event data for untyped event communication. The **NotificationInfo** type has two parts: header and body. The header part consists of three elements: name of the object generating the event (**src_entry_name**), the type of the event (**event_type**) and the time of the event generation (**event_time**).

The body part is in the form any and the structure of the data depends on the type of the event being sent. The IDL type of the event body is generated based on the OBJECTS clause in the associated SNMP SMI based TRAP-TYPE or NOTIFICATION-TYPE macros. The actual event data based on the OBJECTS clause is to be mapped to data of type generated for typed event communication and then put into **notification_info** field as “any.”

The **InformInfo** type is defined to support the **InformRequest** and Report PDU based messages in SNMPv2. It consists of two parts: name of the object sending the inform-request or report message and the body of the inform or report data.

5.2.9.2 The SNMPMgmt::Notifications Interface

The **SNMPMgmt::Notifications** interface is defined to support push model of typed event communication described in the Typed Event Service specification.

```
interface Notifications {
    void snmp_notification (
        in CosNaming::Name src_entry_name,
        in ScopedName event_type,
        in ASN1_GeneralizedTime event_time,
        in any notification_info
    );
    void snmp_inform (
        in CosNaming::Name src_entry_name,
        in InformVariableList inform_variables
    );
    void snmp_report (
        in CosNaming::Name src_entry_name,
        in InformVariableList report_variables
    );
};
```

The **snmp_notification()** operation is invoked to send event data using a typed event channel. The parameters of the **snmp_notification()** operation are defined based on the fields of the **NotificationInfo** data type for untyped event communication.

The **snmp_inform()** operation is invoked to send inform-request messages using a typed event channel. The parameters of the **snmp_inform()** operation is defined based on the fields of the **InformInfo** data type.

The **snmp_report()** operation is invoked to send report messages using a typed event channel. The parameters of the **snmp_inform()** operation is defined based on the fields of the **InformInfo** data type.

5.2.9.3 The SNMPMgmt::PullNotifications Interface

The **SNMPMgmt::PullNotifications** interface is derived from the **SNMPMgmt::Notifications** interface based on the design pattern defined for pull-style typed interface in the typed event service specification. The interface name is defined as **Pull<I>**, where **<I>** is the name of the typed interface for the push model. For each operation (**<op>**) in **<I>** interface, a **try_<op>()** operation and a **pull_<op>()** operation are defined. The “in” parameters of the corresponding push operation in Notifications interface is converted to “out” parameter. The return value of **try_<op>** and **pull_<op>** are boolean and void, respectively.

```
interface PullNotifications {
    boolean try_snmp_notification (
        out CosNaming::Name src_entry_name,
        out ScopedName event_type,
        out ASN1_GeneralizedTime event_time,
        out any notification_info
    );

    void pull_snmp_notification (
        out CosNaming::Name src_entry_name,
        out ScopedName event_type,
        out ASN1_GeneralizedTime event_time,
        out any notification_info
    );

    boolean try_snmp_inform (
        out CosNaming::Name src_entry_name,
        out InformVariableList inform_variables
    );

    void pull_snmp_inform (
        out CosNaming::Name src_entry_name,
        out InformVariableList inform_variables
    );

    boolean try_snmp_report (
        out CosNaming::Name src_entry_name,
        out InformVariableList report_variables
    );
```

```
        void pull_snmp_report (
            out CosNaming::Name src_entry_name,
            out InformVariableList report_variables
        );
};
```

Note – All descriptions are exactly as described in Section 5.2.9.2, “The SNMPMgmt::Notifications Interface,” on page 5-28.

5.3 SNMP Management Information Repository

This section describes the SNMP Management Information Repository service, also known as the SNMP SMI repository service. This service is OPTIONAL-it is not required to provide the full functionality of the model.

During the specification translation of SMI based information module to CORBA IDL not all of the information is mapped to IDL. Some of the information got lost, for example OID information. Also the way SMI related meta information is available through CORBA Interface Repository (IFR) is not very convenient. For example, index variables of table entries are available as string in contrast to sequence of variables. Also the usage of the IDL interface repository interfaces are quite tedious. So we have defined SMI specific repository interface on top of CORBA IFR.

The SMI Repository service consists of two components: OID Repository and SMI macro repository. The OID repository provides the information about OID hierarchy and the textual names associated with each OID node in the OID hierarchy tree. The SMI macro repository provides meta-information about SMI modules and macros for groups, table-entry and variables.

```
#ifndef _SNMPMIR_IDL_
#define _SNMPMIR_IDL_

#include <orb.idl>
#include <ASN1Types.idl>

#pragma prefix “jidm.org”

module SNMPMIR {

    // SnmpvlGenericTrapId defines the identifiers for generic trap
    // types in SNMPv1.

    enum SnmpvlGenericTrapId {
        TRAP_COLDSTART, TRAP_WARMSTART, TRAP_LINKDOWN, TRAP_LINKUP,
        TRAP_AUTHFAIL, TRAP_EGPNEIGHBORLOSS,
        TRAP_ENTERPRISESPECIFIC
    };

    // GENERIC_TRAP_ENTERPRISE_OID defines the enterprise OID for
    // generic traps.
```

```
const ASN1_ObjectIdentifier GENERIC_TRAP_ENTERPRISE_OID =
        “1.3.6.1.4.1.3.1.1”;

// SmiAccessMode defines the enumerated values of the SMI based
// access - mode defined for a specific variables.

enum SmiAccessMode {
    read_only, read_write, read_create, write_only, inaccessible
};

. // Basic and Application specific SMI types.
enum SmiValueType {
    smi_null_value, smi_integer_value, smi_string_value, smi_objectID_value,
    smi_bit_value, smi_ipAddress_value, smi_counter_value, smi_gauge_value,
    smi_timeticks_value, smi_arbitary_value, smi_nsapAddress_value,
    smi_big_counter_value, smi_unsigned_integer_value, smi_unknown_type
};

typedef CORBA::ScopedName ScopedName;
typedef sequence < ScopedName > ScopedNameList;
typedef sequence < string > VarNameList;

typedef sequence < string > ModuleNameList;
typedef sequence < ASN1_ObjectIdentifier > OIDList;

interface OidRepository {

    ScopedName get_scoped_name ( in ASN1_ObjectIdentifier in_oid );

    string get_name ( in ASN1_ObjectIdentifier in_oid );
    ASN1_ObjectIdentifier get_oid ( in ScopedName in_scoped_name );
    ASN1_ObjectIdentifier get_var_oid (
        in ScopedName iface_scoped_name,
        in string var_name
    );

    string get_textual_obj_id ( in ASN1_ObjectIdentifier obj_id );

    void split_var_object_id (
        in ASN1_ObjectIdentifier  var_obj_id,
        out ASN1_ObjectIdentifier var_oid,
        out ASN1_ObjectIdentifier obj_index
    );

    ASN1_ObjectIdentifier get_next_oid ( in ASN1_ObjectIdentifier oid );

    ScopedName get_next_scoped_name ( in ScopedName scoped_name );
    ScopedName get_next_entry_type ( in ScopedName scoped_name );

};

    interface VariableDef : CORBA::AttributeDef {
    readonly attribute ASN1_ObjectIdentifier oid;
    readonly attribute SmiValueType smi_type;
    readonly attribute SmiAccessMode smi_access_mode;
```

```
        readonly attribute any default_value;
    };
    typedef sequence < VariableDef > VariableDefList;

    interface SmiEntryDef : CORBA::InterfaceDef {
        readonly attribute ASN1_ObjectIdentifier oid;
        readonly attribute unsigned long total_no_of_variables;
        readonly attribute VariableDefList var_def_list;
        readonly attribute VarNameList var_name_list;
        readonly attribute ScopedNameList var_scoped_name_list;
        readonly attribute OIDList var_oid_list;
        readonly attribute VarNameList index_var_names;

        readonly attribute ScopedName next_group_or_table;
        VariableDef lookup_variable( in string var_name );
    };
    typedef sequence < SmiEntryDef > SmiEntryDefList;

    interface GroupDef : SmiEntryDef {
        readonly attribute SmiEntryDefList table_entry_list;
    };
    typedef sequence < GroupDef >  GroupDefList;

    interface ModuleDef : CORBA::ModuleDef {
        readonly attribute GroupDefList smi_group_def_list;
        readonly attribute SmiEntryDefList smi_entry_def_list;
        readonly attribute CORBA::InterfaceDef  push_notification_def;
        readonly attribute CORBA::InterfaceDef  pull_notification_def;

        readonly attribute CORBA::InterfaceDef  default_value_def;
        SmiEntryDef lookup_smi_entry( in string smi_entry_name );
    };
    typedef sequence < ModuleDef >  ModuleDefList;

    interface Repository : CORBA::Repository, OidRepository {
        readonly attribute ModuleNameList module_name_list;
        readonly attribute ModuleDefList  module_def_list;
        boolean is_smi_module( in CORBA::Identifier module_name );
        ModuleDef lookup_smi_module( in string a_module_name );
        SmiEntryDef lookup_smi_entry( in ScopedName entry_scoped_name );
        ScopedNameList get_entry_var_list( in ScopedName entry_scoped_name );
        ScopedNameList get_entry_index_var_list( in ScopedName
            entry_scoped_name );
            any get_var_default_value( in ScopedName var_scoped_name );
        string get_generic_trap_desc( in ASN1_Integer trap_type );
    };

};

#endif /* _SNMPMIR_IDL_ */
```

The SMI Repository provides information in an SMI specific way. One can get the names of only those modules in IFR that are generated according to the Specification Translation rules, one can also get the names of SMI specific IDL interfaces. We can also get the default value of a variable (if defined) given its IDL scoped name. SMI

Macro Repository is built on top of the CORBA interface repository (IFR). Similar to CORBA IFR, the SMI Macro Repository follows the SMI containment and extends the corresponding IDL interfaces defined for CORBA IFR.

Table 5-6 Containment Hierarchy of the IDL interfaces in SMI Repository

SNMPMIR::Repository	**CORBA::Repository**
ModuleDef	ModuleDef
GroupDef	InterfaceDef
SmiEntryDef	InterfaceDef
VariableDef	AttributeDef

Table 5-6 describes the IDL interfaces defined to capture the meta information of SMI macros and their relationship with the interfaces defined in the CORBA IFR. The relationship between interfaces for the SMI repository and IFR are based on the SMI to IDL Specification Translation mapping rules.

The SMI related information about the variables are obtained using VariableDef interface. **VariableDef** extends the **CORBA::AttributeDef** interface because according to the Specification Translation we have mapped SMI variables as IDL attributes. The attributes, OID, **smi_type**, and **default_value**, of **VariableDef** interface can be used to obtain the SMI information. The name, IDL scoped name. access-mode and syntax information of the variable can be obtained from the **name**, **id**, **mode**, and **TypeCode** information of the **CORBA::AttributeDef** interface.

The SMI related information about the table entries are obtained using the **SmiEntryDef** interface. The **SmiEntryDef** interface extends the **CORBA::InterfaceDef** because during the specification translation SMI table entries are mapped as IDL interface. In addition to information provided by **CORBA::InterfaceDef**, **SmiEntryDef** interface provides a set of convenience operation optimized for SMI related operation. **var_name_list** and **var_scoped_name_list** attributes of **SmiEntryDef** interface can be used to get the names and IDL scoped names of the variables of a table/group in the lexicographic order of their OID. The **oid** attribute can be used to obtained the OID of a table-entry.

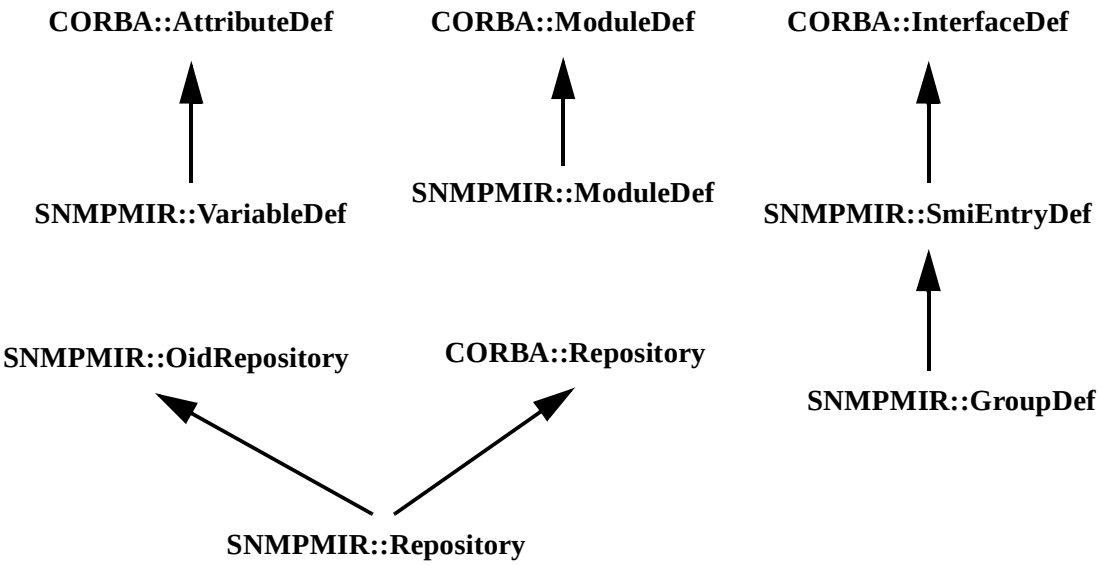

Figure 5-2 Interface Inheritance Hierarchy for the SMI Repository Service

The SMI related information about the groups are obtained using the **GroupDef** interface. The **GroupDef** interface extends the **SNMPMIR::SmiEntryDef** because during the specification translation SMI groups are mapped as if they are SMI table entry with single entry. In addition to information provided by **SNMPMIR::SmiEntryDef**, **GroupDef** interface provides a convenience operation to obtain the list of table-entries in the group. The inherited **SNMPMIR::SmiEntryDef::names** attribute can be used to obtain the names of the non-tabular variables of the group.

The SMI related information about the modules are obtained using the **ModuleDef** interface. The **ModuleDef** interface extends the **CORBA::ModuleDef** because during the specification translation SMI modules are mapped as IDL modules. In addition to information provided by **CORBA::ModuleDef**, **SNMPMIR::ModuleDef** interface provides a set of convenience operation optimized for SMI related operation. **smi_group_def_list** attribute of **SNMPMIR::ModuleDef** interface can be used to get the lists of references of the groups within the scope of the module. **smi_entry_def_list** attributes can be used get the references of both the **SmiEntryDef** and **GroupDef** interface within the scope of the module. **push_notif_interface** and **pull_notif_interface** attributes can be used to obtain the information about the Notification type macros.

Finally, SMI Macro Repository interface (**SNMPMIR::Repository**) is a specialization of the CORBA information repository (**CORBA::Repository**). The **SNMPMIR::Repository** also inherits the **OidRepository** interface so that an application need not keep track of references to two repositories. **SNMPMIR::Repository** interface is used to define a set of convenience operations to get the SMI related information from the repository.

The **SNMPMIR::Repository** is the container for all the SMI related modules that are registered with the CORBA IFR. According to the Specification Translation rules, the SMI related IDL module would not contain nested modules. So the name of a SMI based IDL module is sufficient to uniquely identify it in the repository.
SNMPMIR::Repository interface can be used to obtain the **SmiEntryDef** reference for a table-entry/group by providing the fully scoped name.

5.3.1 The SNMPMIR Module

The **SNMPMIR** module contains the IDL types and interfaces needed to build the SNMP SMI repository extension of the CORBA interface repository.

```
module SNMPMIR {
    // SnmpV1GenericTrapId defines the identifiers for generic trap types in SNMPv1.
    enum SnmpV1GenericTrapId {
        TRAP_COLDSTART, TRAP_WARMSTART, TRAP_LINKDOWN, TRAP_LINKUP,
        TRAP_AUTHFAIL,
        TRAP_EGPNEIGHBORLOSS, TRAP_ENTERPRISESPECIFIC
    };
    // GENERIC_TRAP_ENTERPRISE_OID defines the enterprise OID for generic traps.
    const ASN1_ObjectIdentifier GENERIC_TRAP_ENTERPRISE_OID =
                "1.3.6.1.4.1.3.1.1";

    enum SmiAccessMode {
        read_only, read_write, read_create, write_only, inaccessible };

    enum SmiValueType {
        smi_null_value, smi_integer_value, smi_string_value, smi_objectID_value,
        smi_bit_value, smi_ipAddress_value, smi_counter_value, smi_gauge_value,
        smi_timeticks_value, smi_arbitary_value, smi_nsapAddress_value,
        smi_big_counter_value, smi_unsigned_integer_value, smi_unknown_type
    };

    typedef string    FileName;
    typedef CORBA::ScopedName ScopedName;
    typedef sequence<ScopedName> ScopedNameList;
    typedef sequence<string> VarNameList;
    typedef sequence<string> ModuleNameList;
    typedef sequence<ASN1_ObjectIdentifier> OIDList;
    ....
};
```

The **SmiAccessMode** type defines the enumerated values of the SMI based access-mode defined for specific variables. The **SmiValueType** defines the enumerated values of the basic and application specific SMI types. The **FileName** type is used to specify the name of a file and it is defined for readability purposes. The **OIDList** type is a list of OID in dotted number form.

5.3.2 The OIDRepository Interface

The IDL files generated for each SMI modules do not contain any OID information (in dotted number form) of variables, table-entried groups and object identifiers. **OidRepository** interface (as shown in the example below) is defined to support operations for mapping OIDs in dotted number to textual names and textual names to OIDs. The **OidRepository** interface, can also be used to query information about the OID tree hierarchy.

```
module SNMPMIR {
....
    interface OidRepository {
    ScopedName get_scoped_name( in ASN1_ObjectIdentifier in_oid );
    string get_name( in ASN1_ObjectIdentifier in_oid );

    ASN1_ObjectIdentifier get_oid( in ScopedName in_scoped_name );
    ASN1_ObjectIdentifier get_var_oid(
                in ScopedName iface_scoped_name, in string var_name );

    string get_textual_obj_id( in ASN1_ObjectIdentifier obj_id );
    void split_var_object_id(
                in ASN1_ObjectIdentifier  var_obj_id,
                out ASN1_ObjectIdentifier var_oid, out ASN1_ObjectIdentifier
                obj_index
    );
    ASN1_ObjectIdentifier get_next_oid( in ASN1_ObjectIdentifier oid );

    ScopedName get_next_scoped_name( in ScopedName scoped_name );
    ScopedName get_next_entry_type( in ScopedName scoped_name );

    boolean read_oid_file( in FileName file_name );
    };
....
};
```

The OID Repository is initialized by reading in the OID files generated during specification translation of SNMP MIB information modules. During the initialization, SNMP module names are identified in the IFR and the corresponding OID files are read in. The OID files generated according to the Specification Translation mapping rules provide mapping between IDL scoped name and OID in dotted number form and SMI type information for variables. This interface also supports operations for loading an OID file for a specific MIB module. The **read_oid()** operation is defined to load OID information related to a new SNMP information module.

The **get_name()** and **get_scoped_name()** of **SNMPMIR::OidRepository** can be used to obtain the name and scoped-name respectively of a variable, group, table-entry or an **ObjectIdentifier** constants given its OID in dotted number form. The **SNMPMIR::OidRepository** can also be used to get the oid given a scoped name.

The **SNMPMIR::OidRepository** interface also provides operation to support SNMP GET-NEXT message. OID Repository can be used to get the next OID for a given OID using the **get_next_oid()** operation. **get_next_scoped_name()** returns IDL scoped name given a IDL scoped name.

Note – The OID Repository is only useful at the CORBA/SNMP gateway - in most MIB implementation and management applications it would not be needed.

The **get_scoped_name_by_oid()** operation returns the IDL scoped name of an item in Interface Repository (IFR) given the OID (in the dotted number form) of the item. For example, **oidRepoRef->get_scoped_name("1.3.6.1.2.1.1")** would return **"RFC1213_MIB::system"** and **oidRepoRef->get_scoped_name("1.3.6.1.2.1.2.2.1.3")** would return **"RFC1213_MIB::ifEntry:: ifType."** If the OID does not exist in the repository then the CORBA::OBJECT_NOT_EXIST exception is raised.

The **get_name()** operation returns the identifier of an item in IFR given the OID (in the dotted number form) of the item. For example, **oidRepoRef->get_name("1.3.6.1.2.1.1")** would return "system" and **oidRepoRef->get_name("1.3.6.1.2.1.2.2.1.3")** would return "ifType." If the OID does not exist in the repository then the CORBA::OBJECT_NOT_EXIST exception is raised.

The **get_oid()** operation returns the OID of an item in the IFR given the IDL scoped name of the corresponding item. For example, **oidRepoRef->get_oid("RFC1213_MIB::system")** would return "1.3.6.1.2.1.1" and **oidRepoRef->get_oid ("RFC1213_MIB::ifEntry::ifType")** would return "1.3.6.1.2.1.2.2.1.3". If the OID does not exist in the repository then the CORBA::OBJECT_NOT_EXIST exception is raised.

The **get_var_oid()** operation returns the OID of a variable given the scoped name of the table-entry/group and textual name of the variable. For example, **oidRepoRef->get_oid("RFC1213_MIB::system", "sysDescr"**) would return "1.3.6.1.2.1.1." and **oidRepoRef->get_oid("RFC1213_MIB::ifEntry**", "**ifType**") would return "1.3.6.1.2.1.2.2.1.3". If the **var_name** does not exist in the repository then CORBA::OBJECT_NOT_EXIST exception is raised.

The **get_textual_obj_id()** operation converts an **obj_id** in dotted number form into a scoped name using the largest prefix match and returns a string by concatenating the scoped name for the matched **obj_id** and the unmatched part of the **obj_id**. For example, **oidRepoRef->get_scoped_name("1.3.6.1.2.1.1.1.0")** would return "**RFC1213_MIB::system::sysDescr.0**" and **oidRepoRef->get_scoped_name("1.3.6.1.2.1.2.2.1.3.1"**) would return "**RFC1213_MIB::ifEntry::ifType.1**." If there is no such prefix match, then a copy of the input OID is returned.

The **split_var_object_id()** operation splits an ObjectId into OID form using longest prefix match into two parts: **var_oid** and index-values.

For example,

oidRepoRef->get_scoped_name("1.3.6.1.2.1.1.1.0")

would return "**RFC1213_MIB::system::sysDescr**" in **var_oid** and "**0**" in **obj_index**; and

oidRepoRef->get_scoped_name(“1.3.6.1.2.1.2.2.1.3.1”)

would return “**RFC1213_MIB::ifEntry::ifType**” in **var_oid** and “**1**” in **obj_index**. If there is no index information in the **var_obj_id** then a zero-length string is returned in **obj_index**. For example,

oidRepoRef->get_scoped_name(“1.3.6.1.2.1.1.1”)

would return “**RFC1213_MIB::system:: sysDescr”** in **var_oid** and “**in obj_index**.”

The **get_next_oid()** operation returns the next OID in the OID hierarchy given the input OID. Both input and returned OIDs are in dotted number form. The “next” based on the lexicographic ordering of the OID as per GET-NEXT message. If there is no such OID in the OID hierarchy then OBJECT_NOT_EXIST exception is raised. For example, **oidRepoRef->get_next_oid** (“1.3.6.1.2.1.1.1”) would return “**1.3.6.1.2.1.1.2**” and **oidRepoRef->get_next_oid** (“**1.3.6.1.2.1.2.1**”) would return “**1.3.6.1.2.1.2.2**.”

The **get_next_scoped_name()** operation is similar to **get_next_oid()** but the input and output parameters in the corresponding IDL scoped name form.

For example, **oidRepoRef-> get_next_scoped_name(“RFC1213_MIB::system::sysDescr”)** would return “**RFC1213_MIB::system::sysObjectID**” and **oidRepoRef->get_next_scoped_name(“RFC1213_MIB::interfaces:: ifNumber”)** would return **“RFC1213_MIB::ifTable**.”

oidRepoRef->get_next_scoped_name(s) is equivalent to **oidRepoRef->get_scoped_name(oidRepoRef->get_next_oid(oidRepoRef->get_oid(s)));**

The **get_next_entry_type()** operation returns the IDL scoped name of the next IDL interface for a group/table-entry. **get_next_entry_type()** will always return IDL scoped name for a SMI group/table-entry that correspond to an Specification Translation generated IDL interface name.

For example, **oidRepoRef->get_next_entry_type(“RFC1213_MIB::mib_2”**) would return **“RFC1213_MIB::system**”; **oidRepoRef->get_next_entry_type(“RFC1213_MIB::interfaces::ifNumber”**) would return **“RFC1213_MIB::ifEntry**” and **oidRepoRef->get_next_entry_type(“RFC1213_MIB::ifEntry::ifIndex”)** would return “**RFC1213_MIB::ip” (RFC1213_MIB::at is inaccessible)**. If there is no such next entry in the repository, then CORBA::OBJECT_NOT_EXIST exception is raised.

For each SNMP SMI module, an OID file is generated during specification translation. The name of the OID file for SMI information module is **<smi-module-name>.oid** where **<smi-module-name>** is the IDL identifier of the SMI module in ASN.1. **read_oid_file()** reads the **ScopedName/OID** mapping table from the given input file. The **file_name** could be name of the file, or complete path name of the file.

5.3.3 The VariableDef Interface

VariableDef interface is defined to retrieve SMI specific information of a SMI variable. **VariableDef** interface extends the **CORBA::AttributeDef** interface to provide information that is not available through **CORBA::AttributeDef**.

```
module SNMPMIR {
....
interface VariableDef : CORBA::AttributeDef {
      readonly attribute ASN1_ObjectIdentifier oid;
      readonly attribute SmiValueType smi_type;
      readonly attribute SmiAccessMode smi_access_mode;
      readonly attribute any default_value;
      };
typedef sequence<VariableDef> VariableDefList;
....
};
```

The attribute oid represents the ObjectIdentifier of the SMI variable. During specification translation, the OID of a variable with scoped name **M::I::V** is mapped in **M.oid** file and this value is obtained using the repository interface.

The **smi_type** attribute represent the basic and application specific SMI type defined for the variable. During specification translation, **smi_type** of a variable with scoped name **M::I::V** is mapped in **M.oid** file.

The **smi_access_mode** attribute represents the value of the MAX-ACCESS clause in the OBJECT-TYPE macro for the variable. According to the Specification Translation rules, the, **smi_access_mode** of a variable with scoped name **M::I::V** is mapped in **M.oid** file.

During specification translation, DEFVAL clause (if present) is mapped as **<M>::DefaultValues::<V>()** operation for a variable with **<M>::I::<V>** scoped name. The **default_value** attribute can be obtained by invoking the **<M>::DefaultValues::<V>()** operation where the scoped name of the variable is **<M>::I::<V>**. If the default value is not defined, then CORBA::OBJECT_NOT_EXIST exception is returned.

VariableDefList type represent a list of **VariableDef** interfaces.

5.3.4 The SmiEntryDef Interface

The **SmiEntryDef** interface represents SMI specific information associated with an IDL interface generated from an SMI module. **SmiEntryDef** interface extends **CORBA::InterfaceDef** and provides in information in SMI centric way.

```
module SNMPMIR {
....
interface SmiEntryDef : CORBA::InterfaceDef {
      readonly attribute ASN1_ObjectIdentifier oid;

      readonly attribute unsigned long total_no_of_variables;
```

```
        readonly attribute VariableDefList var_def_list;

        readonly attribute VarNameList var_name_list;

        readonly attribute ScopedNameList var_scoped_name_list;
        readonly attribute OIDList var_oid_list;

        readonly attribute VarNameList index_var_names;

        readonly attribute ScopedName next_group_or_table;


        VariableDef lookup_variable( in string var_name );
        };
typedef sequence<SmiEntryDef> SmiEntryDefList;
....
};
```

The **oid** attribute represents the OID of this interface. According to the Specification Translation rules, the, OID of a table-entry/group with scoped name **M::I** is mapped in the **M.oid** file.

The **total_no_of_variables** attribute represents the total number of SMI based variables in this interface.

The **var_def_list** attribute maintains the list of the **VariableDef** interfaces of the variables of this table-entry/group. It can be derived from the list of attributes of the interface. The list is ordered according to the lexicographic order of the OIDs of the variables.

The **var_name_list** attribute maintains the list of the names of the variables of this table-entry/group. It can be derived from the list of attributes of the interface. The list is ordered according to the lexicographic order of the OIDs of the variables.

The **var_scoped_name_list** attribute maintains the list of the IDL scoped names of the variables of this interface. The IDL scoped name of the variable **V** is **M::I::V** where **M::I** is the scoped name of the interface for table-entry/group. The list is ordered according to the lexicographic order of the OIDs of the variables.

The **var_oid_list** attribute maintains the list of the OID of the variables of this interface. The list is ordered according to the lexicographic order of the OIDs.

The **index_var_names** attributes maintains the index variables of this interface. According to the Specification Translation rules, the INDEX clause of a table-entry is mapped as IDL string constant, called **IndexVarList**, within the scope of the IDL interface for the table-entry. Index var names are obtained by converting the value of **M::I::IndexVarList** of a table entry with scoped name **M::I**, into a sequence of strings (of individual index variables). If there is no **IndexVarList** (e.g., for IDL interface for groups) a zero-length sequence is returned.

The **next_group_or_table** attribute represents the information about the next IDL interface for a group/table-entry according to the lexicographic OID order.

The **lookup_variable()** operation is a convenience operation that return reference to **VariableDef** of a variable of this interface.

The **SmiEntryDefList** type represent a list of **SmiEntryDef** interfaces.

5.3.5 The SmiGroupDef Interface

A group in SNMP SMI is a collection of tables and non-tabular variables. **GroupDef** interface represents those IDL **SmiEntry** interfaces that are generated from SMI based groups. The **SmiGroupDef** interface extends **SmiEntryDef** and acts as a collection entity of table-entry interfaces. The list of non-tabular variables can be obtained from the inherited **var_def_list** attribute.

```
module SNMPMIR {
....
interface GroupDef : SmiEntryDef {
    readonly attribute SmiEntryDefList table_entry_list;
    };
    typedef sequence<GroupDef>  GroupDefList;
....
};
```

The **table_entry_list** attribute maintains the list of the **SmiEntryDef** for the table-entries of this group.

5.3.6 The SmiModuleDef Interface

The **SnmpModuleDef** extends the **CORBA::ModuleDef** interface and provides SNMP SMI specific attributes and functions. An **SNMPModuleDef** contains a list of SNMP interfaces and an interface object for push and pull notification interfaces.

```
module SNMPMIR {
....
interface ModuleDef : CORBA::ModuleDef {
    readonly attribute GroupDefList smi_group_def_list;

    readonly attribute SmiEntryDefList smi_entry_def_list;
    readonly attribute CORBA::InterfaceDef  push_notification_def;

    readonly attribute CORBA::InterfaceDef  pull_notification_def;

    readonly attribute CORBA::InterfaceDef  default_value_def;
    SmiEntryDef lookup_smi_entry( in string smi_entry_name );
    };
    typedef sequence<ModuleDef>  ModuleDefList;
....
};
```

The **smi_group_def_list** attribute maintains the list of SMI specific groups in a SMI information module.

The **entry_interface_list** attribute maintains the list of interfaces which are a subtype of **SNMPMgmt::SmiEntry** defined within the scope of this module. This list includes **SmiEntryDef** interfaces for all the groups and table-entries of this module.

According to the Specification Translation rules, a push notification interface is defined with the IDL scoped name **M::Notifications** for SMI MIB module with name **M**. The **push_notification_def** attribute maintains the reference to **InterfaceDef** for **M::Notifications** interface. If no such interface is defined then a nil object-reference is returned.

According to the Specification Translation rules, a pull notification interface is defined with IDL scoped name **M::PullNotifications** for **SMI MIB** module with name **M**. The **pull_notification_def** attribute maintains the reference to **InterfaceDef** for **M::PullNotifications** interface. If no such interface is defined, then a nil object-reference is returned.

According to the Specification Translation rules, a Default value interface is defined with IDL scoped name **M::DefaultValues** for **SMI MIB** module with name **M**. The **default_value_def** attribute maintains the reference to **InterfaceDef** for **M::DefaultValues** interface. If no such interface is defined, then a nil object-reference is returned.

The **lookup_smi_entry()** operation returns the **SmiEntryDef** for the specified group/table-entry interface within the scope this SMI module.

The **ModuleDefList** type represents a list of **ModuleDef** interfaces.

5.3.7 The Repository Interface

The **SNMPMIR::Repository** interface extends the **CORBA::Repository** interface and it provides SNMP SMI specific specialization of CORBA Interface Repository (IFR) interface. The **SNMPMIR::Repository** interface inherits the **CORBA::Repository** and the **SNMPMIR::OidRepository** interfaces.

```
module SNMPMIR {
....
    interface Repository : CORBA::Repository, OidRepository {
        readonly attribute ModuleNameList module_name_list;
        readonly attribute ModuleDefList  module_def_list;
        boolean is_smi_module( in CORBA::Identifier module_name );
        ModuleDef lookup_smi_module( in string a_module_name );

        SmiEntryDef lookup_smi_entry( in ScopedName entry_scoped_name );
        ScopedNameList get_entry_var_list( in ScopedName entry_scoped_name );
        ScopedNameList get_entry_index_var_list( in ScopedName
                entry_scoped_name );
        any get_var_default_value( in ScopedName var_scoped_name );

        string get_generic_trap_desc( in ASN1_Integer trap_type );
    };
....
};
```

The **module_name_list** attribute maintains the list SNMP SMI specific IDL module names in the IFR.

The **module_def_list** attribute maintains the list **SNMPMIR::ModuleDef** interfaces for the SNMP SMI specific IDL module names in the IFR.

The **lookup_smi_entry()** operation returns the interface def of a specific SNMP table-entry/group. **entry_scoped_name** is specified as **M::I**.

References A

A.1 List of References

[Telefonica I+D]
Common Facilities for Systems Management

Juan J. Hierro, Jesús A. Gonzalez, José M. Lorenzo
Telefónica Investigación y Desarrollo.
June 1997

[Lucent]
Mapping of Common Management Information Service to CORBA Object Services

Subrata Mazumdar
Lucent Technologies
February 1996

[Alcatel]
Alternative proposal to map OSI naming in CORBA

Olivier Potonniée
Alcatel Corporate Research Centre
July 1997

[XSMG]
Systems Management: Reference Model.

X/Open guide number G207.
August 1993.

[OMAG]
Object Management Architecture Guide.

Richard Mark Soley (ed.), OMG document number 92.11.1.
Revision 2.0, September 1992.

[CORBA]
The Common Object Request Broker: Architecture & Specification.

Revision 2.0, July 1995.

[CORBAservices]
CORBAservices: Common Object Services Specification.
(note: see individual CORBA Service documents)

[SYSMANfacilities]
Systems Management: Common Management Facilities, Volume I.

X/Open Preliminary Specification number P421.
June 1995.

[X700]
OSI Management Framework.

ITU-T (CCITT) Recommendation X.701, ISO/IEC 7498-4.
September 1992.

[X701]
Systems Management Overview.

ITU-T (CCITT) Recommendation X.701, ISO/IEC 10040.
January 1992.

[X710]
Common Management Information Service Definition.

ITU-T (CCITT) Recommendation X.710, ISO/IEC 9595.
March 1991.

[X720]
Management Information Model.

ITU-T (CCITT) Recommendation X.720, ISO/IEC 10165-1.
January 1992.

[X721]
Definition of Management Information.

ITU-T (CCITT) Recommendation X.721, ISO/IEC 10165-2.
February 1992.

[X722]
Guidelines for the definition of Managed Objects.

ITU-T (CCITT) Recommendation X.722, ISO/IEC 10165-4.
January 1992.

[X734]
Event Management Function.

ITU-T (CCITT) Recommendation X.734, ISO/IEC 10164-5.
September 1992.

[X735]
Log Control Function.

ITU-T (CCITT) Recommendation X.735, ISO/IEC 10164-6.
September 1992.

[ASN1]
Abstract Syntax Notation One (ASN.1).

ITU-T (CCITT) Recommendation X.208, ISO/IEC 8824-1.
April 1993.

[M3010]
Principles for a Telecommunications Management Network.

ITU-T (CCITT) Recommendation M.3010.
June 1992.

[OMNI]
OMNIPoint Architecture Integration.

Network Management Forum (NMF) document.
October 1994.

[XoJIDM]
Inter Domain Management: Specification Translation

Joint X/Open-NMF Inter-Domain Management (XoJIDM) Task Force.
X/Open Preliminary Specification
May 1997.

Complete OMG IDL B

B.1 Normative IDL

IDL listed in this section is specified in this document. These files can also be found in electronic format, as a ZIP archive file, from the OMG document server, with document number telecom/98-10-11.

B.1.1 JIDM.idl

```
// File: JIDM.idl
#ifndef _JIDM_IDL_
#define _JIDM_IDL_

#include <CosNaming.idl>
#include <CosLifeCycle.idl>
#include <CosEventChannelAdmin.idl>

#pragma prefix “jidm.org”

module JIDM
{
    typedef CosNaming::Name Key;
    typedef CosLifeCycle::Criteria Criteria;

    exception InvalidKey {};
    exception InvalidCriteria {};
    exception CannotMeetCriteria { Criteria reason; };
    exception CannotAccess {};
    exception AlreadyExists {};
    exception NoEventPort {};

    interface ProxyAgent {
        enum DestructionMode {gracefully, non_gracefully};
        readonly attribute Criteria access_criteria;
```

```
        CosLifeCycle::FactoryFinder get_domain_factory_finder ();
        CosNaming::NamingContext get_domain_naming_context ();

        Criteria destroy (in DestructionMode mode, in Criteria the_criteria)
            raises (InvalidCriteria, CannotMeetCriteria);
    };

    interface ProxyAgentController {
        Criteria destruction_is_allowed (in Criteria the_criteria)
            raises (InvalidCriteria,CannotMeetCriteria);
        void destroyed (in Criteria the_criteria);
    };


    interface ProxyAgentFinder {
        ProxyAgent access_domain (in Key k, in Criteria the_criteria)
            raises (InvalidKey, CannotAccess, InvalidCriteria, CannotMeetCriteria);
    };

    interface DomainPort {
        readonly attribute Criteria associated_criteria;
        void destroy ();
    };

    interface DomainPortFactory {
        DomainPort create_domain_port (in Key k, in Criteria creation_criteria)
            raises (InvalidKey, InvalidCriteria, CannotMeetCriteria);
    };

    interface EventPort {
        readonly attribute CosEventChannelAdmin::SupplierAdmin supplier_admin;
        readonly attribute Criteria associated_criteria;
        void destroy ();
    };

    interface EventPortFactory {
        EventPort
            create_event_port (in Key k, in Criteria creation_criteria,
                in CosEventChannelAdmin::SupplierAdmin the_supplier_admin)
                raises (InvalidKey, InvalidCriteria, CannotMeetCriteria,
                        AlreadyExists);
    };

    interface EventPortFinder {
        CosEventChannelAdmin::SupplierAdmin
            find_event_port (in Key k, in Criteria the_criteria)
            raises (InvalidKey, InvalidCriteria, CannotMeetCriteria, NoEventPort);

    };
};

#endif /* _JIDM_IDL_ */
```

B.1.2 OSIMgmt.idl

```
// File: OSIMgmt.idl
#ifndef _OSIMGMT_IDL_
#define _OSIMGMT_IDL_

#include <orb.idl>
#include <JIDM.idl>
#include “X501Inf.idl”
#include “X711CMI.idl”

#pragma prefix “jidm.org”

// Macros used in the `raises’ clauses

#define ROSE_ERRORS\
      OSIMgmt::ROSEDuplicateInvocation,\
      OSIMgmt::ROSEMistypedArgument,\
      OSIMgmt::ROSEResourceLimitation, \
      OSIMgmt::ROSEUnrecognizedOperation

#define CREATE_ERRORS\
      ROSE_ERRORS, \
      OSIMgmt::AccessDenied,\
      OSIMgmt::ClassInstanceConflict,\
      OSIMgmt::DuplicateManagedObjectInstance,\
      OSIMgmt::InvalidAttributeValue,\
      OSIMgmt::InvalidObjectInstance,\
      OSIMgmt::MissingAttributeValue,\
      OSIMgmt::NoSuchAttribute,\
      OSIMgmt::NoSuchObjectClass,\
      OSIMgmt::NoSuchObjectInstance,\
      OSIMgmt::NoSuchReferenceObject,\
      OSIMgmt::ProcessingFailure,\
      OSIMgmt::ProcessingFailureEmpty

#define COMMON_ERRORS \
      ROSE_ERRORS, \
      OSIMgmt::AccessDenied, \
      OSIMgmt::ClassInstanceConflict, \
      OSIMgmt::ComplexityLimitation, \
      OSIMgmt::ComplexityLimitationEmpty, \
      OSIMgmt::InvalidScope, \
      OSIMgmt::InvalidFilter, \
      OSIMgmt::NoSuchObjectClass, \
      OSIMgmt::NoSuchObjectInstance, \
      OSIMgmt::ProcessingFailure, \
      OSIMgmt::ProcessingFailureEmpty, \
      OSIMgmt::SyncNotSupported

#define GET_ERRORS \
      COMMON_ERRORS, \
      OSIMgmt::GetListError, \
      OSIMgmt::OperationCancelled
```

```
#define SET_ERRORS \
      COMMON_ERRORS, \
      OSIMgmt::SetListError

#define ATTRIBUTE_ERRORS \
      COMMON_ERRORS, \
      OSIMgmt::GetListError, \
      OSIMgmt::SetListError

#define ACTION_ERRORS \
      COMMON_ERRORS, \
      OSIMgmt::InvalidArgumentValue, \
      OSIMgmt::NoSuchAction, \
      OSIMgmt::NoSuchArgument

#define DELETE_ERRORS \
       COMMON_ERRORS


module OSIMgmt
{
      // Definitions of ROSE and CMIS exceptions
      exception ROSEDuplicateInvocation { };
      exception ROSEMistypedArgument { };
      exception ROSEResourceLimitation { };
      exception ROSEUnrecognizedOperation { };
      exception AccessDenied { };
      exception ClassInstanceConflict
            { X711CMI::BaseManagedObjectIdType error_info; };
      exception ComplexityLimitation
            { X711CMI::ComplexityLimitationType error_info; };
      exception ComplexityLimitationEmpty { };
      exception DuplicateManagedObjectInstance
            { X711CMI::ObjectInstanceType error_info; };
      exception GetListError
            { X711CMI::GetListErrorType error_info; };
      exception InvalidArgumentValue
            { X711CMI::InvalidArgumentValueType error_info; };
      exception InvalidAttributeValue
            { X711CMI::AttributeType error_info; };
      exception InvalidFilter
            { X711CMI::CMISFilterType error_info; };
      exception InvalidScope
            { X711CMI::ScopeType error_info; };
      exception InvalidObjectInstance
            { X711CMI::ObjectInstanceType error_info; };
      exception MissingAttributeValue
            { X711CMI::MissingAttributeValueType error_info; };
      exception MistypedOperation { };
      exception NoSuchAction
            { X711CMI::NoSuchActionType error_info; };
      exception NoSuchArgument
            { X711CMI::NoSuchArgumentType error_info; };
      exception NoSuchAttribute
            { X711CMI::AttributeIdType error_info; };
```

```
exception NoSuchObjectClass
      { X711CMI::ObjectClassType error_info; };
exception NoSuchObjectInstance
      { X711CMI::ObjectInstanceType error_info; };
exception NoSuchReferenceObject
      { X711CMI::ObjectInstanceType error_info; };
exception OperationCancelled { };
exception ProcessingFailure
      { X711CMI::ProcessingFailureType error_info; };
exception ProcessingFailureEmpty { };
exception SetListError
      { X711CMI::SetListErrorType error_info; };
exception SyncNotSupported
      { X711CMI::CMISSyncType error_info; };
exception NoSuchEventType
      { X711CMI::NoSuchEventTypeType error_info; };
exception NoSuchInvokeId
      { X711CMI::InvokeIdTypeType error_info; };


// Using Multiple Replies exception for Actions
interface RepliesIterator; // forward declaration
exception UsingMR
      { RepliesIterator replies_iterator; };

// Definition of specific types used within this module
typedef string NameString;
typedef sequence<ASN1_ObjectIdentifier> ASN1_ObjectIdentifierSeq;
struct AttributeValue {
      ASN1_ObjectIdentifier attribute_id;
      ASN1_DefinedAny value;
};
typedef sequence<AttributeValue> AttributeValueSeq;

// Type to be used in cmis_create operations
enum CreationKind
      {simple, autonaming, subordinate};



// Type to be used in scoped set operations
enum ModifyOperator
      {replace, add_member, remove_member, replace_with_default};

struct AttributeSetOperator {
      ModifyOperator modify_operator;
      ASN1_ObjectIdentifier attribute_id;
      ASN1_DefinedAny attribute_value;
};
typedef sequence <AttributeSetOperator> SetOperationArgument;

// Forward declaration for ReplyHandler interfaces
interface LinkedReplyHandler;
interface EndOfRepliesHandler;
```

```
// ProxyAgent
interface ProxyAgent : JIDM::ProxyAgent {

    void cmis_create (
            in CORBA::ScopedName interface_name,
            in CreationKind creation_kind,
            in CosNaming::Name object_name,
            in X711CMI::AccessControlTypeOpt access_control,
            in CosNaming::Name reference_object,
            in AttributeValueSeq req_attribute_values,
            in LinkedReplyHandler reply_handler
);

    void cmis_create_sync (
            in CORBA::ScopedName interface_name,
            in CreationKind creation_kind,
            in CosNaming::Name object_name,
            in X711CMI::AccessControlTypeOpt access_control,
            in CosNaming::Name reference_object,
            in AttributeValueSeq req_attribute_values,
            out CORBA::ScopedName created_interface_name,
            out CosNaming::Name created_object_name,
            out X711CMI::ASN1_GeneralizedTimeOpt creation_time,
            out AttributeValueSeq created_attribute_values
    ) raises (CREATE_ERRORS);

    void cmis_get (
            in CORBA::ScopedName interface_name,
            in CosNaming::Name object_name,
            in X711CMI::ScopeType scope,
            in X711CMI::CMISFilterType filter,
            in X711CMI::CMISSyncType synchronization,
            in X711CMI::AccessControlTypeOpt access_control,
            in ASN1_ObjectIdentifierSeq attribute_id_list,
            in LinkedReplyHandler reply_handler,
            in EndOfRepliesHandler end_of_replies_handler
    );

    void cmis_set (
            in CORBA::ScopedName interface_name,
            in CosNaming::Name object_name,
            in X711CMI::ScopeType scope,
            in X711CMI::CMISFilterType filter,
            in X711CMI::CMISSyncType synchronization,
            in X711CMI::AccessControlTypeOpt access_control,
            in SetOperationArgument modification_list,
            in LinkedReplyHandler reply_handler,
            in EndOfRepliesHandler end_of_replies_handler
    );

    void cmis_action (
            in CORBA::ScopedName interface_name,
            in CosNaming::Name object_name,
            in X711CMI::ScopeType scope,
            in X711CMI::CMISFilterType filter,
```

```
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in ASN1_ObjectIdentifier action_name,
                in ASN1_DefinedAny action_info,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
        );

        void cmis_delete (
                in CORBA::ScopedName interface_name,
                in CosNaming::Name object_name,
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
        );
};

const ASN1_ObjectIdentifier ACTUAL_CLASS = "2.9.3.4.3.42";

interface ManagedObject; // forward declaration

interface NamingContext : CosNaming::NamingContext {
        // NOTE: These operations are optional
        ManagedObject resolve_with_intf (
                        in CORBA::ScopedName interface_name,
                        in CosNaming::Name object_name
        ) raises (NotFound, CannotProceed, InvalidName);

        ManagedObject resolve_osi_name (
                in ASN1_ObjectIdentifier managed_object_class,
                in X711CMI::ObjectInstanceType object_instance
        ) raises (NotFound, CannotProceed, InvalidName);

        CosNaming::Name translate_osi_name (
                in X711CMI::ObjectInstanceType object_instance
        ) raises (InvalidName);

        X711CMI::ObjectInstanceType translate_idl_name (
                in CosNaming::Name idl_name
        ) raises (InvalidName);
};

// ManagedObject
interface ManagedObject : NamingContext, CosLifeCycle::LifeCycleObject {
        readonly attribute CosNaming::Name object_name;

        void scoped_get (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in ASN1_ObjectIdentifierSeq attribute_id_list,
```

```
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
        );

        void scoped_set (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in SetOperationArgument modification_list,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
        );

        void scoped_action (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in ASN1_ObjectIdentifier action_name,
                in ASN1_DefinedAny action_info,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
        );

        void scoped_delete (
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType synchronization,
                in X711CMI::AccessControlTypeOpt access_control,
                in LinkedReplyHandler reply_handler,
                in EndOfRepliesHandler end_of_replies_handler
        );

        AttributeValueSeq get_attributes (
                in ASN1_ObjectIdentifierSeq attribute_id_list
        ) raises (GET_ERRORS);

        AttributeValueSeq set_attributes (
                in SetOperationArgument modification_list
        ) raises (SET_ERRORS);

        ASN1_DefinedAny perform_action (
                in ASN1_ObjectIdentifier action_name,
                in ASN1_DefinedAny action_info
        ) raises (ACTION_ERRORS, UsingMR);

        void delete_mo () raises (DELETE_ERRORS);
};

// ManagedObjectFactory
interface ManagedObjectFactory {
        ManagedObject create (
                in CORBA::ScopedName interface_name,
                in CosNaming::Name object_name,
```

```
                in ManagedObject reference_object,
                in AttributeValueSeq requested_attribute_values
        ) raises (CREATE_ERRORS);

        ManagedObject create_with_auto_naming (
                in CORBA::ScopedName interface_name,
                in ManagedObject reference_object,
                in AttributeValueSeq requested_attribute_values
        ) raises (CREATE_ERRORS);

        ManagedObject create_subordinate (
                in CORBA::ScopedName interface_name,
                in CosNaming::Name superior_name,
                in ManagedObject reference_object,
                in AttributeValueSeq requested_attribute_values
        ) raises (CREATE_ERRORS);
};

// LocalRoot
typedef sequence<ManagedObject> ManagedObjectSeq;

interface LocalRoot : ManagedObject {
        exception NoDescendants {};
        ManagedObjectSeq list_orphans ( );

        ManagedObjectSeq
            list_orphan_descendants (in CosNaming::Name object_name)
            raises (NoDescendants);
};

// LName
interface LName {
        exception InvalidName {};

        readonly attribute boolean is_distinguished_name;
        readonly attribute unsigned long num_components;

        void from_osi_form (in X711CMI::ObjectInstanceType osi_name);
        X711CMI::ObjectInstanceType to_osi_form ()
                        raises(InvalidName);
        void from_idl_form (in CosNaming::Name idl_name);
        CosNaming::Name to_idl_form ()
                        raises(InvalidName);

        LName to_ancestor_name (in unsigned long levels_up)
                        raises(InvalidName);
        LName to_relative_name (in unsigned long levels_up)
                        raises(InvalidName);
        LName append (in LName name);
        LName append_ava (in X501Inf::AttributeValueAssertionType ava)
                        raises(InvalidName);
        X501Inf::AttributeValueAssertionType get_ava (in unsigned long index)
                        raises(InvalidName);

        boolean equals (in LName name);
```

```
        LName copy ();

        void from_string_form (in NameString name_string);
        NameString to_string_form ()
                        raises(InvalidName);
        void destroy ();
};

// ReplyHandler interfaces
interface LinkedReplyHandler {
        void send_reply (
                in CORBA::ScopedName object_interface,
                in CosNaming::Name object_name,
                in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                in any reply_info
        );

        void send_mo_error (
                in CORBA::ScopedName object_interface,
                in CosNaming::Name object_name,
                in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                in short error_code,
                in any error_info
        );

        void send_subtree_error (
                in CORBA::ScopedName object_interface,
                in CosNaming::Name object_name,
                in X711CMI::ASN1_GeneralizedTimeOpt current_time,
                in short error_code,
                in any error_info
        );
};

interface EndOfRepliesHandler {
        void end_of_replies ( );
};

interface MultipleRepliesHandler : LinkedReplyHandler, EndOfRepliesHandler {};

// BufferedRepliesHandler
        struct Reply {
        CORBA::ScopedName  object_interface;
        CosNaming::Name  object_name;
        X711CMI::ASN1_GeneralizedTimeOpt  current_time;
        any reply_info;
};
typedef sequence<Reply> ReplyList;

interface RepliesIterator  {
        exception MoError {
        CORBA::ScopedName object_interface;
        CosNaming::Name object_name;
        X711CMI::ASN1_GeneralizedTimeOpt current_time;
        short error_code;
```

```
        any error_info;
    };

    exception SubtreeError {
        CORBA::ScopedName object_interface;
        CosNaming::Name object_name;
        X711CMI::ASN1_GeneralizedTimeOpt current_time;
        short error_code;
        any error_info;
    };

    boolean  get_reply (out Reply r) raises (MoError, SubtreeError);

    boolean  get_n_replies (in unsigned long how_many, out ReplyList r_list)
                raises (MoError, SubtreeError);

    boolean  finished (out unsigned long num_pending);
    void destroy ();
    };

interface BufferedRepliesHandler : MultipleRepliesHandler, RepliesIterator {};

};

#define UsingMR OSIMgmt::UsingMR

#endif /* _OSIMGMT_IDL_ */
```

B.1.3 SNMPMgmt.idl

```
// File: SNMPMgmt.idl
#ifndef _SNMPMGMT_IDL_
#define _SNMPMGMT_IDL_

#include <orb.idl>
#include <CosPropertyService.idl>
#include <ASN1Types.idl>
#include <JIDM.idl>

#pragma prefix “jidm.org”

module SNMPMgmt {
    const string ManagementDomainKeyId = “Internet Management”;
    const string ManagementDomainKeyKind = “XSM environment”;
    const string ProtocolVer = “Protocol Version”;
    const string TransportProtocol = “Transport Protocol”;
    const string DomainTitle = “Domain Title”;
    const string TransportAddress = “Transport Address”;
    const string TransportPort = “Transport Port”;
    const string CommunityName = “Community Name”;
    const string ContextEngineID = “Context EngineID”;
    const string ContextName = “Context Name”;

// Redefinition of types
```

```
typedef CORBA::ScopedName ScopedName;
typedef CosLifeCycle::Criteria Criteria;
typedef CosPropertyService::PropertyName VarName;
typedef CosPropertyService::PropertyNames VarNameList;
typedef CosPropertyService::Property NameValuePair;
typedef CosPropertyService::Properties NVPairList;

typedef ASN1_ObjectIdentifier EntryIndex;
typedef sequence < EntryIndex > EntryIndexList;

typedef string TAddress; // Transport address of an agent

enum ProtocolVersion { snmpV1, snmpV2c, snmpV3 };

// SNMP Protocol specific exceptions
exception ProtocolError {
      ASN1_Integer error_status;
      ASN1_Integer error_index;
};
exception MultVarProtocolError {
      ASN1_Integer error_status;
      VarNameList  error_var_list;
      NVPairList   result_var_list;
};

// SMI information module specific exceptions.
exception NoSuchSmiModule { };
exception NoSuchSmiEntry { };
exception NoSuchVariable { };

// MIB entry specific exceptions
exception NoSuchHost { };
exception NoSuchObject { };
exception EndOfMibView { };

exception AlreadyExists { };

interface SmiEntry : CosLifeCycle::LifeCycleObject,
                     CosPropertyService::PropertySet {
      // the value of entry_name is always “0” for the groups.
      readonly attribute ASN1_ObjectIdentifier entry_name;
};
typedef sequence < SmiEntry > SmiEntryList;

interface SmiTableIterator {
      boolean next_one_entry( out SmiEntry smi_entry );
      boolean next_n_entries (
            in unsigned long how_many,
            out SmiEntryList smi_entry_list
      );
      void destroy();
};

 interface GenericFactory : CosLifeCycle::GenericFactory {
      SmiEntry create_mib_entry (
```

```
            in ScopedName t_entry_type,
            in ASN1_ObjectIdentifier entry_index,
            in Criteria create_criteria
        ) raises ( NoSuchSmiEntry, AlreadyExists );

        SmiEntry create_mib_entry_with_auto_name (
            in ScopedName t_entry_type,
            in Criteria create_criteria
        ) raises ( NoSuchSmiEntry, AlreadyExists );
};

interface GetNextEntryIterator {
        // Get the next entry index according to lexical ordering rule
        // of SNMP OIDs -- follows SNMP get-next traversal rule
        boolean next_one_entry ( out EntryIndex entry_index );
        boolean next_n_entries (
                in unsigned long how_many,
                out EntryIndexList entry_index_list
        );
        void destroy();
};

// NamingContext extends CosNaming::NamingContext to provide
// navigating the SNMP name space in the lexicographic order
// and SNMP specific name and context resolution

interface NamingContext : CosNaming::NamingContext {
        string get_next_entry(
                in string entry_name
        ) raises ( InvalidName, NotFound, CannotProceed );

        GetNextEntryIterator get_next_entry_iterator(
                in string initial_entry_name
        ) raises ( InvalidName, NotFound );
};

interface NamingDirectory : NamingContext {
        NamingContext resolve_domain_context(
                in TAddress p_host_name
        ) raises ( NoSuchHost, CannotProceed, InvalidName, NotFound );

        NamingContext resolve_smi_module(
                in TAddress p_host_name,
                in string p_smi_module_name
        ) raises ( NoSuchHost, NoSuchSmiModule, InvalidName, NotFound );

        NamingContext resolve_smi_entry(
                in TAddress p_host_name,
                in ScopedName p_entry_type
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed, InvalidName,
                        NotFound );

        SmiEntry resolve_mib_entry(
                in TAddress p_host_name,
                in ScopedName p_entry_type,
```

```
            in string p_entry_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed, InvalidName,
                    NotFound );

        void list_smi_entries(
            in TAddress p_host_name,
            in ScopedName p_entry_type,
            in unsigned long how_many,
            out SmiEntryList out_list,
            out SmiTableIterator table_iterator
        ) raises ( NoSuchHost, NoSuchSmiEntry, CannotProceed, InvalidName,
                    NotFound );
};

        // ProxyAgent

interface ProxyAgent : JIDM::ProxyAgent {

        readonly attribute TAddress host_name;

        ASN1_Any get_a_variable (
            in TAddress p_host_name,
            in ScopedName p_var_scoped_name,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchVariable, NoSuchObject, ProtocolError );

        NVPairList get_variables (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in VarNameList p_var_name_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
                    MultVarProtocolError );

        void set_a_variable (
            in TAddress p_host_name,
            in ScopedName p_var_scoped_name,
            in EntryIndex p_var_index,
            in ASN1_Any p_var_new_value
        ) raises ( NoSuchHost, NoSuchVariable, NoSuchObject, ProtocolError );

        void set_variables (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in NVPairList p_var_nvp_list,
            in EntryIndex p_var_index
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject,
                    MultVarProtocolError );

        void list_mib_entries(
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name,
            in long p_how_many,
            out EntryIndexList p_entry_index_list,
            out GetNextEntryIterator p_entry_name_list_itr
```

```
        ) raises ( NoSuchHost, NoSuchSmiEntry, NoSuchObject, ProtocolError );

        boolean mib_entry_exists (
            in TAddress p_host_name,
            in ScopedName p_entry_scoped_name
        ) raises ( NoSuchHost, NoSuchSmiEntry, ProtocolError );

        boolean is_mib_module_supported (
            in TAddress p_host_name,
            in string p_smi_module_name
        ) raises ( NoSuchHost, NoSuchSmiModule, ProtocolError );

};
struct EntryVarBind {
        ScopedName entry_name; // IDL scoped name of the interface for table-entry
        string entry_index; // row index of an entry in the form of ObjectId string
        CosPropertyService::Properties nvp_list;
};

typedef sequence<EntryVarBind> EntryVarBindList;
typedef EntryVarBindList NotificationVariableList;
typedef EntryVarBindList InformVariableList;

struct NotificationInfo { // to be sent when using untyped event channel
        CosNaming::Name src_entry_name;
        ScopedName event_type;
        ASN1_GeneralizedTime event_time;
        any notification_info;
};
struct InformInfo { // to be sent when using untyped event channel
        CosNaming::Name src_obj_name;
        InformVariableList inform_info;
};

interface Notifications {
        void snmp_notification (
            in CosNaming::Name src_entry_name,
            in ScopedName event_type,
            in ASN1_GeneralizedTime event_time,
            in any notification_info
        );
        void snmp_inform (
            in CosNaming::Name src_entry_name,
            in InformVariableList inform_variables
        );
        void snmp_report (
            in CosNaming::Name src_entry_name,
            in InformVariableList report_variables
        );
};

interface PullNotifications {
        boolean try_snmp_notification (
            out CosNaming::Name src_entry_name,
            out ScopedName event_type,
```

```
            out ASN1_GeneralizedTime event_time,
            out any notification_info
        );
        void pull_snmp_notification (
            out CosNaming::Name src_entry_name,
            out ScopedName event_type,
            out ASN1_GeneralizedTime event_time,
            out any notification_info
        );

        boolean try_snmp_inform (
            out CosNaming::Name src_entry_name,
            out InformVariableList inform_variables
        );

        void pull_snmp_inform (
            out CosNaming::Name src_entry_name,
            out InformVariableList inform_variables
        );

        boolean try_snmp_report (
            out CosNaming::Name src_entry_name,
            out InformVariableList report_variables
        );

        void pull_snmp_report (
            out CosNaming::Name src_entry_name,
            out InformVariableList report_variables
        );
    };

};

#endif /* _SNMPMGMT_IDL_ */
```

B.2 Imported IDL

IDL listed in this section is specified elsewhere, but is listed here for ease of reference and completeness. The ultimate authority for these IDLs should be the original source of the IDL specifications.

B.2.1 ASN1Types.idl

Specified in the JIDM Specification Translation document XoJIDM (see “#endif /* _SNMPMIR_IDL_ */” on page B-54), and amended by the JIDM Specification Translation Issues List resolutions.

```
// File: ASN1Types.idl
#ifndef _ASN1TYPES_IDL_
#define _ASN1TYPES_IDL_

#pragma prefix “jidm.org”
```

```
// ASN.1 base types

// Null type
typedef char                ASN1_Null;
const ASN1_Null             ASN1_NullValue = '\000';

// Boolean
typedef boolean             ASN1_Boolean;

// Integers
typedef unsigned short      ASN1_Unsigned16;
typedef unsigned long       ASN1_Unsigned;
typedef unsigned long long  ASN1_Unsigned64;
typedef short               ASN1_Integer16;
typedef long                ASN1_Integer;
typedef long long           ASN1_Integer64;

// Real
typedef double              ASN1_Real;

// ASN.1 strings which may not contain binary zeros
typedef string              ASN1_NumericString;
typedef string              ASN1_PrintableString;
typedef string              ASN1_VisibleString;
typedef ASN1_VisibleString  ASN1_ISO646String;
typedef string              ASN1_GraphicString;
typedef ASN1_GraphicString  ASN1_ObjectDescriptor;
typedef string              ASN1_TeletexString;
typedef ASN1_TeletexString  ASN1_T61String;

// Times
typedef ASN1_VisibleString  ASN1_GeneralizedTime; // PIDL defined
typedef ASN1_VisibleString  ASN1_UTCTime;

// ASN.1 strings which may contain binary zeros
typedef sequence<octet>     ASN1_OctetString;
typedef sequence<octet>     ASN1_GeneralString;
typedef sequence<octet>     ASN1_IA5String;
typedef sequence<octet>     ASN1_VideotexString;

// ASN.1 strings of wide characters (which may contain binary zeros)
typedef sequence<unsigned short>   ASN1_BMPString;
typedef sequence<unsigned long>    ASN1_UniversalString;

// Object Identifier
typedef string              ASN1_ObjectIdentifier;


// Bit String
typedef sequence<octet>     ASN1_BitString; // PIDL defined

// Any
typedef any                 ASN1_Any;
typedef any                 ASN1_DefinedAny;
```

```
// ASN.1 recursive references
typedef any                         ASN1_Recursive;


// External

module X208Ext {

  union ASN1_ObjectIdentifierOpt
    switch (boolean) {
      case TRUE: ASN1_ObjectIdentifier value;
  };

  union ASN1_IntegerOpt
    switch (boolean) {
      case TRUE: ASN1_Integer value;
  };

  union ASN1_ObjectDescriptorOpt
    switch (boolean) {
      case TRUE: ASN1_ObjectDescriptor value;
  };

  enum ExternalEncodingTypeChoice { single_ASN1_typeChoice,
      octet_alignedChoice, arbitraryChoice };

  union ExternalEncodingType
    switch(ExternalEncodingTypeChoice) {
      case single_ASN1_typeChoice:
          ASN1_Any single_ASN1_type;
      case octet_alignedChoice:
          ASN1_OctetString octet_aligned;
      case arbitraryChoice:
          ASN1_BitString arbitrary;
  };

     struct ExternalType {
          ASN1_ObjectIdentifierOpt direct_reference;
          ASN1_IntegerOpt indirect_reference;
          ASN1_ObjectDescriptorOpt data_value_descriptor;
          ExternalEncodingType encoding;
     };

};

typedef X208Ext::ExternalType ASN1_External;


// define constants for ASN.1 Real infinity values
#include <ASN1Limits.idl>
const  ASN1_Real  plus_infinity  = MAX_FLT;
const  ASN1_Real  minus_infinity = MIN_FLT;

#endif /*  _ASN1TYPES_IDL_ */
```

B.2.2 ASN1Limits.idl

Specified in the JIDM Specification Translation document XoJIDM and amended by the JIDM Specification Translation Issues List resolutions.

```
// File: ASN1Limits.idl
#ifndef _ASN1LIMITS_IDL_
#define _ASN1LIMITS_IDL_

// Substitute <MAX> and <MIN> by the max and min (biggest negative)
// double values your machine can hold for IDL interfaces.
// Conditional compilation can be used to support multiple architectures.

#define MIN_FLT <MIN>
#define MAX_FLT <MAX>

#endif /* _ASN1LIMITS_IDL_ */
```

B.3 Generated IDL

IDL listed in this section is automatically generated, following the JIDM Specification Translation process, as specified in XoJIDM and amended by the JIDM Specification Translation Issues List resolutions. It is listed here for ease of reference and completeness. The ultimate authority for these IDLs should be the use of the originally published document and a compliant JIDM Specification Translation compiler.

B.3.1 X501Inf.idl

The original source for the ASN.1 document that translates into this IDL is [X501].

```
// File: X501Inf.idl
#ifndef _X501INF_IDL_
#define _X501INF_IDL_

//
//  ASN.1 Module name: InformationFramework
//  ASN.1 Module OID:  2.5.1.1
//  ASN.1 Module nickname: X501Inf
//

#include <ASN1Types.idl>

module X501Inf {

    // Assignments mapping

    typedef ASN1_ObjectIdentifier AttributeTypeType;

    typedef ASN1_Any AttributeValueType;

    typedef sequence <AttributeValueType>
        AttributeValuesType;
```

```
        struct AttributeType {
            AttributeTypeType type;
            AttributeValuesType values;
        };

        struct AttributeValueAssertionType {
            AttributeTypeType attributeType;
            ASN1_DefinedAny attributeValue;  // defined by:attributeType
        };

        typedef sequence <AttributeValueAssertionType>
            RelativeDistinguishedNameType;

        typedef sequence <RelativeDistinguishedNameType>
            RDNSequenceType;

        enum NameTypeChoice { rDNSequenceChoice };

        union NameType
            switch(NameTypeChoice) {
            case rDNSequenceChoice:
            RDNSequenceType rDNSequence;
        };

        typedef RDNSequenceType DistinguishedNameType;

        //NO complex constant declarations
        //interface ConstValues empty
    };

    #endif  /* _X501INF_IDL_ */
```

B.3.2 X227ACS.idl

The original source for the ASN.1 document that translates into this IDL is [X227].

```
// File: X227ACS.idl
#ifndef _X227ACS_IDL_
#define _X227ACS_IDL_

//
//  ASN.1 Module name: ACSE-1
//  ASN.1 Module OID:  2.2.0.0.1
//  ASN.1 Module nickname: X227ACS
//

#include <ASN1Types.idl>

#include "X501Inf.idl"

module X227ACS {

    // definitions imported from: X501Inf
```

```
typedef X501Inf::NameType
      NameType;

typedef X501Inf::RelativeDistinguishedNameType
      RelativeDistinguishedNameType;

// Assignments mapping

const ASN1_ObjectIdentifier acse_as_id =
      “2.2.1.0.1”;

const ASN1_ObjectIdentifier aCSE_id =
      “2.2.3.1.1”;

typedef ASN1_BitString AARQ_apduProtocol_versionType;

union AARQ_apduProtocol_versionTypeOpt
      switch (boolean) {
            case TRUE: AARQ_apduProtocol_versionType value;
};

typedef AARQ_apduProtocol_versionTypeOpt AARQ_apduProtocol_versionTypeDef;

typedef ASN1_ObjectIdentifier Application_context_nameType;

typedef NameType AP_title_form1Type;

typedef ASN1_ObjectIdentifier AP_title_form2Type;

enum AP_titleTypeChoice { form1Choice, form2Choice };

union AP_titleType
      switch(AP_titleTypeChoice) {
            case form1Choice:
      AP_title_form1Type form1;
            case form2Choice:
      AP_title_form2Type form2;
};

union AP_titleTypeOpt
      switch (boolean) {
            case TRUE: AP_titleType value;
};

typedef RelativeDistinguishedNameType AE_qualifier_form1Type;

typedef ASN1_Integer AE_qualifier_form2Type;

enum AE_qualifierTypeChoice { form1Choice_1, form2Choice_1 };

union AE_qualifierType
      switch(AE_qualifierTypeChoice) {
            case form1Choice_1:
      AE_qualifier_form1Type form1;
```

```
            case form2Choice_1:
        AE_qualifier_form2Type form2;
};

union AE_qualifierTypeOpt
        switch (boolean) {
            case TRUE: AE_qualifierType value;
};

typedef ASN1_Integer AP_invocation_identifierType;

union AP_invocation_identifierTypeOpt
        switch (boolean) {
            case TRUE: AP_invocation_identifierType value;
        };

typedef ASN1_Integer AE_invocation_identifierType;

union AE_invocation_identifierTypeOpt
        switch (boolean) {
            case TRUE: AE_invocation_identifierType value;
};

typedef ASN1_BitString ACSE_requirementsType;

union ACSE_requirementsTypeOpt
        switch (boolean) {
            case TRUE: ACSE_requirementsType value;
};

 typedef ASN1_ObjectIdentifier Mechanism_nameType;

        union Mechanism_nameTypeOpt
            switch (boolean) {
        case TRUE: Mechanism_nameType value;
 };

struct Authentication_valueOtherType {
        Mechanism_nameType other_mechanism_name;
            ASN1_DefinedAny other_mechanism_value;  // defined
                    by:other_mechanism_name
 };

enum Authentication_valueTypeChoice { charstringChoice,
        bitstringChoice, externalChoice, otherChoice };

union Authentication_valueType
        switch(Authentication_valueTypeChoice) {
            case charstringChoice:
                ASN1_GraphicString charstring;
             case bitstringChoice:
                ASN1_BitString bitstring;
            case externalChoice:
                ASN1_External external;
            case otherChoice:
```

```
                Authentication_valueOtherType other;
};

union Authentication_valueTypeOpt
    switch (boolean) {
        case TRUE: Authentication_valueType value;
};

typedef ASN1_GraphicString Implementation_dataType;

union Implementation_dataTypeOpt
    switch (boolean) {
        case TRUE: Implementation_dataType value;
};

typedef sequence <ASN1_External>
    Association_informationType;

union Association_informationTypeOpt
    switch (boolean) {
        case TRUE: Association_informationType value;
};

struct AARQ_apduType {
    AARQ_apduProtocol_versionTypeDef protocol_version;
    Application_context_nameType application_context_name;
    AP_titleTypeOpt called_AP_title;
    AE_qualifierTypeOpt called_AE_qualifier;
    AP_invocation_identifierTypeOpt called_AP_invocation_identifier;
    AE_invocation_identifierTypeOpt called_AE_invocation_identifier;
    AP_titleTypeOpt calling_AP_title;
    AE_qualifierTypeOpt calling_AE_qualifier;
    AP_invocation_identifierTypeOpt calling_AP_invocation_identifier;
    AE_invocation_identifierTypeOpt calling_AE_invocation_identifier;
    ACSE_requirementsTypeOpt sender_acse_requirements;
    Mechanism_nameTypeOpt mechanism_name;
    Authentication_valueTypeOpt calling_authentication_value;
    Implementation_dataTypeOpt implementation_information;
    Association_informationTypeOpt user_information;
};

typedef ASN1_BitString AARE_apduProtocol_versionType;

union AARE_apduProtocol_versionTypeOpt
    switch (boolean) {
        case TRUE: AARE_apduProtocol_versionType value;
};

typedef AARE_apduProtocol_versionTypeOpt AARE_apduProtocol_versionTypeDef;

typedef ASN1_Integer Associate_resultType;

typedef ASN1_Integer Associate_source_diagnosticAcse_service_userType;

typedef ASN1_Integer Associate_source_diagnosticAcse_service_providerType;
```

```
enum Associate_source_diagnosticTypeChoice {
    acse_service_userChoice, acse_service_providerChoice };

union Associate_source_diagnosticType
    switch(Associate_source_diagnosticTypeChoice) {
        case acse_service_userChoice:
            Associate_source_diagnosticAcse_service_userType acse_service_user;
        case acse_service_providerChoice:
            Associate_source_diagnosticAcse_service_providerType
    acse_service_provider;
};

struct AARE_apduType {
    AARE_apduProtocol_versionTypeDef protocol_version;
    Application_context_nameType application_context_name;
    Associate_resultType result;
    Associate_source_diagnosticType result_source_diagnostic;
    AP_titleTypeOpt responding_AP_title;
    AE_qualifierTypeOpt responding_AE_qualifier;
    AP_invocation_identifierTypeOpt responding_AP_invocation_identifier;
    AE_invocation_identifierTypeOpt responding_AE_invocation_identifier;
    ACSE_requirementsTypeOpt responder_acse_requirements;
    Mechanism_nameTypeOpt mechanism_name;
    Authentication_valueTypeOpt responding_authentication_value;
    Implementation_dataTypeOpt implementation_information;
    Association_informationTypeOpt user_information;
    };

typedef ASN1_Integer Release_request_reasonType;

union Release_request_reasonTypeOpt
    switch (boolean) {
        case TRUE: Release_request_reasonType value;
};

struct RLRQ_apduType {
    Release_request_reasonTypeOpt reason;
    Association_informationTypeOpt user_information;
};

typedef ASN1_Integer Release_response_reasonType;

union Release_response_reasonTypeOpt
    switch (boolean) {
        case TRUE: Release_response_reasonType value;
};

struct RLRE_apduType {
    Release_response_reasonTypeOpt reason;
    Association_informationTypeOpt user_information;
};

typedef ASN1_Integer ABRT_sourceType;
```

```
enum ABRT_diagnosticType { no_reason_given, protocol_error,
      authentication_mechanism_name_not_recognized,
      authentication_mechanism_name_required,
      authentication_failure, authentication_required };

union ABRT_diagnosticTypeOpt
      switch (boolean) {
      case TRUE: ABRT_diagnosticType value;
};

struct ABRT_apduType {
      ABRT_sourceType abort_source;
      ABRT_diagnosticTypeOpt abort_diagnostic;
      Association_informationTypeOpt user_information;
};

enum ACSE_apduTypeChoice { aarqChoice, aareChoice, rlrqChoice,
      rlreChoice, abrtChoice };

union ACSE_apduType
      switch(ACSE_apduTypeChoice) {
      case aarqChoice:
            AARQ_apduType aarq;
      case aareChoice:
            AARE_apduType aare;
      case rlrqChoice:
            RLRQ_apduType rlrq;
      case rlreChoice:
            RLRE_apduType rlre;
      case abrtChoice:
            ABRT_apduType abrt;
};

const unsigned long version1 =
      0;

const unsigned long version1_1 =
      0;

const ABRT_sourceType acse_service_user =
      0;

const ABRT_sourceType acse_service_provider =
      1;

const unsigned long authentication =
      0;

typedef NameType AE_title_form1Type;

typedef ASN1_ObjectIdentifier AE_title_form2Type;

enum AE_titleTypeChoice { form1Choice_2, form2Choice_2 };
union AE_titleType
      switch(AE_titleTypeChoice) {
```

```
            case form1Choice_2:
                AE_title_form1Type form1;
            case form2Choice_2:
                AE_title_form2Type form2;
};

const Associate_resultType accepted =
    0;

const Associate_resultType rejected_permanent =
    1;

const Associate_resultType rejected_transient =
    2;

const Associate_source_diagnosticAcse_service_userType null =
    0;

const Associate_source_diagnosticAcse_service_userType no_reason_given_1 =
    1;

const Associate_source_diagnosticAcse_service_userType
        application_context_name_not_supported =
     2;

const Associate_source_diagnosticAcse_service_userType
        calling_AP_title_not_recognized =
    3;

const Associate_source_diagnosticAcse_service_userType
        calling_AP_invocation_identifier_not_recognized =
    4;

const Associate_source_diagnosticAcse_service_userType
        calling_AE_qualifier_not_recognized =
     5;

const Associate_source_diagnosticAcse_service_userType
        calling_AE_invocation_identifier_not_recognized =
    6;

const Associate_source_diagnosticAcse_service_userType
        called_AP_title_not_recognized =
    7;

const Associate_source_diagnosticAcse_service_userType
        called_AP_invocation_identifier_not_recognized =
    8;

    const Associate_source_diagnosticAcse_service_userType
        called_AE_qualifier_not_recognized =
    9;

const Associate_source_diagnosticAcse_service_userType
        called_AE_invocation_identifier_not_recognized =
```

```
    10;

const Associate_source_diagnosticAcse_service_userType
        authentication_mechanism_name_not_recognized_1 =
    11;

const Associate_source_diagnosticAcse_service_userType
        authentication_mechanism_name_required_1 =
    12;

const Associate_source_diagnosticAcse_service_userType authentication_failure_1 =
    13;

const Associate_source_diagnosticAcse_service_userType
        authentication_required_1 =
    14;

const Associate_source_diagnosticAcse_service_providerType null_1 =
    0;

const Associate_source_diagnosticAcse_service_providerType no_reason_given_2 =
    1;

const Associate_source_diagnosticAcse_service_providerType
        no_common_acse_version =
     2;

const Release_request_reasonType normal =
    0;

const Release_request_reasonType urgent =
    1;

const Release_request_reasonType user_defined =
    30;

const Release_response_reasonType normal_1 =
    0;

const Release_response_reasonType not_finished =
    1;

const Release_response_reasonType user_defined_1 =
    30;

// Complex constants declaration.

 interface ConstValues {

    // ** Generated values for <AARQ_apduType::protocol_version>:

    AARQ_apduProtocol_versionType protocol_versionDefault();
    // returns: {(ASN.1: version1)}

     // ** Generated values for <AARE_apduType::protocol_version>:
```

```
            AARE_apduProtocol_versionType protocol_versionDefault_1();
            // returns: {(ASN.1: version1)}
            };

    };

#endif  /* _X227ACS_IDL_ */
```

B.3.3 X711CMI.idl

The original source for the ASN.1 document that translates into this IDL is [X711].

```
// File: X711CMI.idl
#ifndef _X711CMI_IDL_
#define _X711CMI_IDL_

//
//  ASN.1 Module name: CMIP-1
//  ASN.1 Module OID:  2.9.1.0.3
//  ASN.1 Module nickname: X711CMI
//

#include <ASN1Types.idl>

#include "X501Inf.idl"

module X711CMI {

    // definitions imported from: X501Inf

    typedef X501Inf::DistinguishedNameType
        DistinguishedNameType;

    typedef X501Inf::RDNSequenceType
        RDNSequenceType;

    // Assignments mapping


    typedef ASN1_Integer InvokeIdTypeType;

    typedef ASN1_External AccessControlType;

    enum ObjectClassTypeChoice { globalFormChoice_3, localFormChoice_3 };

    union ObjectClassType
        switch(ObjectClassTypeChoice) {
            case globalFormChoice_3:
                ASN1_ObjectIdentifier globalForm;
            case localFormChoice_3:
                ASN1_Integer localForm;
    };
```

```
enum ObjectInstanceTypeChoice { distinguishedNameChoice,
     nonSpecificFormChoice, localDistinguishedNameChoice };

union ObjectInstanceType
     switch(ObjectInstanceTypeChoice) {
          case distinguishedNameChoice:
               DistinguishedNameType distinguishedName;
          case nonSpecificFormChoice:
               ASN1_OctetString nonSpecificForm;
          case localDistinguishedNameChoice:
               RDNSequenceType localDistinguishedName;
};

union AccessControlTypeOpt
      switch (boolean) {
          case TRUE: AccessControlType value;
 };

enum CMISSyncType { bestEffort, atomic };

union CMISSyncTypeOpt
     switch (boolean) {
          case TRUE: CMISSyncType value;
};

typedef CMISSyncTypeOpt CMISSyncTypeDef;

typedef ASN1_Integer ScopeLevelType;

enum ScopeTypeChoice { levelChoice, individualLevelsChoice,
     baseToNthLevelChoice };

union ScopeType
     switch(ScopeTypeChoice) {
          case levelChoice:
               ScopeLevelType level;
          case individualLevelsChoice:
               ASN1_Integer individualLevels;
          case baseToNthLevelChoice:
               ASN1_Integer baseToNthLevel;
};

union ScopeTypeOpt
     switch (boolean) {
          case TRUE: ScopeType value;
};

typedef ScopeTypeOpt ScopeTypeDef;

enum AttributeIdTypeChoice { globalFormChoice_1, localFormChoice_1 };

union AttributeIdType
     switch(AttributeIdTypeChoice) {
          case globalFormChoice_1:
               ASN1_ObjectIdentifier globalForm;
```

```
            case localFormChoice_1:
                ASN1_Integer localForm;
};

struct FilterItemSubstringsItemInitialStringType {
    AttributeIdType attributeId;
        ASN1_DefinedAny string_1;  // defined by:attributeId
};

struct FilterItemSubstringsItemAnyStringType {
     AttributeIdType attributeId;
        ASN1_DefinedAny string_1;  // defined by:attributeId
};

struct FilterItemSubstringsItemFinalStringType {
    AttributeIdType attributeId;
        ASN1_DefinedAny string_1;  // defined by:attributeId
};

enum FilterItemSubstringsItemTypeChoice { initialStringChoice,
    anyStringChoice, finalStringChoice };

union FilterItemSubstringsItemType
    switch(FilterItemSubstringsItemTypeChoice) {
        case initialStringChoice:
            FilterItemSubstringsItemInitialStringType initialString;
        case anyStringChoice:
            FilterItemSubstringsItemAnyStringType anyString;
        case finalStringChoice:
            FilterItemSubstringsItemFinalStringType finalString;
};

typedef sequence <FilterItemSubstringsItemType>
    FilterItemSubstringsType;

enum FilterItemTypeChoice { equalityChoice, substringsChoice,
    greaterOrEqualChoice, lessOrEqualChoice, presentChoice,
    subsetOfChoice, supersetOfChoice,
    nonNullSetIntersectionChoice };

union FilterItemType
    switch(FilterItemTypeChoice) {
        case equalityChoice:
            AttributeType equality;
        case substringsChoice:
            FilterItemSubstringsType substrings;
        case greaterOrEqualChoice:
            AttributeType greaterOrEqual;
        case lessOrEqualChoice:
            AttributeType lessOrEqual;
        case presentChoice:
             AttributeIdType present;
         case subsetOfChoice:
            AttributeType subsetOf;
        case supersetOfChoice:
```

```
                AttributeType supersetOf;
            case nonNullSetIntersectionChoice:
                AttributeType nonNullSetIntersection;
};

enum CMISFilterTypeChoice { itemChoice, andChoice, orChoice,
    notChoice };

union CMISFilterType
    switch(CMISFilterTypeChoice) {
        case itemChoice:
            FilterItemType item;
        case andChoice:
            sequence<CMISFilterType> and;
        case orChoice:
            sequence<CMISFilterType> or;
        case notChoice:
             sequence<CMISFilterType, 1> not;
};

union CMISFilterTypeOpt
    switch (boolean) {
        case TRUE: CMISFilterType value;
};

typedef CMISFilterTypeOpt CMISFilterTypeDef;

enum ActionTypeIdTypeChoice { globalFormChoice, localFormChoice };

union ActionTypeIdType
    switch(ActionTypeIdTypeChoice) {
        case globalFormChoice:
            ASN1_ObjectIdentifier globalForm;
        case localFormChoice:
            ASN1_Integer localForm;
 };

union ASN1_DefinedAnyOpt
    switch (boolean) {
        case TRUE: ASN1_DefinedAny value;
    };

struct ActionInfoType {
    ActionTypeIdType actionType;
        ASN1_DefinedAnyOpt actionInfoArg;  // defined by:actionType
 };

 struct ActionArgumentType {
    ObjectClassType baseManagedObjectClass;
     ObjectInstanceType baseManagedObjectInstance;
    AccessControlTypeOpt accessControl;
    CMISSyncTypeDef synchronization;
    ScopeTypeDef scope;
    CMISFilterTypeDef filter;
    ActionInfoType actionInfo;
```

```
};

const ScopeLevelType baseObject =
      0;

union ObjectClassTypeOpt
      switch (boolean) {
            case TRUE: ObjectClassType value;
};

union ObjectInstanceTypeOpt
      switch (boolean) {
            case TRUE: ObjectInstanceType value;
 };

union ASN1_GeneralizedTimeOpt
      switch (boolean) {
            case TRUE: ASN1_GeneralizedTime value;
};

enum ActionErrorInfoErrorStatusType { accessDenied, noSuchAction,
      noSuchArgument, invalidArgumentValue };

struct NoSuchArgumentActionIdType {
      ObjectClassTypeOpt managedObjectClass;
      ActionTypeIdType actionType;
};

enum EventTypeIdTypeChoice { globalFormChoice_2, localFormChoice_2 };

union EventTypeIdType
      switch(EventTypeIdTypeChoice) {

case globalFormChoice_2:
       ASN1_ObjectIdentifier globalForm;
case localFormChoice_2:
       ASN1_Integer localForm;
};

struct NoSuchArgumentEventIdType {
      ObjectClassTypeOpt managedObjectClass;
      EventTypeIdType eventType;
};

enum NoSuchArgumentTypeChoice { actionIdChoice, eventIdChoice };

union NoSuchArgumentType
      switch(NoSuchArgumentTypeChoice) {
            case actionIdChoice:
                  NoSuchArgumentActionIdType actionId;
            case eventIdChoice:
                  NoSuchArgumentEventIdType eventId;
};

struct InvalidArgumentValueEventValueType {
```

```
        EventTypeIdType eventType;
        ASN1_DefinedAnyOpt eventInfo;  // defined by:eventType
    };

    enum InvalidArgumentValueTypeChoice { actionValueChoice,
        eventValueChoice };

    union InvalidArgumentValueType
        switch(InvalidArgumentValueTypeChoice) {
            case actionValueChoice:
                ActionInfoType actionValue;
            case eventValueChoice:
                InvalidArgumentValueEventValueType eventValue;
    };

    enum ActionErrorInfoErrorInfoTypeChoice { actionTypeChoice,
        actionArgumentChoice, argumentValueChoice };

    union ActionErrorInfoErrorInfoType
        switch(ActionErrorInfoErrorInfoTypeChoice) {
            case actionTypeChoice:
        ActionTypeIdType actionType;
            case actionArgumentChoice:
                NoSuchArgumentType actionArgument;
            case argumentValueChoice:
                InvalidArgumentValueType argumentValue;
    };

     struct ActionErrorInfoType {
        ActionErrorInfoErrorStatusType errorStatus;
        ActionErrorInfoErrorInfoType errorInfo;
    };

    struct ActionErrorType {
        ObjectClassTypeOpt managedObjectClass;
        ObjectInstanceTypeOpt managedObjectInstance;
        ASN1_GeneralizedTimeOpt currentTime;
        ActionErrorInfoType actionErrorInfo;
    };

    struct ActionReplyType {
        ActionTypeIdType actionType;
        ASN1_DefinedAny actionReplyInfo;  // defined by:actionType
    };

    union ActionReplyTypeOpt
        switch (boolean) {
            case TRUE: ActionReplyType value;
    };

    struct ActionResultType {
        ObjectClassTypeOpt managedObjectClass;
        ObjectInstanceTypeOpt managedObjectInstance;
        ASN1_GeneralizedTimeOpt currentTime;
        ActionReplyTypeOpt actionReply;
```

```
};

enum AttributeErrorErrorStatusType { accessDenied_1,
      noSuchAttribute, invalidAttributeValue, invalidOperation,
      invalidOperator };

typedef ASN1_Integer ModifyOperatorType;

union ModifyOperatorTypeOpt
      switch (boolean) {
            case TRUE: ModifyOperatorType value;
 };

struct AttributeErrorType {
      AttributeErrorErrorStatusType errorStatus;
      ModifyOperatorTypeOpt modifyOperator;
      AttributeIdType attributeId;
      ASN1_DefinedAnyOpt attributeValue;  // defined by:attributeId
 };

enum AttributeIdErrorErrorStatusType { accessDenied_2,
      noSuchAttribute_1 };

struct AttributeIdErrorType {
      AttributeIdErrorErrorStatusType errorStatus;
      AttributeIdType attributeId;
};

 struct BaseManagedObjectIdType {
      ObjectClassType baseManagedObjectClass;
      ObjectInstanceType baseManagedObjectInstance;
};

struct ComplexityLimitationType {
      ScopeTypeOpt scope;
      CMISFilterTypeOpt filter;
      CMISSyncTypeOpt sync;
 };

enum CreateArgumentObjectInstanceTypeChoice {
      managedObjectInstanceChoice, superiorObjectInstanceChoice };

union CreateArgumentObjectInstanceType
      switch(CreateArgumentObjectInstanceTypeChoice) {
            case managedObjectInstanceChoice:
                  ObjectInstanceType managedObjectInstance;
            case superiorObjectInstanceChoice:
                  ObjectInstanceType superiorObjectInstance;
};

union CreateArgumentObjectInstanceTypeOpt
      switch (boolean) {
            case TRUE: CreateArgumentObjectInstanceType value;
};
```

```
typedef sequence <AttributeType>
      CreateArgumentAttributeListType;

union CreateArgumentAttributeListTypeOpt
       switch (boolean) {
            case TRUE: CreateArgumentAttributeListType value;
};

struct CreateArgumentType {
      ObjectClassType managedObjectClass;
      CreateArgumentObjectInstanceTypeOpt objectInstance;
      AccessControlTypeOpt accessControl;
      ObjectInstanceTypeOpt referenceObjectInstance;
      CreateArgumentAttributeListTypeOpt attributeList;
};

typedef sequence <AttributeType>
      CreateResultAttributeListType;

union CreateResultAttributeListTypeOpt
      switch (boolean) {
            case TRUE: CreateResultAttributeListType value;
};

struct CreateResultType {
      ObjectClassTypeOpt managedObjectClass;
      ObjectInstanceTypeOpt managedObjectInstance;
      ASN1_GeneralizedTimeOpt currentTime;
      CreateResultAttributeListTypeOpt attributeList;
};

struct DeleteArgumentType {
      ObjectClassType baseManagedObjectClass;
      ObjectInstanceType baseManagedObjectInstance;
      AccessControlTypeOpt accessControl;
      CMISSyncTypeDef synchronization;
      ScopeTypeDef scope;
      CMISFilterTypeDef filter;
};

enum DeleteErrorDeleteErrorInfoType { accessDenied_3 };

struct DeleteErrorType {
      ObjectClassTypeOpt managedObjectClass;
      ObjectInstanceTypeOpt managedObjectInstance;
      ASN1_GeneralizedTimeOpt currentTime;
      DeleteErrorDeleteErrorInfoType deleteErrorInfo;
};

struct DeleteResultType {
      ObjectClassTypeOpt managedObjectClass;
      ObjectInstanceTypeOpt managedObjectInstance;
      ASN1_GeneralizedTimeOpt currentTime;
};
```

```
struct EventReplyType {
      EventTypeIdType eventType;
      ASN1_DefinedAnyOpt eventReplyInfo;  // defined by:eventType
};

struct EventReportArgumentType {
      ObjectClassType managedObjectClass;
      ObjectInstanceType managedObjectInstance;
      ASN1_GeneralizedTimeOpt eventTime;
      EventTypeIdType eventType;
      ASN1_DefinedAnyOpt eventInfo;  // defined by:eventType
};

union EventReplyTypeOpt
      switch (boolean) {
            case TRUE: EventReplyType value;
 };

struct EventReportResultType {
      ObjectClassTypeOpt managedObjectClass;
      ObjectInstanceTypeOpt managedObjectInstance;
      ASN1_GeneralizedTimeOpt currentTime;
      EventReplyTypeOpt eventReply;
};

typedef sequence <AttributeIdType>
      GetArgumentAttributeIdListType;

union GetArgumentAttributeIdListTypeOpt
      switch (boolean) {
            case TRUE: GetArgumentAttributeIdListType value;
 };

struct GetArgumentType {
      ObjectClassType baseManagedObjectClass;
      ObjectInstanceType baseManagedObjectInstance;
      AccessControlTypeOpt accessControl;
      CMISSyncTypeDef synchronization;
      ScopeTypeDef scope;
      CMISFilterTypeDef filter;
      GetArgumentAttributeIdListTypeOpt attributeIdList;
};

enum GetInfoStatusTypeChoice { attributeIdErrorChoice,
      attribute_1Choice };

union GetInfoStatusType
      switch(GetInfoStatusTypeChoice) {
            case attributeIdErrorChoice:
                  AttributeIdErrorType attributeIdError;
            case attribute_1Choice:
                  AttributeType attribute_1;
};

typedef sequence <GetInfoStatusType>
```

```
        GetListErrorGetInfoListType;

struct GetListErrorType {
        ObjectClassTypeOpt managedObjectClass;
        ObjectInstanceTypeOpt managedObjectInstance;
        ASN1_GeneralizedTimeOpt currentTime;
        GetListErrorGetInfoListType getInfoList;
};

typedef sequence <AttributeType>
        GetResultAttributeListType;

union GetResultAttributeListTypeOpt
        switch (boolean) {
                case TRUE: GetResultAttributeListType value;
};

struct GetResultType {
        ObjectClassTypeOpt managedObjectClass;
        ObjectInstanceTypeOpt managedObjectInstance;
        ASN1_GeneralizedTimeOpt currentTime;
        GetResultAttributeListTypeOpt attributeList;
};

typedef sequence <AttributeType>
        SetResultAttributeListType;

union SetResultAttributeListTypeOpt
        switch (boolean) {
                case TRUE: SetResultAttributeListType value;
};

struct SetResultType {
        ObjectClassTypeOpt managedObjectClass;
        ObjectInstanceTypeOpt managedObjectInstance;
         ASN1_GeneralizedTimeOpt currentTime;
        SetResultAttributeListTypeOpt attributeList;
 };

enum SetInfoStatusTypeChoice { attributeErrorChoice,
        attribute_1Choice_1 };

union SetInfoStatusType
        switch(SetInfoStatusTypeChoice) {
                case attributeErrorChoice:
                        AttributeErrorType attributeError;
                case attribute_1Choice_1:
                        AttributeType attribute_1;
};

typedef sequence <SetInfoStatusType>
        SetListErrorSetInfoListType;

struct SetListErrorType {
        ObjectClassTypeOpt managedObjectClass;
```

```
        ObjectInstanceTypeOpt managedObjectInstance;
        ASN1_GeneralizedTimeOpt currentTime;
        SetListErrorSetInfoListType setInfoList;
};

struct SpecificErrorInfoType {
        ASN1_ObjectIdentifier errorId;
        ASN1_DefinedAny errorInfo;  // defined by:errorId
};

struct ProcessingFailureType {
        ObjectClassType managedObjectClass;
        ObjectInstanceTypeOpt managedObjectInstance;
        SpecificErrorInfoType specificErrorInfo;
};

enum LinkedReplyArgumentTypeChoice { getResultChoice,
        getListErrorChoice, setResultChoice, setListErrorChoice,
        actionResultChoice, processingFailureChoice,
        deleteResultChoice, actionErrorChoice, deleteErrorChoice };

union LinkedReplyArgumentType
        switch(LinkedReplyArgumentTypeChoice) {
                case getResultChoice:
                        GetResultType getResult;
                case getListErrorChoice:
                        GetListErrorType getListError;
                case setResultChoice:
                        SetResultType setResult;
                case setListErrorChoice:
                        SetListErrorType setListError;
                case actionResultChoice:
                        ActionResultType actionResult;
                case processingFailureChoice:
                        ProcessingFailureType processingFailure;
                case deleteResultChoice:
                        DeleteResultType deleteResult;
                case actionErrorChoice:
                        ActionErrorType actionError;
                case deleteErrorChoice:
                        DeleteErrorType deleteError;
 };

const ModifyOperatorType replace =
        0;

const ModifyOperatorType addValues =
        1;

const ModifyOperatorType removeValues =
        2;

const ModifyOperatorType setToDefault =
        3;
```

```
struct NoSuchActionType {
      ObjectClassType managedObjectClass;
      ActionTypeIdType actionType;
};

struct NoSuchEventTypeType {
       ObjectClassType managedObjectClass;
      EventTypeIdType eventType;
};

const ScopeLevelType firstLevelOnly =
      1;

const ScopeLevelType wholeSubtree =
      2;

typedef ModifyOperatorTypeOpt ModifyOperatorTypeDef;

struct SetArgumentModificationListItemType {
      ModifyOperatorTypeDef modifyOperator;
      AttributeIdType attributeId;
      ASN1_DefinedAnyOpt attributeValue;  // defined by:attributeId
};

typedef sequence <SetArgumentModificationListItemType>
      SetArgumentModificationListType;

struct SetArgumentType {
      ObjectClassType baseManagedObjectClass;
      ObjectInstanceType baseManagedObjectInstance;
      AccessControlTypeOpt accessControl;
      CMISSyncTypeDef synchronization;
      ScopeTypeDef scope;
      CMISFilterTypeDef filter;
      SetArgumentModificationListType modificationList;
};

const ModifyOperatorType modifyOperatorDefault =
      replace;

typedef sequence <AttributeIdType>
      MissingAttributeValueType;

// Complex constants declaration.

interface ConstValues {

 // ** Generated values for <ActionArgumentType::synchronization>:

 CMISSyncType synchronizationDefault();
 // returns: bestEffort

 // ** Generated values for <ActionArgumentType::scope>:

 ScopeType scopeDefault();
```

```
        // returns: baseObject

        // ** Generated values for <ActionArgumentType::filter>:

        CMISFilterType filterDefault();
        // returns: {}

        // ** Generated values for <DeleteArgumentType::synchronization>:

        CMISSyncType synchronizationDefault_1();
        // returns: bestEffort

        // ** Generated values for <DeleteArgumentType::scope>:

        ScopeType scopeDefault_1();
        // returns: baseObject

        // ** Generated values for <DeleteArgumentType::filter>:

        CMISFilterType filterDefault_1();
        // returns: {}

        // ** Generated values for <GetArgumentType::synchronization>:

        CMISSyncType synchronizationDefault_2();
        // returns: bestEffort

        // ** Generated values for <GetArgumentType::scope>:

        ScopeType scopeDefault_2();
        // returns: baseObject

        // ** Generated values for <GetArgumentType::filter>:

        CMISFilterType filterDefault_2();
        // returns: {}

        // ** Generated values for <SetArgumentType::synchronization>:

        CMISSyncType synchronizationDefault_3();
        // returns: bestEffort

        // ** Generated values for <SetArgumentType::scope>:

        ScopeType scopeDefault_3();
        // returns: baseObject

        // ** Generated values for <SetArgumentType::filter>:

        CMISFilterType filterDefault_3();
        // returns: {}
      };

};
```

```
#endif  /* _X711CMI_IDL_ */
```

B.4 Optional IDL

IDL listed in this section is part of this specification, however, it is part of some optional facility, and therefore not required from any implementation.

B.4.1 ASN1.idl

Specification of the Dynamic ASN1 Any API.

```
// File: ASN1.idl
#ifndef _ASN1_IDL_
#define _ASN1_IDL_

#include <orb.idl>
#include <ASN1Types.idl>

#pragma prefix “jidm.org”

module ASN1 {

    typedef CORBA::Identifier Identifier;

    enum Kind {
        ak_none, // used when value is not ASN.1 based
        ak_null, ak_boolean,
        ak_integer, ak_real,
        ak_numericstring, ak_printablestring,
        ak_visiblestring, ak_iso646string,
        ak_graphicstring, ak_objectdescriptor,
        ak_teletexstring, ak_t61string,
        ak_generalizedtime, ak_utctime,
        ak_octetstring, ak_generalstring,
        ak_ia5string, ak_videotexstring,
        ak_bmpstring, ak_universalstring,
        ak_objectidentifier,
        ak_bitstring,
        ak_any, ak_definedany,
        ak_external,
        ak_enum,
        ak_sequence, ak_set,
        ak_sequenceof, ak_setof,
        ak_choice
    };

    interface DynAny : CORBA::DynAny {
        Kind asn1_kind() raises (Invalid);
        Identifier asn1_type_name () raises (Invalid);
        Identifier asn1_module_name() raises (Invalid);
        Identifier asn1_module_nickname() raises (Invalid);
        ASN1_ObjectIdentifier asn1_module_oid() raises (Invalid);
```

```
void asn1_assign (in DynAny asn1_dyn_any) raises (Invalid);
void from_dyn_any (in CORBA::DynAny dyn_any) raises (Invalid);
CORBA::DynAny to_dyn_any() raises (Invalid);
DynAny asn1_copy();

void insert_asn1_null(in ASN1_Null value) raises(InvalidValue);
void insert_asn1_boolean(in ASN1_Boolean value) raises(InvalidValue);
void insert_asn1_unsigned16(in ASN1_Unsigned16 value) raises(InvalidValue);
void insert_asn1_unsigned(in ASN1_Unsigned value) raises(InvalidValue);
void insert_asn1_unsigned64(in ASN1_Unsigned64 value) raises(InvalidValue);
void insert_asn1_integer16(in ASN1_Integer16 value) raises(InvalidValue);
void insert_asn1_integer(in ASN1_Integer value) raises(InvalidValue);
void insert_asn1_integer64(in ASN1_Integer64 value) raises(InvalidValue);
void insert_asn1_real(in ASN1_Real value) raises(InvalidValue);
void insert_asn1_numericstring(in ASN1_NumericString value) raises(InvalidValue);
void insert_asn1_printablestring(in ASN1_PrintableString value) raises(InvalidValue);
void insert_asn1_visiblestring(in ASN1_VisibleString value) raises(InvalidValue);
void insert_asn1_iso646string(in ASN1_ISO646String value) raises(InvalidValue);
void insert_asn1_graphicstring(in ASN1_GraphicString value) raises(InvalidValue);
void insert_asn1_objectdescriptor(in ASN1_ObjectDescriptor value) raises(InvalidValue);
void insert_asn1_teletexstring(in ASN1_TeletexString value) raises(InvalidValue);
void insert_asn1_t61string(in ASN1_T61String value) raises(InvalidValue);

void insert_asn1_generalizedtime(in ASN1_GeneralizedTime value) raises(InvalidValue);
void insert_asn1_utctime(in ASN1_UTCTime value) raises(InvalidValue);

void insert_asn1_octetstring(in ASN1_OctetString value) raises(InvalidValue);
void insert_asn1_generalstring(in ASN1_GeneralString value) raises(InvalidValue);
void insert_asn1_ia5string(in ASN1_IA5String value) raises(InvalidValue);
void insert_asn1_videotexstring(in ASN1_VideotexString value) raises(InvalidValue);

void insert_asn1_bmpstring(in ASN1_BMPString value) raises(InvalidValue);
void insert_asn1_universalstring(in ASN1_UniversalString value) raises(InvalidValue);

void insert_asn1_objectidentifier(in ASN1_ObjectIdentifier value) raises(InvalidValue);

void insert_asn1_bitstring(in ASN1_BitString value) raises(InvalidValue);

void insert_asn1_any(in ASN1_Any value) raises(InvalidValue);
void insert_asn1_definedany(in ASN1_DefinedAny value) raises(InvalidValue);

void insert_asn1_external(in ASN1_External value) raises(InvalidValue);

ASN1_Null get_asn1_null() raises(TypeMismatch);
ASN1_Boolean get_asn1_boolean() raises(TypeMismatch);

ASN1_Unsigned16 get_asn1_unsigned16() raises(TypeMismatch);
ASN1_Unsigned get_asn1_unsigned() raises(TypeMismatch);
ASN1_Unsigned64 get_asn1_unsigned64() raises(TypeMismatch);
ASN1_Integer16 get_asn1_integer16() raises(TypeMismatch);
ASN1_Integer get_asn1_integer() raises(TypeMismatch);
ASN1_Integer64 get_asn1_integer64() raises(TypeMismatch);

ASN1_Real get_asn1_real() raises(TypeMismatch);
```

```
        ASN1_NumericString get_asn1_numericstring() raises(TypeMismatch);
        ASN1_PrintableString get_asn1_printablestring() raises(TypeMismatch);
        ASN1_VisibleString get_asn1_visiblestring() raises(TypeMismatch);
        ASN1_ISO646String get_asn1_iso646string() raises(TypeMismatch);
        ASN1_GraphicString get_asn1_graphicstring() raises(TypeMismatch);
        ASN1_ObjectDescriptor get_asn1_objectdescriptor() raises(TypeMismatch);
        ASN1_TeletexString get_asn1_teletexstring() raises(TypeMismatch);
        ASN1_T61String get_asn1_t61string() raises(TypeMismatch);

        ASN1_GeneralizedTime get_asn1_generalizedtime() raises(TypeMismatch);
        ASN1_UTCTime get_asn1_utctime() raises(TypeMismatch);

        ASN1_OctetString get_asn1_octetstring() raises(TypeMismatch);
        ASN1_GeneralString get_asn1_generalstring() raises(TypeMismatch);
        ASN1_IA5String get_asn1_ia5string() raises(TypeMismatch);
        ASN1_VideotexString get_asn1_videotexstring() raises(TypeMismatch);

        ASN1_BMPString get_asn1_bmpstring() raises(TypeMismatch);
        ASN1_UniversalString get_asn1_universalstring() raises(TypeMismatch);

        ASN1_ObjectIdentifier get_asn1_objectidentifier() raises(TypeMismatch);

        ASN1_BitString get_asn1_bitstring() raises(TypeMismatch);

        ASN1_Any get_asn1_any() raises(TypeMismatch);
        ASN1_DefinedAny get_asn1_definedany() raises(TypeMismatch);

        ASN1_External get_asn1_external() raises(TypeMismatch);

        DynAny current_asn1_component () raises(Invalid);
    };

    interface DynEnum: DynAny, CORBA::DynEnum {
        attribute string value_as_asn1_identifier;
        attribute long value_as_asn1_value;
    };

    interface DynNamedNumber: DynAny {
        attribute string value_as_asn1_identifier;
    };

    typedef CORBA::FieldName FieldName;
    typedef CORBA::NameValuePairSeq NameValuePairSeq;

    interface DynSetSeq: DynAny, CORBA::DynStruct {
        FieldName current_asn1_elem_name ();
        Kind current_asn1_elem_kind ();
        NameValuePairSeq get_asn1_elems() raises(Invalid);
        void set_asn1_elems(in NameValuePairSeq value) raises (InvalidSeq);
        void insert_optional_absent() raises (InvalidValue);
        DynAny insert_optional_present() raises (InvalidValue);
        void insert_default_absent() raises (InvalidValue);
        DynAny insert_default_present() raises (InvalidValue);
        boolean get_optional_presence() raises (TypeMismatch);
        DynAny get_optional_present() raises (TypeMismatch);
```

```
        boolean get_default_presence() raises (TypeMismatch);
        DynAny get_default_present() raises (TypeMismatch);
};

interface DynChoice: DynAny, CORBA::DynUnion {
        DynAny asn1_elem ();
        attribute FieldName asn1_elem_name;
        Kind asn1_elem_kind ();
};

interface DynSetSeqOf : DynAny, CORBA::DynSequence {
        Kind asn1_item_kind ();
};


interface DynAnyFactory {
        exception InconsistentKind {};
        exception InconsistentTypeCode {};

        typedef CORBA::Identifier Identifier;

        DynAny create_asn1_dyn_any(in any value);

        DynAny create_basic_dyn_any(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynStruct create_dyn_struct(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynSequence create_dyn_sequence(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynUnion create_dyn_union(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynEnum create_dyn_enum(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynArray create_dyn_array(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);
        CORBA::DynFixed create_dyn_fixed(in CORBA::TypeCode type)
                raises(InconsistentTypeCode);

        DynAny create_asn1_dyn_primitive(in Identifier asn1_nickname,
                        in Identifier asn1_name)
                raises(InconsistentKind);
        DynEnum create_asn1_dyn_enum(in Identifier asn1_nickname,
                        in Identifier asn1_name)
                raises(InconsistentKind);
        DynSetSeq create_asn1_dyn_setseq(in Identifier asn1_nickname,
                        in Identifier asn1_name)
                raises(InconsistentKind);
         DynSetSeqOf create_asn1_dyn_setseqof(in Identifier asn1_nickname,
                        in Identifier asn1_name)
                raises(InconsistentKind);
        DynChoice create_asn1_dyn_choice(in Identifier asn1_nickname,
                        in Identifier asn1_name)
                raises(InconsistentKind);
};
```

```
};

#endif /* _ASN1_IDL_ */
```

B.4.2 OSICaching.idl

Specification of the OSI Caching facility.

```
// File: OSICaching.idl
#ifndef _OSICACHING_IDL_
#define _OSICACHING_IDL_

#include <OSIMgmt.idl>

#pragma prefix "jidm.org"

module OSICaching {
    typedef unsigned long ExpirationInterval; // in seconds
    typedef ASN1_ObjectIdentifier ManagedObjectClass;
    typedef sequence <ManagedObjectClass> ManagedObjectClassSeq;
    typedef ASN1_ObjectIdentifier AttrId;
    typedef sequence < ASN1_ObjectIdentifier > AttrIdSeq;

    // NoSuchAttributes is raised when any specified attribute identifiers
    // are either unknown or invalid.
    exception NoSuchAttributes {
        AttrIdSeq unknown_attributes;
    };

    // AttributesNotCached is raised when any specified attribute identifiers
    // to relevant caching operations are not being cached.
    exception AttributesNotCached {
        AttrIdSeq attr_id_list;
    };

    // NoSuchObjectClasses is raised when any specified object classes are
    // either unknown or invalid.
    exception NoSuchObjectClasses {
        ManagedObjectClassSeq unknown_mocs;
    };

    // ObjectClassesNotCached is raised when any specified object classes
    // to relevant caching operations are not being cached.
    exception ObjectClassesNotCached {
        ManagedObjectClassSeq moc_list;
    };

    // InvalidObjectClassAttributesPairs is raised when any specified attribute
    // identifiers do not belong to the specified managed object class.
    struct ObjectClassAttributesPair {
        ManagedObjectClass moc;
        AttrIdSeq attr_id_list;
    };
    typedef sequence<ObjectClassAttributesPair> ObjectClassAttributesPairSeq;
```

```
exception InvalidObjectClassAttributesPairs {
      ObjectClassAttributesPairSeq invalid_pairs;
};

/* There may be situations when more than one type of error may occur
 * because of a single invocation of an operation. To accurately convey
 * the different types of error information, CacheConfigException is used
 * by some operations. If any of the members of the following exception
 * are not relevant, then such members shall be empty sequences, i.e.,
 * sequences of zero length. For example, when passing an argument of
 * AttrIdSeq to remove cached attributes , the client may pass some invalid
 * or unkown attribute identifiers, and some valid attribute identifiers
 * that are not cached. In such situations, CacheConfigException is raised
 * with the invalid or unknown attribute identifiers specified in the
 * no_such_attributes member, the valid but not cached attribute
 * identifiers specified in the attrs_not_cached member, and the rest of
 * the members set to zero length sequences.
 */
exception CacheConfigException {
      AttrIdSeq no_such_attributes;
      ManagedObjectClassSeq no_such_classes;
      AttrIdSeq attrs_not_cached;
      ManagedObjectClassSeq mocs_not_cached;
      ObjectClassAttributesPairSeq invalid_moc_attrs_pairs;
};

// abstract interface for configuring all caches
interface CacheConfigurator {
      void set_default_expiration_interval (
            in ExpirationInterval expiration_interval,
            in boolean override_specific_settings
      );
      ExpirationInterval get_default_expiration_interval ();

      void set_caching_enabled (
            in boolean enabled,
            in boolean override_specific_settings
      );
      boolean is_caching_enabled ();
};

// cached attribute information
struct CachedAttribute {
      AttrId attr_id;
      ExpirationInterval expiration_interval;
};
typedef sequence < CachedAttribute > CachedAttributeSeq;

// abstract interface to configure per-attribute cache
interface PerAttributeCacheConfigurator {
      void add_cached_attributes (
            in CachedAttributeSeq attr_list,
            in boolean override_specific_settings
      ) raises ( NoSuchAttributes );
```

```
        void remove_cached_attributes (
            in AttrIdSeq attr_id_list,
            in boolean override_specific_settings
        ) raises ( CacheConfigException );

        CachedAttributeSeq get_cached_attributes ();

        ExpirationInterval get_expiration_interval (
            in AttrId attr_id
        ) raises ( CacheConfigException );

        void set_expiration_interval(
            in AttrIdSeq attr_id_list,
            in ExpirationInterval interval
        ) raises ( CacheConfigException );
};

// managed object class with indicated attributes cached
struct CachedObjectClass {
        ManagedObjectClass moc;
        CachedAttributeSeq cached_attributes_list;
};
typedef sequence < CachedObjectClass > CachedObjectClassSeq;

// abstract interface to configure per-class cache
interface PerClassCacheConfigurator {
        void add_cached_classes (
            in CachedObjectClassSeq class_list,
            in boolean override_specific_settings
        ) raises ( CacheConfigException );

        void remove_cached_classes (
            in ManagedObjectClassSeq moc_list,
            in boolean override_specific_settings
        ) raises ( CacheConfigException );

        void remove_cached_attributes_from_class_cache(
            in ManagedObjectClass moc,
            in AttrIdSeq attr_id_list,
            in boolean override_specific_settings
        ) raises ( CacheConfigException );

        CachedObjectClassSeq get_cached_classes ();

        CachedAttributeSeq get_cached_attributes_for_class (
            in ManagedObjectClass moc
        ) raises ( OSIMgmt::NoSuchObjectClass );

        void set_expiration_interval_for_class (
            in ManagedObjectClass moc,
            in AttrIdSeq attr_list,
            in ExpirationInterval extension_duration
        ) raises ( CacheConfigException );
};
```

```
    interface ProxyAgent : OSIMgmt::ProxyAgent,
        CacheConfigurator,
        PerAttributeCacheConfigurator,
        PerClassCacheConfigurator {};

    interface ManagedObject : OSIMgmt::ManagedObject,
            CacheConfigurator,
            PerAttributeCacheConfigurator {

        void refresh_cached_values (
            in AttrIdSeq attr_list
        ) raises ( CacheConfigException );

        void invalidate_cached_values (
            in AttrIdSeq attr_list
        ) raises ( CacheConfigException );
    };

};

#endif /* _OSICACHING_IDL_ */
```

B.4.3 OSITracking.idl

Specification of the OSI Tracking facility.

```
// File: OSITracking.idl
#ifndef _OSITRACKING_IDL_
#define _OSITRACKING_IDL_

#include <OSICaching.idl>

#pragma prefix “jidm.org”

module OSITracking {

    typedef OSICaching::ManagedObjectClassSeq ManagedObjectClassSeq;
    typedef OSICaching::AttrIdSeq AttrIdSeq;

    // abstract interface to configure all tracking
    interface TrackConfigurator {
        void set_tracking_enabled (
            in boolean enabled,
            in boolean override_specific_settings
        );

        boolean is_tracking_enabled ();
    };

    // abstract interface to configure per-attribute tracking
    interface PerAttributeTrackConfigurator {
        void add_tracked_attributes (
            in AttrIdSeq attr_list,
            in boolean override_specific_settings
```

```
        ) raises ( OSICaching::NoSuchAttributes );

        // If the attr_id_list contains an attribute identifier that is not
        // being tracked, then that attribute identifier is ignored
        // by remove_tracked_attributes.
        void remove_tracked_attributes (
            in AttrIdSeq attr_id_list,
            in boolean override_specific_settings
        ) raises ( OSICaching::NoSuchAttributes );

        AttrIdSeq get_tracked_attributes ();
};

// managed object class with indicated attributes tracked
struct TrackedObjectClass {
        OSICaching::ManagedObjectClass moc;
        AttrIdSeq list_of_tracked_attributes;
};

typedef sequence < TrackedObjectClass > TrackedObjectClassSeq;

// TrackConfigException is similar in purpose to
// OSICaching::CacheConfigException
exception TrackConfigException {
        ManagedObjectClassSeq no_such_mocs;
        AttrIdSeq no_such_attr_ids;
        OSICaching::ObjectClassAttributesPairSeq invalid_moc_attrs_pairs;
};

// abstract interface to configure per-class tracking
interface PerClassTrackConfigurator {
        void add_tracked_classes (
            in TrackedObjectClassSeq class_list,
            in boolean override_specific_settings
        ) raises ( TrackConfigException );

        void remove_tracked_classes (
            in ManagedObjectClassSeq moc_list,
            in boolean override_specific_settings
        ) raises ( OSICaching::NoSuchObjectClasses );

        TrackedObjectClassSeq get_tracked_classes ();

        AttrIdSeq get_tracked_attributes_for_class (
            in OSICaching::ManagedObjectClass class_name
        ) raises ( OSIMgmt::NoSuchObjectClass );
};

interface ProxyAgent : OSICaching::ProxyAgent,
        TrackConfigurator,
        PerAttributeTrackConfigurator,
        PerClassTrackConfigurator {};

interface ManagedObject : OSICaching::ManagedObject,
            TrackConfigurator,
```

```
            PerAttributeTrackConfigurator {};

};

#endif /* _OSITRACKING_IDL_ */
```

B.4.4 OSICollection.idl

Specification of the OSI Collection facility.

```
// File: OSICollection.idl
#ifndef _OSICOLLECTION_IDL_
#define _OSICOLLECTION_IDL_

#include <OSIMgmt.idl>

#pragma prefix "jidm.org"

module OSICollection {
    typedef OSIMgmt::ManagedObject ManagedObject;
    typedef sequence < ManagedObject > ManagedObjectSeq;
    exception IteratorInvalid { };
    exception IteratorInBetween { };
    exception CollectionInvalid { };
    exception NotFound { };
    exception InvalidName { };

    interface Iterator {
        // retrieving elements
        boolean  get_element (
            out ManagedObject mo
        ) raises ( IteratorInvalid, IteratorInBetween );
        boolean  get_n_elements (
            in unsigned long how_many,
            out ManagedObjectSeq mo_list
        ) raises ( IteratorInvalid );

        // moving iterator
        void restart () raises ( IteratorInvalid );
        void set_to_next_element () raises ( IteratorInvalid );
        void set_to_next_nth_element (
            in unsigned long how_many
        ) raises ( IteratorInvalid );

        // iterator state
        void invalidate ();
        boolean is_valid ();
        boolean is_in_between ();
        boolean is_equal ( in Iterator other ) raises ( IteratorInvalid );

        // cloning, assigning and destroying
        Iterator clone ();
        void assign ( in Iterator from_where ) raises ( IteratorInvalid );
        void destroy ();
```

```
};

typedef OSIMgmt::LinkedReplyHandler LinkedReplyHandler;
typedef OSIMgmt::EndOfRepliesHandler EndOfRepliesHandler;

// abstract base interface
interface BaseCollection {
    // operations to perform on all elements in the collection
    void perform_get (
        in OSIMgmt::ASN1_ObjectIdentifierSeq attr_id_list,
      in LinkedReplyHandler lrh,
      in EndOfRepliesHandler eorh
    );
    void perform_set (
        in OSIMgmt::SetOperationArgument modif_list,
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );
void perform_action (
        in ASN1_ObjectIdentifier action_id,
        in ASN1_DefinedAny action_info,
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );
    void perform_delete (
        in LinkedReplyHandler lrh,
        in EndOfRepliesHandler eorh
    );

    // statistics
    boolean is_empty ();

    // creating iterators
    Iterator create_iterator (
        in boolean read_only
    ) raises ( CollectionInvalid );

    // destruction
    void destroy ();
};

interface EnumCollection : BaseCollection {
    // adding elements
    void add_element ( in ManagedObject element );
    void add_elements ( in ManagedObjectSeq elem_list );
    void add_all_from ( in BaseCollection collection );

    // removing elements
    void remove_element_at (
        in Iterator where
    ) raises ( IteratorInvalid, IteratorInBetween );
    void remove_all ();
};

interface RuleCollection : BaseCollection {
```

```
            ManagedObject get_base_object () raises ( CollectionInvalid );
            X711CMI::ScopeType get_scope () raises ( CollectionInvalid );
            X711CMI::CMISFilterType get_filter () raises ( CollectionInvalid );
            X711CMI::CMISSyncType get_synchronization () raises ( CollectionInvalid );
        };

        interface CollectionFactory {
            EnumCollection create_enum_collection ();

            EnumCollection create_enum_collection_from_collection (
              in BaseCollection collection
            );

            RuleCollection create_rule_collection (
                in OSIMgmt::ManagedObject base_managed_object,
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType sync
            );

            RuleCollection create_rule_collection_by_name (
                in OSIMgmt::ProxyAgent proxy_agent,
                in CORBA::ScopedName base_mo_interface,
                in CosNaming::Name base_mo_name,
                in X711CMI::ScopeType scope,
                in X711CMI::CMISFilterType filter,
                in X711CMI::CMISSyncType sync
            );
        };
};

#endif /* _OSICOLLECTION_IDL_ */
```

B.4.5 SNMPMIR.idl

Specification of the SNMP Management Information Repository facility.

```
// File: SNMPMIR.idl
#ifndef _SNMPMIR_IDL_
#define _SNMPMIR_IDL_

#include <orb.idl>
#include <ASN1Types.idl>

#pragma prefix “jidm.org”

module SNMPMIR {

    // Snmpv1GenericTrapId defines the identfiers for generic trap
    // types in SNMPv1.

    enum Snmpv1GenericTrapId {
        TRAP_COLDSTART, TRAP_WARMSTART, TRAP_LINKDOWN, TRAP_LINKUP,
        TRAP_AUTHFAIL, TRAP_EGPNEIGHBORLOSS, TRAP_ENTERPRISESPECIFIC
```

```
};

// GENERIC_TRAP_ENTERPRISE_OID defines the enterprise OID for
// generic traps.

const ASN1_ObjectIdentifier GENERIC_TRAP_ENTERPRISE_OID = “1.3.6.1.4.1.3.1.1”;

// SmiAccessMode defines the enumerated values of the SMI based
// acces - mode defined for a specific variables.

enum SmiAccessMode {
     read_only, read_write, read_create, write_only, inaccessible
};

// Basic and Application specific SMI types.
enum SmiValueType {
     smi_null_value, smi_integer_value, smi_string_value, smi_objectID_value,
     smi_bit_value, smi_ipAddress_value, smi_counter_value, smi_gauge_value,
     smi_timeticks_value, smi_arbitary_value, smi_nsapAddress_value,
     smi_big_counter_value, smi_unsigned_integer_value, smi_unknown_type
};

typedef CORBA::ScopedName ScopedName;
typedef sequence < ScopedName > ScopedNameList;
typedef sequence < string > VarNameList;

typedef sequence < string > ModuleNameList;
typedef sequence < ASN1_ObjectIdentifier > OIDList;

interface OidRepository {

     ScopedName get_scoped_name ( in ASN1_ObjectIdentifier in_oid );

     string get_name ( in ASN1_ObjectIdentifier in_oid );
     ASN1_ObjectIdentifier get_oid ( in ScopedName in_scoped_name );
     ASN1_ObjectIdentifier get_var_oid (
        in ScopedName iface_scoped_name,
           in string var_name
     );

     string get_textual_obj_id ( in ASN1_ObjectIdentifier obj_id );

     void split_var_object_id (
          in ASN1_ObjectIdentifier  var_obj_id,
          out ASN1_ObjectIdentifier var_oid,
          out ASN1_ObjectIdentifier obj_index
     );

     ASN1_ObjectIdentifier get_next_oid ( in ASN1_ObjectIdentifier oid );

     ScopedName get_next_scoped_name ( in ScopedName scoped_name );
     ScopedName get_next_entry_type ( in ScopedName scoped_name );

};
```

```
        interface VariableDef : CORBA::AttributeDef {
            readonly attribute ASN1_ObjectIdentifier oid;
            readonly attribute SmiValueType smi_type;
            readonly attribute SmiAccessMode smi_access_mode;

            readonly attribute any default_value;
        };
        typedef sequence < VariableDef > VariableDefList;

        interface SmiEntryDef : CORBA::InterfaceDef {
            readonly attribute ASN1_ObjectIdentifier oid;
            readonly attribute unsigned long        total_no_of_variables;
            readonly attribute VariableDefList      var_def_list;
            readonly attribute VarNameList          var_name_list;
            readonly attribute ScopedNameList       var_scoped_name_list;
            readonly attribute OIDList              var_oid_list;
            readonly attribute VarNameList          index_var_names;

            readonly attribute ScopedName           next_group_or_table;
            VariableDef lookup_variable( in string var_name );
        };
        typedef sequence < SmiEntryDef > SmiEntryDefList;

        interface GroupDef : SmiEntryDef {
            readonly attribute SmiEntryDefList table_entry_list;
        };
        typedef sequence < GroupDef >  GroupDefList;

        interface ModuleDef : CORBA::ModuleDef {
            readonly attribute GroupDefList smi_group_def_list;
            readonly attribute SmiEntryDefList smi_entry_def_list;
            readonly attribute CORBA::InterfaceDef  push_notification_def;
            readonly attribute CORBA::InterfaceDef  pull_notification_def;

            readonly attribute CORBA::InterfaceDef  default_value_def;
            SmiEntryDef lookup_smi_entry( in string smi_entry_name );
        };
        typedef sequence < ModuleDef >  ModuleDefList;

        interface Repository : CORBA::Repository, OidRepository {
            readonly attribute ModuleNameList module_name_list;
            readonly attribute ModuleDefList  module_def_list;
            boolean is_smi_module( in CORBA::Identifier module_name );
            ModuleDef lookup_smi_module( in string a_module_name );
            SmiEntryDef lookup_smi_entry( in ScopedName entry_scoped_name );
            ScopedNameList get_entry_var_list( in ScopedName entry_scoped_name );
            ScopedNameList get_entry_index_var_list( in ScopedName entry_scoped_name );
            any get_var_default_value( in ScopedName var_scoped_name );
            string get_generic_trap_desc( in ASN1_Integer trap_type );
        };

};

#endif /* _SNMPMIR_IDL_ */
```

Conformance Statement C

C.1 Conformance Statement

This section presents the different conformance points available for this specification.

C.1.1 General Conformance Requirements

All implementations claiming conformance to this specification will

- provide a complete implementation of an interface specification (mandatory or otherwise) for which conformance is claimed unless some part of the interface specification is identified as optional, and
- conform to the mappings of GDMO, ASN.1 and/or SNMP SMI to IDL as specified in XoJIDM ST XoJIDM (see "[XoJIDM] Inter Domain Management: Specification Translation" mentioned in Appendix A), and amended by the JIDM ST Issues lists resolutions where support of an information model specified in GDMO, ASN.1, or SNMP SMI respectively is also claimed.

C.1.2 Specific Conformance Requirements

An implementation can claim conformance to this specification at four *conformance points* named, respectively:

- JIDM Facilities
- CMISE Access Facilities
- OSI Management Facilities
- SNMP Management Facilities

in either the manager role, the agent role, or both.

C.1.3 JIDM Conformance Point

Manager Role

Implementations claiming conformance to JIDM Facilities in the manager role:

- Use those interfaces specified in the JIDM module that are required to perform its management functionality, namely:
  - **JIDM::ProxyAgentFinder**
  - **JIDM::ProxyAgent**
- If the management model being supported defines one or more managed object interfaces, use those interfaces required to perform the management functionality; additionally, use the following interfaces, if needed to perform its management function:
  - **CosNaming::NamingContext**
  - **CosLifeCycle::FactoryFinder**
  - **CosLifeCycle::GenericFactory**
- Implement the interface and behaviors specified for the **JIDM::ProxyAgentController** object, if required.
- Implement the **JIDM::EventPortFinder** interface, providing access to **CosEventChannelAdmin::SupplierAdmin** objects in the manager domain, if events are to be received by the manager.

Agent Role

Implementations claiming conformance to JIDM Facilities in the agent role:

- Provide implementations of the following interfaces:
  - **JIDM::ProxyAgentFinder**
  - **JIDM::ProxyAgent**
- If the management model being supported defines one or more managed object interfaces, implement those interfaces being supported by the agent. Name resolution of objects exposing those interfaces will be supported by providing an implementation of the **CosNaming::NamingContext** interface. If creation of managed objects is supported by the agent, then implementations of the **CosLifeCycle::FactoryFinder** and **CosLifeCycle::GenericFactory** interfaces are provided.
- Execute the client behavior of the **JIDM::ProxyAgentController** interface, if requested by a manager.
- Execute the client behavior of the **JIDM::EventPortFinder** and supply events to the corresponding **CosEventChannelAdmin::SupplierAdmin** interface, if the agent implementation is capable of emitting event reports.

C.1.4 CMISE Access Conformance Point

Manager Role

Implementations claiming conformance to CMISE Access Facilities in the manager role:

- Use those interfaces specified in the JIDM and OSIMgmt modules that are required to perform its management function:
  - **JIDM::ProxyAgentFinder**
  - **OSIMgmt::ProxyAgent**
- Implement the interface and behavior specified for the **JIDM::ProxyAgentController** object, if required.
- Implement the interface and behavior specified for the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** objects.
- Implement the **JIDM::EventPortFinder** interface, providing access to **CosEventChannelAdmin::SupplierAdmin** objects in the manager domain, if events are to be received by the manager.

Agent Role

Implementations claiming conformance to CMISE Access Facilities in the agent role:

- Provide implementations of the following interfaces:
  - **JIDM::ProxyAgentFinder**
  - **OSIMgmt::ProxyAgent**
- Execute the client behavior of the **JIDM::ProxyAgentController** interface, if requested by a manager.
- Execute the client behavior of the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** interfaces.
- Execute the client behavior of the **JIDM::EventPortFinder** and supply events to the corresponding **CosEventChannelAdmin::SupplierAdmin** interface, if the agent implementation is capable of emitting event reports.

C.1.5 OSI Management Conformance Point

It is the intent of the OSI Management Conformance Points to ensure that a conformant manager role implementation interoperates with a conformant agent role implementation.

Manager Role

Implementations claiming conformance to OSI Management Facilities in the manager role:

- Use those interfaces specified in the JIDM and OSIMgmt modules that are required to perform its management function, namely:
  - **JIDM::ProxyAgentFinder**
  - **OSIMgmt::ProxyAgent**
  - **OSIMgmt::NamingContext**
  - **CosLifeCycle::FactoryFinder**
  - **CosLifeCycle::GenericFactory**
  - **OSIMgmt::ManagedObjectFactory**
  - **OSIMgmt::ManagedObject**
- Use whatever managed object interface(s) specific to an information model are required to perform its management function.
- Implement the interface and behavior specified for the **JIDM::ProxyAgentController** object, if required.
- Implement the interface and behavior specified for the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** objects, if required.
- Implement the **JIDM::EventPortFinder** interface, providing access to **CosEventChannelAdmin::SupplierAdmin** objects in the manager domain, if events are to be received by the manager.

Agent Role

Implementations claiming conformance to OSI Management Facilities in the agent role:

- Provide implementations of the following interfaces:
  - **JIDM::ProxyAgentFinder**
  - **OSIMgmt::ProxyAgent**
  - **OSIMgmt::NamingContext**
  - **CosLifeCycle::FactoryFinder**
  - **OSIMgmt::ManagedObject**
- Implement the managed object interface(s) specific to the information model being supported by the agent.
- Provide implementations of the following additional interfaces, if the agent is capable of creating objects as a result of management operations:
  - **CosLifeCycle::GenericFactory**
  - **OSIMgmt::ManagedObjectFactory**
- Execute the client behavior of the **JIDM::ProxyAgentController** interface, if requested by a manager.
- Execute the client behavior of the **OSIMgmt::LinkedReplyHandler** and **OSIMgmt::EndOfRepliesHandler** interfaces, if requested by a manager.
- Execute the client behavior of the **JIDM::EventPortFinder** and supply events to the corresponding **CosEventChannelAdmin::SupplierAdmin** interface, if the agent implementation is capable of emitting event reports.

C.1.6 SNMP Management Conformance Point

It is the intent of the SNMP Management Conformance Points to ensure that a conformant manager role implementation interoperates with a conformant agent role implementation.

Manager Role

Implementations claiming conformance to SNMP Management Facilities in the manager role:

- Use those interfaces specified in the JIDM and SNMPMgmt modules that are required to perform its management function, namely:
  - **JIDM::ProxyAgentFinder**
  - **SNMPMgmt::ProxyAgent**
  - **SNMPMgmt::NamingContext**
  - **CosLifeCycle::FactoryFinder**
  - **SNMPMgmt::GenericFactory**
  - **SNMPMgmt::SMIEntry**
- Use whatever managed object interface(s) specific to an information model are required to perform its management function.
- Implement the interface and behavior specified for the **JIDM::ProxyAgentController** object, if required.
- Implement the **JIDM::EventPortFinder** interface, providing access to **CosEventChannelAdmin::SupplierAdmin** objects in the manager domain, if events are to be received by the manager.

Agent Role

Implementations claiming conformance to SNMP Management Facilities in the agent role:

- Provide implementations of the following interfaces:
  - **JIDM::ProxyAgentFinder**
  - **SNMPMgmt::ProxyAgent**
  - **SNMPMgmt::NamingContext**
  - **CosLifeCycle::FactoryFinder**
  - **SNMPMgmt::SMIEntry**
- Implement the managed object interface(s) specific to the information model being supported by the agent.
- Provide implementations of the following additional interfaces, if the agent is capable of creating objects as a result of management operations:
  - **SNMPMgmt::GenericFactory**
- Execute the client behavior of the **JIDM::ProxyAgentController** interface, if requested by a manager.

- Execute the client behavior of the **JIDM::EventPortFinder** and supply events to the corresponding **CosEventChannelAdmin::SupplierAdmin** interface, if the agent implementation is capable of emitting event reports.

Index

Index

Index