Preliminary Specification

Inter-domain Management: Specification Translation

The Open Group

Preliminary Specification

Inter-domain Management: Specification Translation

ISBN: 1-85912-150-0
Document Number: P509

Published in the U.K. by The Open Group, February 1997.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Examples

List of Figures

List of Tables

Preface

The Open Group

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

This document was developed by the Joint Inter-Domain Management (JIDM) working group, an activity jointly sponsored by X/Open and the Network Management Forum (NMF). It addresses the need to provide tools that enable interworking between management systems based on different technologies, notably OSI and SNMP network management and Object Management Group (OMG) CORBA-based management frameworks.

Both OSI Management and SNMP form the basis for long-term network management solutions. Similarly, object-oriented development tools such as those being specified by the OMG are increasingly being used as the basis for systems management frameworks.

In order to facilitate integration between these different management disciplines it is necessary to provide interworking between the different technologies that are employed. In addition, this permits the introduction of technology from one domain into other domains, for example allowing OMG CORBA technology to be integrated into OSI and SNMP network management systems.

Structure

This document is divided into several parts:

- **Part 1:** Introduction
- **Part 2:** ASN.1 to OMG IDL Translation Algorithm
- **Part 3:** GDMO to OMG IDL Translation Algorithm
- **Part 4:** OMG IDL to GDMO/ASN.1 Translation Algorithm
- **Part 5:** SNMP to OMG IDL Translation Algorithm
- **Part 6:** OMG IDL to SNMP Translation Algorithm
  This part of the document is currently not provided.
- **Part 7:** IDL Modules and Examples
- **Part 8:** Comparison of Object Models
  This is an updated version of NMF Technical Report TR107.

Typographical Conventions

Within this dcoument the following conventions are used:

`SNMP, ASN.1 and GDMO text fragments appear in Courier.`

IDL text fragments appear in Helvetica Bold.

PIDL text fragments appear in Helvetica.

In a number of sections, grammars appear. In these grammars, <xxx> denotes a non-terminal element and a **bold** typeface denotes literals.

Trademarks

Motif®, OSF/1® and UNIX® are registered trade marks and the ‘‘X Device’’™ and The Open Group™ are trade marks of The Open Group.

Acknowledgements

The Open Group acknowledges the work of the Joint Inter-domain Management (JIDM) working group, comprising members of The Open Group and the Network Management Forum (NMF). This document was developed by the JIDM, under the auspices of the Collaboration Agreement between NMF and The Open Group. The Open Group and NMF wish to thank all those experts who contributed to the development of this **Inter-domain Management: Specification Translation** document. Special thanks are due to:

Colin Ashford	BNR
Prabha Chadayammuri	HP
Jesus Gonzalez	Telefonica Investigacion y Desarrollo
Juan Jose Hierro	Telefonica Investigacion y Desarrollo
Ulf Hollberg	IBM
Martin Kirk	The Open Group
Subrata Mazumdar	Lucent Technologies (formerly at IBM Research Labs)
Tim Roberts	BNR
Tom Rutt	Lucent Technologies (formerly at AT&T Bell Labs)
Nader Soukouti	SMILE (formerly at ESIGETEL)

Referenced Documents

The following documents are referenced in this specification:

ASN.1
ITU-T Recommendation X.208 | ISO 8824: 1990 Information Technology — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1).

ASN.1:1994
ITU-T Recommendation X.680 (1994) | ISO/IEC 8824-1:1995, Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation.

Boo91
Booch, G., Object-oriented Design with Applications, Benjamin/ Cummings, Redwood City, CA, 1991.

CMIP
ITU-T Recommendation X.711 | ISO/IEC 9596-1: 1991, Information Technology — Open Systems Interconnection — Common Management Information Protocol, Part 1: Specification.

CORBA
The Common Object Request Broker: Architecure and Specification, OMG Document, Revision 2.0, July 1995

COS,
CORBA Services: Common Object Services Specification, OMG Document Number 95-3-31, Revised Edition, March 1995

DIR
ISO/IEC 9594: 1991, Information Technology — Open Systems Interconnection — Management Information Services — The Directory.

ESS
Event Service Specification, chapter 4 in CORBA Services: Common Object Services Specification, OMG Document Number 95-3-31, March 1995.

GDMO
ITU-T Recommendation X.722 | ISO/IEC 10165-4:1992, Information Technology — Open Systems Interconnection — Structure of Management Information — Part 4: Guidelines for the Definition of Managed Objects.

Gold89
Goldberg, A. and Robson, D., Smalltalk-80, Addison-Wesley, Reading, Mass,1989.

Hausz86
Hauzer, B. M. A Model for Naming, Addressing, and Routing. ACM Trans. Off. Inf.Sys, 4(4), Oct 1986.

IADM RFC1445, J.R. Davin, J.M. Galvin, K.McCloghrie, Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2), April 1993.

ISMI
RFC 1155, M. Rose and K. McCloghrie, Structure and Identification of Management Information for TCP/IP based Internets, May 1990.

ISMIV2
RFC 1442, J.D. Case, K. McCloghrie, M.T. Rose, S.L.Waldbusser, Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2), April 1993.

ISO / IEC 7498-4
ITU-T Recommendation X.700 | ISO / IEC 7498-4: 1989, Information Processing Systems — Open Systems Interconnection — Basic Reference Model — Part 4: Management Framework.

ISO 8859-1
ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

Jacqm90
Jacqmot, C., Milgrom, E., Joosen, W, and Berbers, Y., Naming and Network Transparent Process Migration in Loosely Coupled Distributed Systems.

In Decentralised Systems, Eds Cosnard, E and Girault, C, Elsevier, North-Holland, 1990.

Kent91
Kent, W., A Rigourous Model of Object Reference, Identity, and Existence, Journal of Object-Oriented Programming, (4)3, June 1991.

Krug92
Krueger, C. W., Software Reuse, ACM Comput. Surv. 24(2), June 1992.

Lalon91
Lalonde, W. and Pugh, J., Subclassing vs. Subtyping, Journal of Object-oriented Programming, (3)5, January 1991.

Mey88
Meyer, B., Object-Oriented Software Construction, Prentice Hall, Englewood Cliffs, NJ, 1988.

MPR
Network Management Forum, Modelling Principles for Managed Objects, TR102, Bernardsville, NJ, 1991.

Naur68
Naur, P. and Randell, B., Eds., Software Engineering: Report on a Conference by the NATO Science Committee, NATO Scientific Affairs Division, Brussels. 1968.

ODP93-2
ITU-T Recommendation X.902 | ISO / IEC 10746-2, Open Distributed Processing — Reference Model — Part 2: Foundations.

ODP93-3
ITU-T Recommendation X.903 | ISO / IEC 10746-2, Open Distributed Processing — Reference Model — Part 3: Architecture.

OMNI
Network Management Forum, Discovering OMNIPoint, PTR Prentice Hall, New Jersey. 1993.

OOM
Object Management Group / Object Model Task Force, The OMG Object Model V0.9, Boulder, CO, 1991.

XAP-ROSE

Preliminary Specification, January 1994, ACSE/Presentation: Remote Operations Service Element API (XAP-ROSE) (ISBN: 1-872630-86-3, P302).

Rumb91

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W., Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.

SMI

ITU-T Recommendation X.720 | ISO/IEC 10165-1 1991, Information Technology — Open System Interconnection — Management Information Services — Structure of Management Information — Part1: Management Information Model.

SNMP

RFC 1157, J.D. Case, M.S. Fedor, M.L. Schoffstall,C. Davin, Simple Network Management Protocol (SNMP), May1990.

SNMPV2

RFC 1448, J.D. Case, K. McCloghrie, M.T. Rose, S.L.Waldbusser, Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2), April 1993.

Weg90

Wegner, P., Concepts and Paradigms of Object-oriented Programming, OOPS Messenger, (1)1, August 1990.

Preliminary Specification

Part 1:
Introduction

The Open Group

Chapter 1

Introduction

1.1 Scope and Purpose

This document is the first deliverable from the Joint Inter-domain Management (JIDM) working group, an activity jointly sponsored by The Open Group and the Network Management Forum (NMF). This project was initiated in response to a perceived need to provide tools that would enable interworking between management systems based on different technologies.

In the real world there are several technologies that are appropriate to solving this complex task. Each has strengths and weaknesses and will undoubtedly feature in future network management systems. The Open Group and Open-Network Management Forum Joint Inter-domain Management (JIDM) group has identified three key technologies:

- CMIP (see reference **CMIP**)
- SNMP (see reference **SNMP**)
- CORBA (see reference **CORBA**),

and is seeking to enable interoperability between them, both within a single organisation and between organisations. SNMP has a large embedded base in the general purpose computing market, CMIP is mandated in the telecommunications arena by the TMN standard, and CORBA is recognised as the emerging standard covering distributed object oriented programming. Each technology has its strength; thus full interoperability will enable designers to select the most appropriate technology to apply to any given problem. The Network Management Forum ISO-Internet Management Coexistence (see reference **IIMC**) group has addressed SNMP/CMIP interoperability. Thus JIDM has chosen to concentrate on CMIP/CORBA and SNMP/CORBA interworking.

To enable interworking it is necessary to be able to map between the relevant object models and to build on this to provide a mechanism to handle protocol conversion on the domain boundaries. The results of comparing the object models of OSI Management, CORBA and Internet Management is provided in Part 8 of this document. For a particular pair of domains, the specification of the mapping is split into two parts:

- The first part, covered in this document, is referred to as Specification Translation, and is expressed as a mechanism for translating between GDMO (the object definition language used in conjunction with CMIP — see reference **GDMO**), SNMP MIB Definition language (see references **SNMPv2** and **ISMIv2**), and CORBA's Interface Definition Language (IDL — see reference the CORBA domain.
- The second part, to be detailed in a subsequent document, is known as Interaction Translation and covers the mechanisms to dynamically convert between the protocols in one domain and the protocols within the other without either party necessarily being aware of the conversion. 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. For example, an object in the CORBA domain should be able to interact with a GDMO object as if it were in the CORBA domain, ideally without having to know that the target object is in a different domain. Naturally the converse is also true, that an OSI Manager should be able to manage CORBA objects as if they were defined in GDMO (this requires the reverse mapping).

The main advantage of this is that the strength of CORBA (object oriented system with wel-ldefined APIs which are aimed at standardising and simplifying the task of creating distributed applications) can be combined with the strength of CMIP (powerful protocol with strong wire compatibility allowing integration of multi-vendor hardware) to give the best of both worlds. The implementor would have an effective environment in which to implement manager or agent functionality and yet be able to easily integrate components from multiple vendors. Figure 1-1 on page 7 illustrates the main interoperability scenarios identified for the OSI and CORBA domains. Figure 1-2 on page 8 illustrates those for Internet and CORBA.

1.2 Specification Translation

Specification Translation covers the process by which specifications are translated from one specification to another. It is a static process which may be required to generate additional material for use in Interaction Translation. In this document an algorithm for the static translation of GDMO specifications to and from IDL interfaces, and the static translation from SNMP MIBs to IDL only, is described. This document does not attempt to detail the generation of additional information required by Interaction Translation but does make reference to the Interaction Translation process where this may impact on the static translation.

When translating from GDMO to IDL, trade-offs are encountered between enabling access to the full power of CMIP and generating simple IDL representations which simplify the application programmer's task. Wherever possible, these have been resolved according to the principle of keeping it simple in the most number of cases at the expense of making lesser used constructs more complex.

1.3 Interaction Translation

Interaction Translation covers the process by which interactions from one domain are mapped onto one or more interactions from the other domain. A gateway, the entity responsible for translating interactions, might receive a CMIP PDU and must map this into one or more requests or replies on IDL interfaces. For example, if a scoped and filtered CMIP GET request is received, the gateway would have to identify the set of objects matching the filter within the scope and invoke the appropriate operation on each of those objects. The results would be collated and formatted into one or more CMIP PDUs in reply. It is the responsibility of the Interaction Translation document to define how this kind of translation is performed.

This is clearly a fundamental part of the work. To date, significant effort has been put in to address the content of this. At this stage, it seems that production of a gateway for a particular set of GDMO or SNMP MIB definitions is entirely feasible. The main outstanding issue is the provision of dynamic access in the CORBA domain to Management Information about these specifications, for example, for example, default values, sub-ranges not representable in IDL, behaviours, etc. This dynamic access allows more powerful and generic tools to be constructed and, in particular, a gateway that is independent of the particular object model could be constructed.

In addition, Interaction Translation must cover initialisation identifying how the gateway is initialised and populated. How it identifies the existing object population and what other service instances it may need to use. The gateway will probably interact with existing standard services in the CORBA domain, for example, OMG Name Service to resolve Distinguished Names, OMG Lifecycle Service to create new object instances and the use of OMG Event Channels for event distribution.

Interaction Translation requires that the interactions be captured by a gateway which converts them in accordance with the mapping rules. Thus, in the OSI/CORBA scenarios, the gateway must:

- receive any incoming CMIP SET/GET/ACTION request and translate it into one or more invocations to methods supported by some object(s)
- receive any event generated by an application object and translate it into an EVENT-REPORT request to be forwarded to remote systems that had register their interest in receiving events
- receive incoming method invocations and forward them as CMIP SET/GET/ACTION requests to some OSI agent
- receive CMIP EVENT-REPORT requests and forward them as CORBA events to interested parties in the CORBA domain
- receive any incoming CMIP CREATE/DELETE request and translate it into an invocation to a method being supported by an object (for example, a factory object)
- receive method invocations for creating/deleting objects in a remote system and forward a CMIP CREATE/DELETE request to some OSI agent.

This protocol conversion is complicated by such things as the need to map identifiers due to the differences between GDMO and IDL scoping and case-sensitivity, to map between GDMO Distinguished Names and CORBA Object References and to handle CMIP scoping and filtering requests which may require one CMIP request to be mapped to multiple sequences of IDL operations.

Whilst it may seem desirable to map the type `ObjectInstance` from CMIP (see reference **CMIP**), directly to CORBA Object References, this is not the correct mapping because the semantics are different. The translation maps the type `ObjectInstance` exactly as any other

ASN.1 defined compound type would be mapped, that is, retaining the Distinguished Name format. The use of the DistinguishedName is entirely self consistent and does not require the Specification Translation process to translate `ObjectInstance` as a special case. Also the Interaction Translation process will require the ability to convert between `ObjectInstance` and CORBA Object References, so run-time facilities will exist to do the mapping for applications if necessary. These will be addressed within the the Interaction Translation document.

1.4 Usage Overview

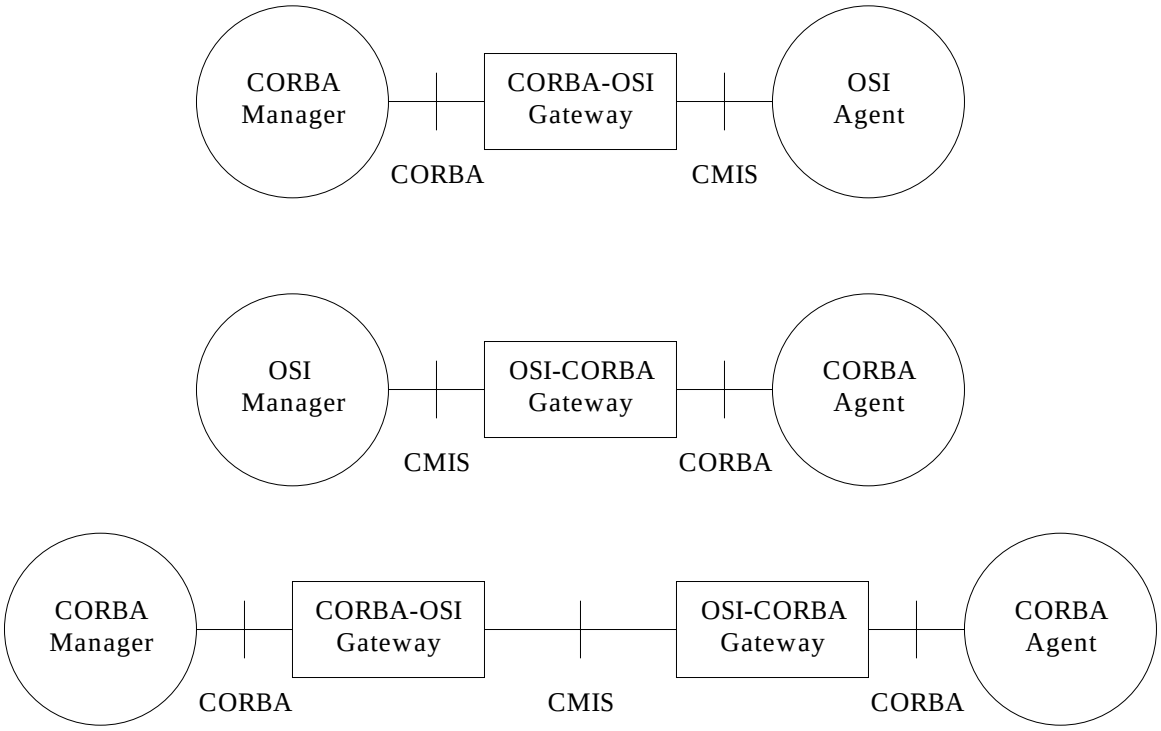

Figure 1-1 OSI / CORBA Interoperability Scenarios

A common scenario is that a set of objects are defined in GDMO. The GDMO is statically translated via the Specification Translation algorithm in this document, into IDL interfaces. A manager implemented as a set of CORBA objects, would manage objects supported by an OSI agent as if they were CORBA objects (that is, via the generated IDL interfaces). These interfaces would be supported by a gateway supporting the Interaction Translation algorithm. This would dynamically translate IDL requests into CMIP PDUs based on the original GDMO specification. The Interaction Translation is obviously bi-directional translating CMIP PDUs originating from the Agent into the appropriate IDL requests and replies. Conversely, if the Manager is an OSI manager, it generates CMISE exactly as if the objects were supported by an OSI agent. The CMIP protocol is terminated by the gateway which dynamically translated the CMIP PDUs into IDL requests on the CORBA implemented object.

The final case illustrates the use of CMIP as an environment-specific interoperability protocol. It allows both the Manager and the Agent to be implemented in the CORBA domain and yet to offer the standard Q3 interface externally. Neither party is aware of the implementation of the other but two gateways back-to-back ensure the smooth working of the system. This is obviously less efficient than direct IDL invocation, however, it does allow use of CMIP as an environment specific interoperability protocol, which could be used in the TMN environment.

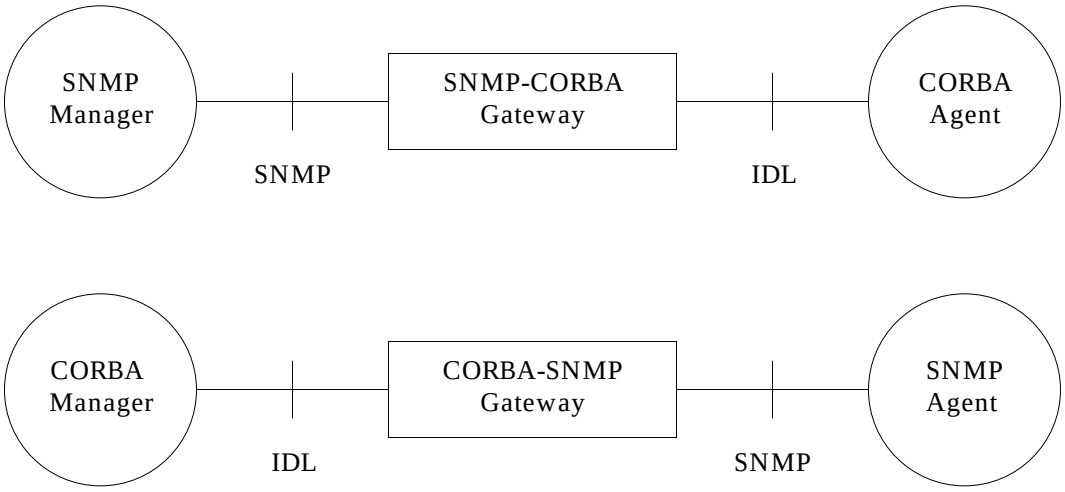

Figure 1-2 SNMP/CORBA Interoperability Scenarios

Figure 1-2 illustrates a similar set of scenarios between Internet management and CORBA.

A CORBA Agent is an Agent which has its object definitions specified in CORBA IDL. A CORBA Manager is a Manager which has its object definitions defined in CORBA IDL.

Using the scenarios described in this section, a CORBA Manager may interact with OSI, SNMP, and CORBA Agents.

1.5 Futures

Figure 1-1 on page 7 identified a use of back-to-back gateways for interoperability. This technique may be re-applied to provide multiple translations as required.

In addition, this gateway approach is entirely in line with the interoperability specification adopted by the OMG as part of CORBA 2.0 (see reference **CORBA**). The OMG is currently in the process of drafting a Request for Proposals (RFP) on CORBA-COM interoperability which is likely to require the details of specification and interaction translation necessary to enable interworking between Microsoft Windows applications and CORBA applications. This move by the OMG to standardise interworking with non-CORBA domains should be extended to cover GDMO and SNMP, so it is desirable that this document and its companions be submitted for approval through the OMG using its *fast-track* process.

It should be noted that future work on Interaction Translation may introduce further requirements on the translator.

1.6 Assumptions and Principles

The algorithms have been designed using a number of guiding principles to resolve issues:

- Completeness:
  The aim was to provide a complete mapping insofar as it was possible. Rules have been provided for all cases regardless of their frequency. In some cases, this means explicitly ignoring information, (usually this will be addressed in Interaction Translation), but all cases should be considered.
- Simplicity (The 80-20 rule):
  ASN.1 allows many constructs that are difficult to map into IDL. Many of these are not frequently used. In the light of the completeness principle, it was decided to select the simplest mapping for 80% of the cases allowing the remaining, more obscure cases, to be more complicated if necessary.
- Reuse of OMG services:
  The CORBA domain does not have a Network Management architecture; it provides a distributed processing environment. It is populated by an increasing number of Object Services such as Naming and Events which are useful building blocks. The JIDM group has tried to exploit these facilities where possible, for example, Event services are used for OSI notifications.
- Freedom of implementation:
  This document refrains from defining or constraining implementations unless it is absolutely necessary. Whilst the group discussions have naturally established the feasibility of implementation, the document does not attempt to provide hints.

1.7 Document Structure

This document defines algorithms for translating between CORBA IDL and the OSI and SNMP notations based on GDMO and ASN.1. Translations between OSI and SNMP notations are not contained in this document as they have already been addressed by the ISO-Internet Management Co-existence (IIMC) work sponsored by the NMF.

This document is divided into several parts:

Part 1: Introduction

Part 2: ASN.1 to OMG IDL Translation Algorithm
This part of the document provides common translation rules used by both the GDMO and SNMP translation algorithms.

Part 3: GDMO to OMG IDL Translation Algorithm
This part of the document addresses the translation of OSI GDMO-based specifications into OMG IDL.

Part 4: OMG IDL to GDMO / ASN.1 Translation Algorithm
This part of the document addresses the translation of OMG IDL-based specifications into GDMO.

Part 5: SNMP to OMG IDL Translation Algorithm
This part of the document addresses the translation of SNMP-based specifications into OMG IDL.

Part 6: OMG IDL to SNMP Translation Algorithm
This part of the document is currently not provided.

Part 7: IDL Modules and Examples
This part of the document contains the standard IDL modules defined in the specification, and also provides informative examples of the application of the algorithms defined within the document. In the case of any discrepancies between the examples and the specification of the algorithms, the specification is to be regarded as definitive.

Part 8: Comparison of Object Models
This part of the document provides a comparison of the OSI, SNMP and OMG object models. It is an updated version of NMF Technical Report TR107.

Preliminary Specification

Part 2:
ASN.1 to OMG IDL Translation Algorithm

The Open Group

Chapter 2

ASN.1 Type to CORBA-IDL Translation

2.1 Introduction

In this part, the Specification Translation process for ASN.1 modules is described in terms of inputs and outputs and a rough outline of the process is given. The process will be implemented via a compiler which operates on a set of input files and results in some output files. Since IDL definitions are processed in terms of files which determine the granularity and reusability of the IDL definitions, it is necessary to specify which definitions are generated and what files they are defined in. In addition, ASN.1 adds some complexities by using lengthy module names such as `InformationNetwork`. Since such names are used to import parts of specifications, there must be a way for the translation process to access the files containing these specifications. In addition, it is desirable to be able to associate the resulting IDL files with the original ASN.1 to facilitate browsing and reuse. This will be done by providing a ‘‘nickname database’’ which maps from the unique registered name of the ASN.1 document (or relevant Object Identifier) to a short nickname suitable for use as a filename base. This nickname will be used to find imported files and to control the names of the generated IDL files.

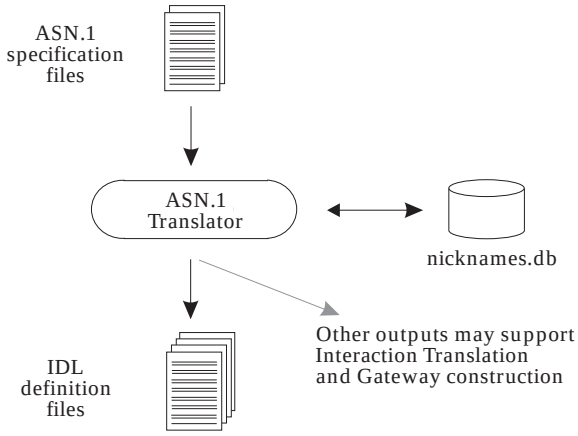

Figure 2-1 Inputs and Outputs for ASN.1 Specification Translation

Since nicknames will be used as the basis for naming files, generated IDL files will only be reusable in an environments where identical nickname databases are used. Therefore, it is desirable to make the nickname database as standard as possible (for example, have standard nicknames for all registered ASN.1 modules). In order to facilitate this, the following nickname selection method is recommended (but not mandatory):

- For ASN.1 modules, the nickname is formed by taking the nickname of the standard in which the module occurs, followed by the first three characters of the module label. For example, the `InformationFramework` module of X.501 would have the nickname *X501Inf*.
- Where more than one module label in a document has the same initial three letters, append a number to the nicknames for the second and subsequent module labels to disambiguate. This means that the first module would have no numeric suffix, the second colliding module would have the suffix ‘‘1’’, the third ‘‘2’’ and so on.

As it is illegal to modify the existing contents of a standard in this context, it is assumed that the nickname always refers to the latest version of the standard. If, for any reason, parts of the original standard are modified in a revision, the last 2 digits of the revision year can be appended to the nickname.

In order to translate between ASN.1 and CORBA IDL, (hereinafter referred to simply as IDL), it is necessary to be able to map the basic definitions (that is, mapping between ASN.1 types and IDL type definitions).

There are two versions of ASN.1 defined, ASN.1:1990 (see reference **ASN1**) and ASN.1:1994 (see reference **ASN1:1994**). Since GDMO explicitly builds on ASN.1 and all new GDMO will provide ASN.1:1990 versions (at least in the short term), this document focuses on that version. However, translation is also provided for all the basic types (for example, BMPString and UniversalString) from ASN.1:1994 as a step towards migration to ASN.1:1994. Further steps on this path may be taken in the future.

In this document, unless otherwise noted, the unqualified name ASN.1 refers to ASN.1:1990.

ASN.1 has a much more complex type system than IDL. As a result, the translation necessarily loses some information; for instance in terms of tag values, sub-range types, compound type constants, etc. Capturing this information for subsequent use in the run-time system is a key issue. A number of schemes have been proposed including the use of string constants and **#pragma** directives, but the definitive statement is deferred to Interaction Translation.

In a number of cases, the complexity of some data types makes it desirable to define operations for manipulating their values. In CORBA, the standard technique is to define pseudo-IDL (PIDL) which allows a fairly tight definition of the operations but with the implication that these operations have library implementations and can only be invoked locally. For example, this is used to provide access methods to support manipulation of the **BitString** data type.

2.2 Outline of the Translation Process

The algorithm that is used to map ASN.1 modules comprises the following steps:

1. Use as input the original published document. The same order of identifiers is fundamental to consistently generating the same IDL code.
2. Map each ASN.1 module to an IDL module in a separate IDL file.
3. Prior to mapping each of the clauses contained in an ASN.1 module, transform it into a canonical form by means of:
   - — ignoring macros[1]
   - — expanding `COMPONENTS OF`, selection types and `WITH COMPONENTS` clauses as described in the following sections in this document.
4. Traverse the contents of the canonical ASN.1 module in order, and map each of the clauses as follows:
   - — Export clauses are ignored.
   - — Import clauses are mapped as described in the following sections in this document.
   - — Type assignments are mapped as described in the following sections in this document. If expanded types are generated, lexical disambiguation of the named type names of the structured outer type (`SET`, `SEQUENCE`, `CHOICE`) must be done prior to anonymous type generation.
   - — Value assignments are mapped either into OMG IDL constants or into operations in a constant interface at the end of the generated IDL module, as explained later in this document.

   Lexical disambiguation is done when generating the mapping in this step, as described in the following sections in this document.
5. Re-order the generated IDL code to obtain valid OMG IDL code by eliminating forward references.

1. Macros are not ignored when translating SNMP.

2.3 File Names and IDL Modules

The output of the above translation is a set of IDL modules and interfaces. These must be organised into files in a way which facilitates reuse and effective generation of code by the CORBA IDL compiler. Thus the Specification Translation process results in potentially several IDL files. During that process, the following rules determine the number and content of each file:

- Each file will start with a comment identifying the ASN.1 module from which it was generated.
- Definitions contained in IDL generated files must be enclosed within **#ifndef ... #endif** directives as shown below and will enclose mappings of ASN.1 template definitions.

```
#ifndef _<capitalised_nickname>_IDL_
#define _<capitalised_nickname>_IDL_
module <nickname> {

<generated-IDL-code>

};
#endif /* _<capitalised_nickname>_IDL_ */
```

- For each ASN.1 module that is contained in the input file, generate an IDL file named **<module_nickname>.idl** containing an IDL module named **<module_nickname>**, where **<module_nickname>** is the nickname that has been assigned to the ASN.1 module.
- For each ASN.1 module from which symbols are imported, add the directive:

  ```
  #include "<module_nickname>.idl"
  ```

  to the corresponding IDL file, where <module_nickname> is the nickname that has been assigned to the imported ASN.1 module. This directive appears before the module being defined.

2.3.1 Standard Files for Specification Translation

The specification translation assumes the existence of a number of standard files containing base definitions and classes. These are as follows:

ASN1Types.idl — contains the base definitions for translating ASN.1 types (see Section 2.11 on page 58).

ASN1Limits.idl — contains the definitions for ASN.1 limits (see Section 12.1.2 on page 168).

2.3.2 Example

Using the recommended nickname convention, the nickname for the standard X.501 InformationFramework ASN.1 module will be *X501Inf*, and will also be used as the filename for the generated IDL:

Example 2-1 File X501Inf.idl

```
// Generated from X501.asn1
// X501Inf.idl file:

#ifndef _X501INF_IDL_
#define _X501INF_IDL_

module X501Inf {


};

#endif /* _X501INF_IDL_ */
```

2.4 Lexical Translation

Since the IDL uses ISO Latin-1 (ISO/IEC 8859-1) character set, each character in the ASN.1 character set maps to itself in IDL.

In ASN.1, names for type-references, identifiers, value-references and module-references consist of an arbitrary sequence of one or more letters, digits, and hyphens. The letters and digits in ASN.1 names are mapped directly retaining case. The ASN.1 hyphen (‘‘-’’) maps to IDL underscore (‘‘_’’). Subject to the possible suffixing as described below, names will be preserved, for example, CMISFilter type in ASN.1 will be mapped as CMISFilterType[n] in IDL.

ASN.1 names are case sensitive and IDL identifiers are not. In addition, ASN.1 has different name-spaces for types-references, identifiers and value-references, whereas IDL has a single namespace. Both have scoped naming spaces, but the naming scopes do not directly map. For example, enumerated types create a new naming scope in ASN.1 but not in IDL. It is therefore not possible to simply map ASN.1 names to IDL identifiers. To handle this, the translator must maintain a table of all identifiers in each resulting IDL scope and modify colliding identifiers within a given IDL scope to avoid such conflicts.

Not all conflicts arise from ASN.1 names. Additional conflicts could arise from clashes with IDL reserved words such as ‘‘interface’’ and identifiers and types defined in the base IDL file ASN1Types.idl and ASN1Limits.idl. The translator must take account of such existing definitions and use the same disambiguating mechanism to avoid clashes.

Clearly, all identifiers could be modified in order to disambiguate, however the goal of the translation mechanism to generate the simplest mapping where possible, leads to a slightly more complex algorithm but preserves a direct mapping where no ambiguity arises. The mapping used will therefore be context dependent and hence the final mapping will be a necessary input to the Interaction Translation since any gateway must be able to replicate such mappings. The rules for disambiguation are as follows:

Rule 1

The first identifier is mapped ‘‘as it is’’, that is, the case is preserved. Second and subsequent identifiers in the same IDL scope that differ only in case will be suffixed with a double underbar followed by a numeric disambiguator, for example, <__n> where n is the count of such instances. Thus the first identifier will not have a suffix, and second and subsequent clashing identifiers will have suffixes <__1>, <__2> and so on. For example, the ASN.1 identifiers `aab`, `aAB` and `aaB` would be translated as **aab**, **aAB__1** and **aaB__2** respectively.

This rule is also applied in the case where ‘‘Choice’’ and ‘‘Choice<__n>’’ is appended in order to disambiguate `CHOICE` values.

Rule 2

The first type reference is mapped with the suffix ‘‘Type’’. Second and subsequent type references in the same IDL scope that differ only in case will be suffixed with ‘‘Type<n>’’ where <n> is the count of such instances. Thus the first type reference will not have the suffix ‘‘Type’’, and second and subsequent clashing type references will have suffixes ‘‘Type1’’, ‘‘Type2’’ and so on. The case of the type reference is always preserved. For example, the ASN.1 type references `Aab`, `AAB` and `AaB` would be translated as **AabType**, **AABType1** and **AaBType2** respectively.

This rule is also applied in the case where ‘‘Choice’’ and ‘‘Choice<n>’’ is appended in order to disambiguate `CHOICE` types.

Rule 3

Value-references are handled as identifiers.

With this scheme, type references cannot clash with identifiers or value references and these latter two are disambiguated numerically. Since the most commonly occurring clash is between data type names and identifiers differentiated only by the case of the first letter, this algorithm avoids the majority of clashes. For example, constructs such as `eventRecord EventRecord` translate to **`EventRecordType eventRecord;`** . In addition, it is easy to recover the original ASN.1 type identifier by finding the ‘‘Type’’ suffix and removing it and all subsequent text from the identifier. Giving preference to identifiers ensures minimum impact on the names of members of sequences and sets and such like so that application code is minimally impacted. The ‘‘Type<n>’’ suffix marks all types explicitly which, in many ways, helps code readability.

Further ambiguity can arise from directly translated names colliding with generated names. For example, a type may have name `MyData`, and a variable may have name `myDataType`. The translation of `MyData` would be **`MyDataType`** and this would collide with the direct translation of `myDataType`. This would be resolved by the normal disambiguation rule. In the same way, translations of ASN.1 constructs may append Opt, Default, etc, to type names, so this must also be considered during disambiguation.

The use of double underbar (‘‘__’’) to separate numeric disambiguators guarantees that there will be no clash with ASN.1 identifiers since ‘‘--’’ (which would translate to ‘‘__’’) is an ASN.1 comment delimiter and thus cannot be part of an identifier. ‘‘_’’ is not a valid character for ASN.1 identifiers.

In subsequent examples, it is assumed that there are no clashes with identifiers or types outside the given ASN.1 text. Thus identifiers largely map unchanged and type identifiers are mapped mapped with the ‘‘Type’’ suffix.

Warning

This disambiguation scheme is order sensitive. Thus changes to the ASN.1 which make no difference to the meaning and hence are allowed, may impact the translation changing some of the disambiguation. It is important that users are aware of this and use identical source for both manager and agent sides. It is recommended that the unmodified standard texts are used.

For lexical mapping collisions, a strict order of the ASN.1 input actually mapped to IDL is applied. That is, identifiers, names or literals that are not used for the translation process, are not considered for lexical collision purposes. This means that:

Whenever mapping says that the original ASN.1 code should first be modified or expanded prior to being translated, the resulting code is the valid one for the lexical mapping collision algorithm. This actually affects selection types, `COMPONENTS OF` clauses, `WITH COMPONENTS` clauses, and other unused identifiers, (for example, value numbers of an enumerated literal when the value is specified as a defined value, or identifiers of unreachable code as in the examples below).

2.4.1 Example

Here the ‘‘x’’ representing the value of the enumerated literal ‘‘b’’ in type ‘‘A’’ is not used in the translation process:

```
A ::= ENUMERATED { a(1), b(x) }
B ::= ENUMERATED { x(1), y(2) }
x ::= INTEGER = 3
```

The defined value ‘‘x’’ of the ‘‘b’’ literal in ‘‘A’’ type is not mapped, so it is not considered. Thus resulting mapping code would be:

```
enum AType = {a, b};
enum BType = {x, y};
const ASN1_Integer x__1 = 3;
```

2.5 Mapping ASN.1 Module to IDL Module

ASN.1 Modules are defined according to the production rules in Table 2-1 which are printed here for reference for the discussion of the subsequent text.

```
<ModuleDefinition> ::=
    <ModuleIdentifier> DEFINITIONS <TagDefault> ::= BEGIN
        <ModuleBody>
    END

<ModuleIdentifier> ::=
    <modulereference>
    <DefinitiveIdentifier>

<TagDefault> ::= EXPLICIT TAGS | IMPLICIT TAGS |
                 AUTOMATIC TAGS | empty

<ModuleBody> ::=
    <Exports> <Imports> <AssignmentList> | empty

<Exports> ::=
    EXPORTS <SymbolsList> ; | empty

<Imports> ::=
    IMPORTS <SymbolsImported> ; | empty

<SymbolsImported> ::=
    <SymbolsFromModuleList> | empty

<SymbolsFromModuleList> ::=
    <SymbolsFromModule> | <SymbolsFromModuleList> <SymbolsFromModule>

<SymbolsFromModule> ::=
    <SymbolList> FROM <GlobalModuleReference>

<GlobalModuleReference> ::=
    <modulereference AssignedIdentifier>

<SymbolList> ::=
    <Symbol> | <SymbolList> , <Symbol>

<AssignmentList> ::=
    <Assignment> | <AssignmentList> <Assignment>
```

Table 2-1 Production of Module Definition

2.5.1 Mapping of Module Identifier

A module identifier is mapped as the name of the corresponding IDL module. Module identifier information is mapped as an IDL comment as follows:

// ModuleIdentifier:<ModuleIdentifier>

In addition, an IDL **#pragma** is generated as follows:

#pragma ID <moduleNickname> "OSIOID:<DefinitiveIdentifier>"

If there is no definitive identifier then nothing is generated.

2.5.2 Mapping of Tag Default

The information in the TagDefault production is ignored during the mapping of ASN.1 to IDL.

2.5.3 Mapping of Exports

The information pertaining to Exports productions is ignored during the mapping of ASN.1 to IDL.

2.5.4 Mapping of Imports

`IMPORTS` clauses are mapped as a list of #include directives for the file corresponding to the imported module inside the ASN.1 module and a list of typedefs and constants. The relevant references enable the module to be disambiguated and hence its nickname identified. It should also be noted that the compiler must parse any imported document in order to apply the disambiguation rules for clashing identifiers. Naturally, this is recursive; when parsing an imported document, any documents it imports must also be parsed. If a required document is not available, the behaviour of the translator is implementation defined.

For each module imported from, an include directive is generated before the importing module definitions as follows:

```
#include <module_nickname>.idl
```

The production:

```
IMPORTS <symbol1>, ..., <symboln> FROM <GlobalModuleReference>
```

is mapped as follows for imported types:

typedef <moduleNickname>::<mapped symbol1> <mapped symbol1>
.....
typedef <moduleNickname>::<mapped symboln> <mapped symboln>

The two mapped symbols may be different as they are disambiguated in different naming contexts - the imported from and the importing modules.

For imported ASN.1 values, code will be generated depending on whether their types are simple or complex (see Section 2.7.2 on page 29). For simple types the following code will be generated:

const <type of imported const> <mapped symbol1> =
<moduleNickname>::<mapped symbol1>
...
const <type of imported const> <mapped symboln> =
<moduleNickname>::<mapped symboln>

For imported ASN.1 values of complex types, a method with a corresponding identifier will be generated within the **constValues** interface of the IDL module being generated.

Note that <GlobalModuleReference> is only used to identify the unique module nickname and does not appear in the IDL.

2.5.5 Mapping of Referencing Type and Value Definition

When identifiers from external modules are referenced, they must be mapped as IDL scoped identifiers where the scope is <moduleNickname> since these are unique in the translation environment, no further scoping is required.

Otherwise, they are mapped as per normal.

2.5.6 Mapping of Assigning Types

Type assignments are mapped to **typedefs** in IDL.

2.5.7 Mapping of Assigning Values

Each constant value defined in ASN.1 is mapped as the definition of a constant literal in IDL. IDL supports constants for only the primitive types, and values not directly representable in IDL are mapped via the mechanism in Section 2.7.2 on page 29.

2.6 Mapping of ASN.1 Comments

The mapping of comments is optional. If they are mapped, it should be noted that re-ordering of the IDL may fail to re-order the associated comments, thus rendering the translated comments of doubtful value.

An ASN.1 comment commences with a pair of adjacent hyphens (‘‘--’’) and ends with the next pair of adjacent hyphens or at the end of the line, which ever comes first. In IDL, comments are delimited as per C++, using either **/*** and ***/** or **//** and end of line. The choice of which delimitation to use is left as an implementation concern.

2.7 Mapping of Primitive ASN.1 Types and Values

2.7.1 Mapping of ASN.1 Primitive Types

Table 2-2 describes the default mapping of primitive ASN.1 types to IDL types. The algorithm for mapping types is based on reducing all types to primitive elements and mapping each primitive according to this table. As an aid to mapping back to the ASN.1 document, typedefs are provided which define the ASN.1 primitive typename in IDL. For example, the ASN.1 type INTEGER maps to the IDL type long. However, a typedef of the IDL identifier ASN1_Integer to long is provided to improve consistency between the original document and the resultant IDL.

ASN.1 Type	IDL Type
BOOLEAN	**typedef boolean ASN1_Boolean;**
INTEGER	**typedef long ASN1_Integer;**
REAL	**typedef double ASN1_Real;**
NULL	**typedef char ASN1_Null;** **const ASN1_Null ASN1_NullValue = '\x00';**
ENUMERATED	**enum <enumName> {<elem1>,....., <elemn>};**
BIT STRING	**typedef sequence<octet> ASN1_BitString;** **// supported by PIDL**
OCTET STRING	**typedef sequence<octet> ASN1_OctetString;**
IA5 STRING ISO646 STRING NUMERIC STRING PRINTABLE STRING TELETEXT STRING T61 STRING VIDEO STRING VISIBLE STRING	**typedef string ASN1_IA5String;** **typedef string ASN1_ISO646String;** **typedef string ASN1_NumericString;** **typedef string ASN1_PrintableString;** **typedef string ASN1_TeletextString;** **typedef string ASN1_T61String;** **typedef string ASN1_VideoString;** **typedef string ASN1_VisibleString;**
GENERAL STRING GRAPHIC STRING	**typedef sequence<octet> ASN1_GeneralString;** **typedef sequence<octet> ASN1_GraphicString;**
BMP STRING UNIVERSAL STRING	**typedef sequence<unsigned short> ASN1_BMPString;** **typedef sequence<unsigned long> ASN1_UniversalString;**
OBJECT IDENTIFIER	**typedef string ASN1_ObjectIdentifier;**
ANY ANY DEFINED BY	**typedef any ASN1_Any;** **typedef any ASN1_DefinedAny;**
Tagged	**as untagged type**
EXTERNAL	**struct ASN1_External {** **ASN1_ObjectIdentifier syntax;** **ASN1_DefinedAny data_value; // by 'syntax'** **};**

Table 2-2 Mapping of ASN.1 Types to IDL Types

2.7.2 Mapping of Values

IDL only permits constants of primitive types (integer, boolean, floating-point, character and string types). As such it cannot adequately represent constant values of more complex ASN.1 types. Simple and typesafe access to these constants, which are widely used for such things as default values, is important. For all constant values which cannot be represented in IDL, a special constant values PIDL interface is defined in the same module as the module containing the const declaration. Whilst this does not actually define a constant, the constant value is accessible as the return value of an operation in this interface. The implementation of this constant values interface is responsible for ensuring that the values returned correspond to the declaration[2].

Functions defined as translation of complex ASN.1 constants are included at the end of the IDL modules in the **ConstValues** interface. In general, a constant declaration of type `<ConstType>` with identifier `<constName>` would result in the generation of the operation:

```
interface ConstValues {
        <ConstType> <constName>(); // returns "<text of constant value definition>"
    ........
};
```

For examples, see Section 2.7.8.2 on page 34.

2.7.3 Mapping of NULL types

The ASN.1 NULL type is a type which has only one value, NULL. This type is mostly used to indicate the absence of a component of a sequence or set, or of an alternative of a choice, The ASN.1 NULL type is mapped to a typedef and a constant with the value:

```
typedef   char    ASN1_Null;
const     ASN1_Null ASN1_NullValue = '\x00';
```

2.7.4 Mapping of Boolean Type

ASN.1 `BOOLEAN` types map directly to the IDL **boolean** type however, a typedef for the type ASN1_Boolean is provided in IDL. This is used in the translation to give greater textual correspondence to the source ASN.1.

2.7.4.1 Examples

Example 2-2 Mapping of Boolean Type

ASN.1	IDL
`Married ::= BOOLEAN` `maritalStatus Married ::= TRUE`	**typedef ASN1_Boolean MarriedType;** **const MarriedType maritalStatus = TRUE;**

2. These routines could be automatically generated as part of the translation process.

2.7.5 Mapping of Integer Type

Unconstrained ASN.1 `INTEGER` is mapped to the IDL type **long**. As the type **ASN1_Integer** is provided in the IDL, the mapping uses this directly. ASN.1 integer subtypes with value ranges have special mappings covered in Section 2.10.2.1 on page 53. It is assumed that non-subtyped `INTEGER` can be handled by 32-bit integer. In addition, name-number lists result in additional IDL constants holding the named values.

For a named-number, the form `<name>(<number>)` within an integer type `<type>` generates the IDL const declaration:

const <type>Type[n] <name> = <number>;

The form `<name>(<identifier>)` within an integer type `<type>` generates the IDL const declaration:

const <type>Type[n] <name> = <translated identifier>;

Note that the identifier must also be translated in order to ensure that the right IDL identifier is referenced. Note also that these named values are subject to disambiguation in the normal way.

2.7.5.1 Examples

Example 2-3 Mapping of Integer Type

ASN.1	IDL
`T0 ::= INTEGER`	**typedef ASN1_Integer T0;**
`a INTEGER ::= 1`	**const ASN1_Integer a = 1;**
`T1::= INTEGER { a(2) }`	**typedef ASN1_Integer T1Type;** **const T1Type a = 2;**
`ax INTEGER ::= 1` `aX INTEGER ::= 2` `T2 ::= INTEGER { a(3), b(aX) }` `c T2 ::= b` `d T2 ::= a`	**const ASN1_Integer ax = 1;** **const ASN1_Integer aX__1 = 2;** **typedef ASN1_Integer T2Type;** **const T2Type a = 3;** **const T2Type b = aX__1;** **const T2Type c = b;** **const T2Type d = a;**

2.7.6 Mapping of Real Type

The ASN.1 REAL type is mapped to IDL type **double**, and a typedef for the type **ASN1_Real** is provided and is used in the translated IDL. The reason for selecting double is to provide maximum precision since there is no way of determining the required precision.

ASN.1 constants PLUS-INFINITY and MINUS-INFINITY are mapped to IDL double precision floating point constants **plus_infinity** and **minus_infinity** respectively. The values of these constants are defined int the file ASN1Limits.idl. In ASN.1, a floating point value may be defined as a triple of mantissa, base and exponent. In this case, the value of the constant is calculated and the corresponding double precision decimal floating point constant is used

2.7.6.1 Examples

Example 2-4 Mapping of Real Type

ASN.1	IDL
`AngleInRadians ::= REAL`	**typedef ASN1_Real AngleInRadiansType;**
`pi REAL ::= { 3141592653897, 10, -12}`	**const ASN1_Real pi =3.141592653897;**

2.7.7 Mapping of Enumerated Type

As shown in Table 2-2 on page 28, ASN.1 `ENUMERATED` is mapped to an IDL **enum** type. The primary benefit of this mapping is that it is more natural for the programmer. With the OMGs C++ mapping, IDL enum is mapped to a C++ enum so there is also strong type checking available.

This does not preserve the actual values of the element values and the Interaction Translation process would need to support the dynamic mapping between values received on the wire and the corresponding enum values. As a consequence of this, any application requiring to access the actual values would be able to retrieve them via the management knowledge at run-time. The full details of the creation and maintenance of the management knowledge is addressed in the Interaction Translation Document.

It is important to note that the names of values in an enumerated type are identifiers in the enclosing scope since in IDL, enum does not define a new naming scope. This means that enum names are subject to disambiguation in the normal way but in a slightly wider scope than might otherwise be expected.

2.7.7.1 Examples

Example 2-5 Mapping of Enumerated Type

ASN.1	IDL
`Message ::= ENUMERATED {basic(0), extended(1)}`	**enum MessageType { basic, extended };**
`DayOfTheWeek ::= ENUMERATED { sunday(0),monday(1), tuesday (2), wednesday (3), thursday(4), friday(5), saturday(7)}` `first DayOfTheWeek ::= sunday`	**enum DayOfTheWeekType { sunday, monday, tuesday, wednesday, thursday, friday, saturday };** **interface ConstValues {** **....** **DayOfTheWeekType first(); // returns "sunday";** **....** **};**
`MaritalStatus ::= ENUMERATED { single(0), married(2), widowed(1)} }`	**enum MaritalStatusType { single, married, widowed };**

2.7.8 Mapping of Bit String Type

ASN.1 `BIT STRING` is mapped to the IDL type **ASN1_BitString** defined as **sequence<octet>** with global scope in the IDL file ASN1Types.idl. **ASN1_BitString** is defined as follows to match the BER encoding of `BIT STRING`. The sequence of octet shall have an initial octet followed by zero, one, or more subsequent octets.The bits in the bitstring, commencing with the first bit and proceeding to the trailing bit, shall be placed in bits 8 to 1 of the second octet, followed by bits 8 to 1 of each octet in turn, followed by as many bits as are need in the final octet, commencing with bit 8 (the notation ‘‘first bit’’ and ‘‘"trailing bit’’ is specified in ISO 8824). The initial octet shall encode, as an unsigned binary integer with bit 1 as the least significant bit, the number of unused bits in the final octet. The number shall be in the range zero to seven. If the bitstring is empty, there shall be no subsequent octets, and the initial octet shall be zero.

`BIT STRING` literal values encoded in this way, cannot be represented as IDL constants and are handled via the mechanism defined in Section 2.7.2 on page 29.

Each named bit is mapped as an IDL constant of type **unsigned long** with value equal to the offset into the bit string. The name of the constant is the given name, disambiguated by the usual rules:

const unsigned long <bitname> = <offset>;

Note that offset may be defined by another constant.

2.7.8.1 PIDL for BitString Access Functions

To facilitate manipulation of BitString values, the following PIDL interface is proposed. This essentially defines a library of utility BitString access functions to standardise and simplify BitString manipulation.

```
interface ASN1_BitStringHandler {      // PIDL
typedef short BitValue;

ASN1_BitString createBitString (in unsigned long number_of_bits);

BitValue getBit (in ASN1_BitString bit_string, in unsigned long
position);

void setBit (inout ASN1_BitString bit_string, in unsigned long position,
      in BitValue new_bit_value);

long length (in ASN1_BitString bit_string);

string asString (in ASN1_BitString bit_string);
// produces a string with binary values ("1001011B")

void setFromString (inout ASN1_BitString bit_string, in string
string_value);
};
```

2.7.8.2 Examples

Example 2-6 Mapping of Bit String Type

ASN.1	IDL
`MessageFlag ::= BIT STRING {` `  posResp (0), negResp (1),` `  doNotForward (2) }`	**typedef ASN1_BitString MessageFlagType;** **const unsigned long posResp = 0;** **const unsigned long negResp = 1;** **const unsigned long doNotForward = 2;**
`T0 ::= BIT STRING` `a INTEGER ::= 1` `T1::= INTEGER { a(2) }` `T2 ::= BIT STRING { a(3), b(a) }`	**typedef ASN1_BitString T0Type;** **const ASN1_Integer a = 1;** **typedef ASN1_Integer T1Type;** **const T1Type a__1 = 2;** **typedef ASN1_BitString T2Type;** **const unsigned long a__2 = 3;** **const unsigned long b = a;**
`G3FacsimilePage ::= BIT STRING` `-- a sequence of bits conforming to` `-- Recommendation T.4` `image G3FacsimilePage ::=` `  ’100110100100001110110’B` `trailer BIT STRING ::=` `  ’0123456789ABCDEF’H`	**typedef ASN1_BitString G3FacsimilePageType;** **interface ConstValues {** **G3FacsimilePageType image();** **// "’100110100100001110110’B";** **BitString trailer();** **// "’0123456789ABCDEF’H";** **};**
`PersonalStatus ::= BIT STRING` `  {married (0), employed (1),` `  veteran(2), collegeGraduate (3)}` `johnDoe PersonalStatus ::=` `  {married, employed,` `  collegeGraduate}`	**typedef ASN1_BitString PersonalStatusType;** **const unsigned long married = 0;** **const unsigned long employed = 1;** **const unsigned long veteran = 2;** **const unsigned long collegeGraduate = 3;** **interface ConstValues {** **PersonalStatusType johnDoe();** **// "married, employed, collegeGraduate";** **};**

2.7.9 Mapping of Octet String Type

ASN.1 `OCTET STRING` is mapped to the IDL type **ASN1_OctetString** which is defined as **sequence<octet>**.

`OCTET STRING` literals can be either binary of hexadecimal strings. In either case, the value is expanded from left to right as a sequence of bits with octets taken in order from the left. The final octet is padded with 0 bits if necessary.

Octet string literals cannot be represented as constants in IDL and hence, are mapped in accordance with Section 2.7.2 on page 29.

2.7.9.1 Examples

Example 2-7 Mapping of Octet String Type

ASN.1	IDL
`G4FacsimilePage ::= OCTET STRING` `-- a sequence of octets conforming to` `-- Recommendation T.5 and T.6` `image G4FacsimilePage ::=` `  'H`	**typedef ASN1_OctetString G4FacsimilePageType;** **interface ConstValues {** **G4FacsimilePageType image();** **// "'H";** **};**

2.7.10 Mapping of ASN.1 String Types

Builtin ASN.1 string types can be divided into two categories: those that cannot contain value X'00; in them and those that can. Strings in the first category are simply mapped as **string**. Strings in the second category are further classified in two groups:

- strings containing single byte characters, mapped as **sequence<octet>**
- strings that support wide characters, mapped as **sequence** of wider types.

The IDL file ASN1Types.idl contains the following typedefs which can then be used directly in the translated IDL:

```
// These can contain X'00'
typedef sequence<octet> ASN1_GeneralString;
typedef sequence<octet> ASN1_GraphicString;

// These should support wide characters
typedef sequence<unsigned long> ASN1_UniversalString;
typedef sequence<unsigned short> ASN1_BMPString;

// These cannot
typedef string ASN1_IA5String;
typedef string ASN1_ISO646String;
typedef string ASN1_NumericString;
typedef string ASN1_PrintableString;
typedef string ASN1_TeletexString;
typedef string ASN1_T61String;
typedef string ASN1_VideotexString;
typedef string ASN1_VisibleString;
```

2.7.10.1 Mapping of Useful Type

ASN.1 provides a further set of useful sub-types of string. The main difference between these types and their base types is that they have well-defined Tags. Since the tag information is ignored in Specification Translation, these types are defined by the following IDL typedefs and they are used directly in generated IDL.

```
typedef ASN1_GraphicString ASN1_ObjectDescriptor;
```

`GeneralizedTime` and `UTCTime` are formatted string representations of time values (see reference ASN.1, clause 32). As strings, these values are not particularly easy to manipulate, for example, for comparison. For this reason PIDL functions are provided to manipulate these values converting to and from a more useful form corresponding to the POSIX timeval structure. Note that both `GeneralizedTime` and `UTCTime` optionally contain time zone information. Where this is not present, all operations assume that the time is local time and work accordingly.

2.7.10.2 PIDL for Time Access Functions

This PIDL provides routines to convert between string and struct and also a comparison routine. This routine takes two values, t1 and t2, and returns -1, 0 or 1 depending on whether t1 < t2, t1 = t2 or t1 > t2 respectively.

```
#include "ASN1Types.idl"

interface ASN1_GeneralizedTimeHandler // PIDL
{
   void set(inout ASN1_GeneralizedTime t, in unsigned long seconds,
      in long useconds, in long tzp);
   void get(in ASN1_GeneralizedTime t, out unsigned long seconds,
      out long useconds, out long tzp);
   short compare(in ASN1_GeneralizedTime t1, in ASN1_GeneralizedTime t2);
   void setFromString( inout ASN1_GeneralizedTime t, in string s );
   string asString(in ASN1_GeneralizedTime t );
}

interface ASN1_UTCTimeHandler // PIDL
{
   void set(inout ASN1_UTCTime t, in unsigned long seconds,
      in long useconds, in long tzp);
   void get(in ASN1_UTCTime t, out unsigned long seconds,
      out long useconds, out long tzp);
   short compare(in ASN1_UTCTime t1, in ASN1_UTCTime t2);
   void setFromString(inout ASN1_UTCTime t, in string s );
   string asString(in ASN1_UTCTime t );
}
```

The format of the time strings is the ASN.1 value notation.

2.7.11 Mapping of Object Identifier

ASN.1 `OBJECT IDENTIFIER` is mapped to IDL type **ASN1_ObjectIdentifier** which is defined as **string**:

```
typedef string ASN1_ObjectIdentifier;
```

The contents of an **ASN.1_ObjectIdentifier** will be one of the following:

- a CORBA scoped name — if the `OBJECT IDENTIFIER` is defined in a document
- the value of the ASN.1 object identifier in dot notation — if the `OBJECT IDENTIFIER` is not defined in any document.

Constants of this **ASN.1_ObjectIdentifier** type are represented as string literals which contain the CORBA scoped name instead of the original ASN.1 `OBJECT IDENTIFIER`. In addition to this IDL **const**, a **#pragma ID** will be generated with the value of the original object identifier literal in dot notation with the prefix of ‘‘OSIOID:’’. The use of this pragma makes this information available via the CORBA Interface Repository as the **RepositoryId** field for use during Interaction Translation or by any application which requires the OID value.

Note: This section of the document is likely to be impacted by subsequent work on Interaction Translation.

The Interaction Translation process will need to map between `OBJECT IDENTIFIER` and Management Meta-knowledge, , for example,.g IDL Typedefs, for resolving `ANY DEFINED BY`. This is left as an issue for the Interaction Translation document.

2.7.11.1 Examples

Example 2-8 Mapping of Object Identifier

ASN.1	IDL
`AttributeId ::= OBJECT IDENTIFIER`	**typedef ASN1_ObjectIdentifier AttributeIdType;**
`arfProbableCause OBJECT IDENTIFIER ::= { joint-iso-ccitt ms(9) smi(3) part2(2) standardSpecificExtension(0) arf(0) }` `adapterError OBJECT IDENTIFIER ::= { arfProbableCause 1 }`	**const ASN1_ObjectIdentifier arfProbableCause = "X721Att::arfProbableCause";** **#pragma ID arfProbableCause "OSIOID:2.9.3.2.0.0"** **const ASN1_ObjectIdentifier adapterError = "X721Att::adapterError";** **#pragma ID adapterError "OSIOID:2.9.3.2.0.0.1"**

2.7.12 Mapping of Any Type

ASN.1 `ANY`, `ANY DEFINED BY` are mapped to the IDL types **ASN1_Any**, **ASN1_DefinedAny** respectively. The defined IDL types are derived from the IDL **any** type. This is because there is no way of capturing the defining information available from `ANY DEFINED BY`. However, the IDL **any** type is capable of carrying arbitrary types and `ANY DEFINED BY` primarily references parameters types which are mapped as IDL types and hence may be carried - only the strong typing is lost. To preserve the information, the defining clause of an `ANY DEFINED BY` is mapped as a comment.

2.7.12.1 Examples

Example 2-9 Mapping of Any Type

ASN.1	IDL
`Attribute ::= SEQUENCE {` `  attributeId OBJECT IDENTIFIER,` `  attributeValue ANY DEFINED BY attributeId` `}`	**struct AttributeType {** **ASN1_ObjectIdentifier attributeId;** **ASN1_DefinedAny attributeValue;** **// defined by attributeId** **};**

2.7.13 Mapping of Tagged Type

Tag information is ignored during Specification Translation and a tagged type is mapped in the same way as an untagged type. Mapping tag information is an issue for Interaction Translation.

2.7.14 Mapping of External Type

ASN.1 `EXTERNAL` is mapped to the following IDL struct for simplicity:

```
struct ASN1_External {
        ASN1_ObjectIdentifier syntax;
        ASN1_DefinedAny data_value; // by 'syntax'
};
```

This mapping requires the gateway to fully resolve the encoded form of the External value it receives and re-encoded it using the CORBA **any** type. This ensures that no CORBA objects need support BER to handle this type. The Object Identifier for the abstract syntax is passed to provide semantic information in cases where the function cannot be determined solely by the type and also to enable the gateway to re-encode it when translating back to the OSI domain.

2.8 Mapping of Recursive Types

Unlike ASN.1, IDL does not support recursive type definition (except with sequences), and it is therefore necessary to take steps to address this issue. It is important to note that only direct recursion can be supported. Indirect (mutual) recursion is explicitly not supported.

When recursion is detected, the following rules are applied, in the order given below:

Rule 1: ASN.1 recursive type definitions are mapped by converting the type reference to a bounded IDL sequence of size 1. If a recursive reference occurs directly in a `SET OF` or `SEQUENCE OF`, then the bound is omitted (see Example 2-13 on page 46).

Rule 2: ASN.1 recursive type definitions are expanded in place (see example below).

When a recursion is encountered together with `OPTIONAL` functionality, Rule 2 is applied.

2.8.0.1 Examples

The following examples illustrate the application of these rules:

Example 2-10 Mapping of Recursive Types

ASN.1	IDL
`NameTree ::= SEQUENCE` `{` `  rdnInfo RDNINFO,` `  children SET OF NameTree` `}`	**struct NameTreeType {** ** RDNInfoType rdnInfo;** ** sequence <NameTreeType> children;** **}**
`NameTree ::= SEQUENCE` `{` `  rdnInfo RDNInfo,` `  children SET OF NameTree OPTIONAL` `}`	**struct NameTreeType {** ** RDNInfoType rdnInfo;** ** union childrenOpt switch (boolean) {** ** case TRUE: sequence <NameTreeType> value;** ** } children;** **};**
`A ::= SET OF B` `B ::= SEQUENCE { a  INTEGER,` `  b  A }`	**// It is not possible to translate this** **// example of mutual recursion**

2.9 Mapping of ASN.1 Constructed Types

Constructed types are those which are composed of several other types (constructed or not). ASN.1 supports the following constructors: `CHOICE`, `SET`, `SEQUENCE`, `SET OF` and `SEQUENCE OF`.

The approach to mapping constructed types is to map them according to Table 2-3. However, there are some additional complexities which must be addressed. To facilitate type manipulation by programmers, complex data structures will be simplified by creating new IDL intermediate types (see Section 2.9.1 on page 42). Some ASN.1 constructs must be resolved to name-type pairs and then translated. These include selection types and `COMPONENTS OF`. On the other hand, `OPTIONAL` elements are converted to an IDL union type which is then used as the type of the optional element. Subtype constraints based on the use of `WITH COMPONENTS` clause will be explained in the corresponding section for subtype mapping. It is anticipated that an intermediate ASN.1 type is generated, equivalent to the original one but without the subtype constraint, and then the common mapping is applied.

`CHOICE`, `SET` and `SEQUENCE` result from the aggregation of other types. From the ASN.1 grammar point of view, they are a list of elements. Each element is basically formed by an ASN.1 Named Type and some optional extra features.

In the case of `SET` and `SEQUENCE` type definitions, elements can be `OPTIONAL` or have default values. Also the `COMPONENTS OF` structure may be specified instead of the set of elements it represents.

In the case of the `CHOICE` type definition, none of these features are allowed , but instead the type of the Named Type can take the form of a selection type (which is not allowed for `SET` or `SEQUENCE`).

`SET OF` and `SEQUENCE OF` result from the aggregation of several instances of the same type which will be called an ‘‘item’’ in this document.

ASN.1 Construct	IDL Construct	Comments
`CHOICE`	**union/switch**	An enum is to be defined for switch case-constant.
`SET` `SEQUENCE`	**struct**	struct members are declared in the same order as they are declared in the ASN.1 type.
`Selection Type`	**mapped as type of selected element**	that is, the selection type is first transformed to obtain its real element type.
`OPTIONAL`	**union <mapped-type-name>Opt** **switch (boolean) {** **case TRUE: <type> value;** **};**	an extra type is created to indicate whether the element is present.
`SET-OF type` `SEQUENCE-OF type`	**sequence<mapped-type-name>**	

Table 2-3 Mapping of ASN.1 Type Constructors to IDL Type Constructors

Rules to map each element or item in a constructed type are provided below:

1. If element contains a selection type or it is a `COMPONENTS OF` clause, expand it to obtain the correponding element(s).

2. Obtain the name of the element or item. Apply rules for Anonymous Elements and Items if necessary. (See Section 2.9.2 on page 44).
3. Resolve name collisions with previous elements inside the scope of the constructed type.
4. Once the name of the element or item has been obtained, then obtain its corresponding type:
   - If recursion is found, apply rules described in Section 2.8 on page 40.
   - Check for composite types and create corresponding new types if necessary. Follow the rules described for Composite Types (see Section 2.9.1).
   - If `OPTIONAL` flag is present, create corresponding optional type following the rules described in Section 2.9.5.2 on page 49.
5. Once the name and the type of the element / item is obtained, create the corresponding IDL mapped element / item.
6. Finally, if default values were defined, create the corresponding constant as described in Section 2.9.5.3 on page 49.

2.9.1 Composite Types

As described above, an element or item always has a type definition, and depending of its type, special treatment may be needed. In such cases, these types are called Composite Types. Composite Types include the following:

- Constructed type:
  - `CHOICE`
  - `SET, SEQUENCE`
  - `SET OF, SEQUENCE OF`
- Enumeration
- Integer with Named Number List
- BitStrings with Named Bits.

When used in the elements or items of a constructed type, these are considered composite types. To avoid nested type declarations, they are extracted and a new IDL type is defined (unless recursion is detected). Thus a named IDL type definition is created.

The name of the new IDL type assignment is formed by concatenating the ASN.1 name of the container type (that is, the name of the constructed type that contains the current element / item without the ‘‘Type[<n>]’’ suffix) with the name of the element / item with an upper case first letter. Note that if it is an element, disambiguation with prior element names in the same constructed type must be done. Then the general rules for lexical mapping disambiguation of types are applied (that is, the corresponding Type[<n>] suffix is added).

Note that if recursion is detected, no external IDL named types are created, as described in Section 2.8 on page 40.

2.9.1.1 Examples

Example 2-11 Mapping of ASN.1 Constructed Types

ASN.1	IDL
`-- example with elements` `Bar ::= SEQUENCE {` `  -- definition of an composite type` `  paff SEQUENCE {` `  a INTEGER,` `  b VisibleString` `  },` `  -- definition of a composite type` `  dummy ENUMERATED {` `  one(1), two(2)` `  },` `  -- reference to primitive type,` `  c INTEGER` `}` `-- example with items (no composite type)` `Array ::= SET OF INTEGER`	**// mapping of the ASN.1 Bar type** **struct BarPaffType** **{** **ASN1_Integer a;** **ASN1_VisibleString b;** **};** **enum BarDummyType {one, two);** **struct BarType** **{** **BarPaffType paff;** **BarDummyType dummy;** **ASN1_Integer c;** **};** **//mapping of the ASN.1 Array type** **typedef sequence<ASN1_Integer> ArrayType;**

2.9.2 Anonymous Elements and Items

As stated before, an element is basically composed of an ASN.1 Named Type. This is composed of an identifier which is the name of the Named Type and the type itself. The presence of this name is optional in the ASN.1 grammar, though this is strongly discouraged (and is not allowed in ASN.1:1994).

IDL does not fully support similar constructs, therefore the mapping needs to resolve these issues.

An element without an explicit name is converted to an element with a name taking the form **elem<n>**, where <n> specifies the position of such element within the definition of the constructed type..

When dealing with an item, if it represents a composite type, a new type is created as described above. A name has to be provided because these definition types have no name associated with them. To provide an homogeneous way to solve the mapping, the identifier **item** will be used as if it were the name of an element whose type is the one defined by the item. Thus the item identifier will be used as the name of the item when Composite Type rules are to be applied (see Section 2.9.1 on page 42).

Note that when `SET OF` and `SEQUENCE OF` constructs have an item which is a Composite Type, it is always an anonymous item also. This is because items do not have an identifier name in the ASN.1 grammar.

2.9.2.1 Examples

Example 2-12 Anonymous Elements and Items

ASN.1	IDL
`-- example with anonymous elements` `A::=SEQUENCE {` `  INTEGER,` `  b INTEGER,` `  BOOLEAN,` `  ENUMERATED {` `  one(1), two(2)` `  }` `}` `-- example of anonymous items` `CorrelNotif ::= SET OF SEQUENCE {` `  correlNotif SET OF` `  NotificationIdentifier` `}`	**//mapping of ASN.1 A type** **enum AElem4Type {one, two};** **struct AType {** **ASN1_Integer elem1;** **ASN1_Integer b;** **ASN1_Boolean elem3;** **AElem4Type elem4;** **};** **// mapping of ASN.1 CorrelNotif type** **typedef sequence <NotificationIdentifierType>** **CorrelNotifItemCorrelNotifType;** **struct CorrelNotifItemType {** **CorrelNotifItemCorrelNotifType** **correlNotif;** **};** **typedef sequence<CorrelNotifItemType>** **CorrelNotifType;**

2.9.3 Mapping of Choice

ASN.1 `CHOICE` types are mapped to IDL **union** types. Since IDL unions are discriminated, an IDL **enum** type will be defined for the type of the discriminator. In order to do this it is first necessary to transform any composite types in the list of alternates. This is done via the mechanism described in Section 2.9.2 on page 44. Similarly `SELECTION` alternates are first resolved as per Section 2.9.4 on page 48.

The IDL identifier for the type of the discriminator is formed by adding a suffix ‘‘Choice’’ to the translated identifier of the `CHOICE` type. For each alternate, there will be one identifier in the discriminator type. This translated identifier is the identifier of the alternate suffixed by ‘‘Choice’’ and disambiguated within its IDL scope in the normal way.

Finally, the union type itself is constructed with one case for each alternate with label according to the enum type and the type and identifier according to the (intermediate) alternate. The resulting IDL looks like:

```
enum <choicetype>Choice {
        <translated-identifier>Choice,
        ..... // one for each alternate
}

union <choicetype> switch (<choicetype>Choice) {
        case <translated-identifier>Choice: <type> <translated-identifier>;
        ..... // one for each alternate
}
```

where **<choicetype>** is the name of the translated type (including ‘‘Type’’ suffix and numeric disambiguator if necessary).

2.9.3.1 Examples

The following shows examples of Choice and Recursion. Note that Composite Type Removal is not illustrated.

Example 2-13 Examples of Choice and Recursion

ASN.1	IDL
`Context ::= CHOICE {` `  id INTEGER,` `  data EXTERNAL` `}`	**enum ContextTypeChoice { idChoice, dataChoice };** **union ContextType switch (ContextTypeChoice) {** **case idChoice: ASN1_Integer id;** **case dataChoice: ASN1_External data;** **};**
`Filter::= CHOICE {` `  item [8] FilterItem,` `  and [9] IMPLICIT SET OF Filter,` `  or [10] IMPLICIT SET OF Filter,` `  not [11] Filter` `}`	**enum FilterTypeChoice {itemChoice, andChoice,** **orChoice, notChoice};** **union FilterType switch (FilterTypeChoice) {** **case itemChoice: FilterItemType item;** **case andChoice: sequence<FilterType> and;** **case orChoice: sequence<FilterType> or;** **case notChoice: sequence<FilterType,1> not;** **};** **-- note the replacement of the 'not' alternate** **-- (a self-referential type) with SET (SIZE(1)** **-- OF Filter a sequence of size 1 which results in** **-- an IDL sequence**

Here is a fragment of the X.721 | ISO/IEC 10165-2 (1992) definition of ProbableCause showing the constants `adapterError` and `applicationSubsystemFailure`.

```
Attribute-ASN1Module {
  joint-iso-ccitt ms(9) smi(3) part2(2) asn1Module(2) 1}
  DEFINITIONS IMPLICIT TAGS ::= BEGIN
    IMPORTS .......;
  ....
  adapterError ProbableCause ::= globalValue : { arfProbableCause 1 }
  applicationSubsystemFailure ProbableCause ::=
    globalValue : { arfProbableCause 2 }
  ..........
  ProbableCause ::= CHOICE {
    globalValue OBJECT IDENTIFIER, localValue INTEGER
 }
 ..........
END
```

This translates as (assuming that *X721Att* is the nickname for the Attribute module):

```
module X721Att {
        enum ProbableCauseTypeChoice {
                globalValueChoice,
                localValueChoice
        };

        union ProbableCauseType switch (ProbableCauseTypeChoice) {
                case globalValueChoice: ASN1_ObjectIdentifier globalValue;
                case localValueChoice:  ASN1_Integer localValue;
        };
        ...
        interface ConstValues {
                ProbableCauseType adapterError();
                ProbableCauseType applicationSubsystemFailure();
                ...
        };
};
```

Then an application program might access the values along the following lines:

```
// obtain access to the predefined ConstValue object for accessing the
// interface defined constants. This would probably be a pseudo object
X721Att::ConstValues constants (...);
// use constant in an assignment
ProbableCauseType pc; pc = constants.adapterError();
```

2.9.4 Mapping of Selection

The ASN.1 `SELECTION` construct provides a mechanism where a type may be defined by reference to a named element of a `CHOICE`. This is handled by first resolving the `SELECTION` to the selected <identifier,type> pair and using these instead. Where `SELECTION` is used to identify a type, it is resolved to the named type of the selected element only. When it is used as a name-type pair, it is resolved to the selected <identifier,type> pair.

2.9.4.1 Examples

Example 2-14 Mapping of Selection

ASN.1	IDL
`Attribute ::= CHOICE {` `  number INTEGER,` `  name VisibleString` `}` `Ident ::= CHOICE {` `  id number < Attribute,` `  name < Attribute` `}`	**// map the Attribute type** **enum AttributeTypeChoice {** **numberChoice,** **nameChoice** **};** **union AttributeType switch** **(AttributeTypeChoice) {** **case numberChoice:ASN1_Integer number;** **case nameChoice: ASN1_VisibleString name;** **};** **// map the Ident type** **enum IdentTypeChoice {** **idChoice,** **nameChoice__1** **};** **union IdentType switch (IdentTypeChoice) {** **case idChoice: ASN1_Integer id;** **case nameChoice__1: ASN1_VisibleString name;** **};**

2.9.5 Mapping of Sequence and Set

In ASN.1, a `SEQUENCE` type is an ordered collection of components which are typed and potentially named. `SET` types are identical except that the elements are unordered. Since IDL has no unordered record type, both of these types are handled identically and are mapped to an IDL struct. However, ASN.1 has some additional complexities in that it has additional mechanisms to extract elements of sub-records (via the `COMPONENTS OF <type>` construct) and also allows components to be declared as `OPTIONAL`.

2.9.5.1 COMPONENTS OF <type> Production

When a sequence or set type appears as `<type>` in `COMPONENTS OF <type>`, that sequence type is resolved to its list of elements before translating to IDL (that is, each component of the sub-record is extracted out effectively flattening the record structure).

2.9.5.2 OPTIONAL Components

If an element of a `SEQUENCE` or `SET` type is declared `OPTIONAL`, then that component may or may not be present. This is mapped to an IDL union with a boolean switch with only the True case having a value being that of the optional element. This new type is named by appending the type of the optional element with ‘‘Opt’’. The translator must take steps to avoid duplicating these types and this optional type should be generated at most once per module. For the ASN.1 clause:

```
SEQUENCE { <identifier> <type> OPTIONAL}
```

the resultant IDL would be:

union <mapped type name>Opt switch (boolean) {case TRUE: <mapped type name> value;};

2.9.5.3 DEFAULT Components

In ASN.1, clause 20.5 states that if DEFAULT occurs, the omission of a value of that type is exactly equivalent to insertion of the default value. This means that the value is mandatory, but that implementations need not put it on the wire. There is no direct equivalent in IDL hence access is simply provided to the default value; either as a constant or via a PIDL function returning the value. Where a programmer in the CORBA domain wants to use the default value, this constant or function call must be explicitly used. So if an ASN.1 `DEFAULT` value is present for a set or sequence element, an IDL constant will be generated, named by appending ‘‘Default’’ to the element’s identifier and defined as the given default value , for example:

const <mapped type name> <element-name>Default = <value>;

If the type of the element precludes the use of an IDL constant then the mechanism of Section 2.7.2 on page 29 applies. Disambiguation Rule 2, (see Section 2.4 on page 21), must be applied to the generated constant name.

2.9.5.4 Translating to IDL

Once any `COMPONENTS OF` and `OPTIONAL` clauses have been translated, an IDL struct is generated. Now each element is mapped as a member of the IDL struct in the order in which they appear in the resolved member list.

2.9.5.5 Examples

Example 2-15 Mapping of Sequence and Set

ASN.1	IDL
`T::= SEQUENCE {a Ta, b Tb, c Tc}` `E::= SEQUENCE {f1 E1, f2 T, f3 E3}`	**struct TType {TaType a; TbType b; TcType c;};** **struct EType {E1Type f1; TType f2; E3Type f3;};**
`T::= SEQUENCE {` `  a Ta,` `  b SEQUENCE {b1 T1, b2 T2,b3 T3}` `  c Tc }`	**struct TbType** **{T1Type b1; T2Type b2; T3Type b3;}; struct TType** **{TaType a; TbType b; TcType c;}; // The algorithm results in an additional // type TbType for the composite SEQUENCE // type in T.**
`W ::= SEQUENCE {x Wx, COMPONENTS OF` `  T, y Wy}`	**struct WbType {** **T1Type b1; T2Type b2; T3Type b3; };** **struct WType {** **WxType x;** **TaType a;** **WbType b;** **TcType c;** **WyType y; }; // The algorithm expands the // COMPONENTS OF part of W before // translating to IDL.**
`UserName ::= SET { -- SET treated as` `  -- SEQUENCE` `  personalName VisibleString,` `  countryName VisibleString OPTIONAL` `}`	**union ASN1_VisibleStringOpt** **switch (boolean) {** **case TRUE: ASN1_VisibleString value;** **}; struct UserName {** **ASN1_VisibleString personalName,** **ASN1_VisibleStringOpt countryName, };**
`A ::= SET {` `  a ENUMERATED {a, b},` `  b ENUMERATED {a, b} }`	**enum AaType {a, b}; enum AbType {a__1, b__1}; struct AType {** **AaType a;** **AbType b; };**

ASN.1	IDL
`Data ::= SEQUENCE {` `  replaceWithDefault` `  BOOLEAN DEFAULT FALSE,` `  defaultValue ValueSpecifier,` `  keyword SEQUENCE {` `  type-reference DefinedType,` `  field Identifier` `  } OPTIONAL,` `  createModifier BIT STRING {` `  withRefObject` `(0),withAutoNaming (1)` `  } }`	**const ASN1_Boolean replaceWithDefaultDefault=FALSE; struct DataKeywordType { DefinedTypeType type_reference, IdentifierType field, }; union DataKeywordTypeOpt switch (boolean) { case TRUE: DataKeywordType value; }; typedef ASN1_BitString DataCreateModifierType; const unsigned long withRefObject=0; const unsigned long withAutoNaming=1; struct DataType { ASN1_Boolean replaceWithDefault; ValueSpecifierType defaultValue; DataKeywordTypeOpt keyword; DataCreateModifierType createModifier; }**

The last example is worth looking at in more detail. First all composite types are handled by explicitly generating named types, **DataKeywordType**, **DataKeywordTypeOpt** and **DataCreateModifierType**.

Next, apply the rules for `OPTIONAL` and `DEFAULT`, generating a named value for the default and a union type in place of the `OPTIONAL` component, that is, **replaceWithDefaultDefault** and **DataKeywordTypeOpt**.

Now all the problem cases have been removed, the translation proceeds naturally resulting in the IDL as shown in the table above.

2.9.6 Mapping of Sequence Of and Set Of

ASN.1 `SEQUENCE OF` is an ordered list of items of the same type. `SET OF` is similar except that the list is unordered. Both of these are mapped into the IDL **sequence** construct. Again, first remove Composite Types by generating a new type on the same basis as Section 2.9.1 on page 42 and Section 2.9.2 on page 44. That is, using **item** as the name of the item in the concatenation to generate the new type name.

2.9.6.1 Examples

Example 2-16 Mapping of Sequence Of and Set Of

ASN.1	IDL
`RDNSequence::= SEQUENCE OF RDN`	**typedef sequence<RDNType> RDNSequenceType;**
`Status ::= SEQUENCE OF INTEGER {` `  initializationRequired (0),` `  notInitialized (1),` `  initializing (2),` `  reporting(3),` `  terminating (4)` `}`	**typedef ASN1_Integer StatusItemType;** **const StatusItemType initializationRequired = 0;** **const StatusItemType notInitialized = 1;** **const StatusItemType initializing= 2;** **const StatusItemType reporting = 3;** **const StatusItemType terminating = 4;** **typedef sequence<StatusItemType> StatusType;**
`A ::= SEQUENCE OF` `  SEQUENCE OF` `  INTEGER`	**typedef sequence<ASN1_Integer> AItemType;** **typedef sequence<AItemType> AType;**

2.9.7 Mapping of EmbeddedPDV and Character String Types

Mapping of `EmbeddedPDVType` and `CHARACTER STRING` from ASN.1:1994 to IDL is not addressed at this point. A good approach would be to use the associated `SEQUENCE` type for value definition and subtyping.

2.10 Mapping of Constraints and Subtypes

2.10.1 Mapping of Constrained Type

When a size constraint is used for either `SET OF` and `SEQUENCE OF` constructions, it is mapped to IDL as a bounded **sequence**, where the upper bound of the IDL sequence is determined by the upper bound on the permitted integer values specified in the constraint. In addition a constant giving the permitted values list is given. The resulting definition is along the lines of:

typedef sequence <<ItemType>, <upper-bound>> <type>;

where <ItemType> is the type of the `SEQUENCE OF/SET OF` item and <type> is the translated name of the compound type. For example:

```
T1 ::= SEQUENCE SIZE(0..10) OF ObjectInstance
T2 ::= SEQUENCE SIZE(1|3|5) OF INTEGER
```

maps to:

```
typedef sequence<ObjectInstanceType, 10> T1Type;
// T1Type SIZE(0..10)
typedef sequence<ASN1_Integer, 5> T2Type;
// T2Type SIZE(1|3|5)
```

Dynamic access to size information in order to ensure validity may be covered as part of Interaction Translation.

When a size constraint is applied to `BIT STRING` types, a constant integer literal is generated in the IDL. The identifier for the integer literal is generated by adding ‘‘_size’’ as a suffix to the translated parent type. This is because the mapping of `BIT STRING` to sequence of octet does not have the necessary granularity to usefully bound the sequence.

When a size constraint is applied to `OCTET STRING` or other `STRING` derived types, these are mapped as string or bounded sequences depending on the original type definition, where the upper-bound of the sequence in IDL is determined by the upper-end-value of the size constraint.

More complex type constraints are ignored by Specification Translation and may be addressed in Interaction Translation.

2.10.2 Mapping of Subtype Elements

ASN.1 supports different constructs from IDL and, in particular, subtypes are not easy to represent in IDL. Where possible, a fairly coarse grained approach has been defined. It is left as an implementation concern to check other value ranges and the details are left to the Interaction Translation.

2.10.2.1 Mapping of Value Range

When a value range is applied to a type `INTEGER` or a type derived from it, it will be mapped to one of several IDL integer types, depending on the value range, as shown in Table 2-4 on page 54. The default mapping of `INTEGER` with no subtyping is long.

Table 2-4 illustrates the selection of the IDL integer subtype based on the ASN.1 value range interval (Min,Max):

Type	Min	Max	Condition	IDL Native Type
ASN1_Unsigned16	0	2^16-1	none	unsigned short
ASN1_Unsigned	0	2^32-1	2^16 <= upper	unsigned long
ASN1_Unsigned64	0	2^64-1	2^32 <= upper	unsigned long[2]
ASN1_Integer16	-2^15	2^15-1	lower < 0	short
ASN1_Integer	-2^31	2^31-1	lower < 0 && (lower < -2^15 || 2^15 <=upper)	long
ASN1_Integer64	-2^63	2^63-1	lower < 0 && (lower < -2^31 || 2^31 <= upper)	long[2]

Table 2-4 Mapping ASN.1 INTEGER with Value Range to IDL Types

The lower and upper bounds for the ASN.1 `INTEGER` subtype must be within the minimum and maximum stated values for the corresponding IDL type (inclusive), and they should also fulfill the corresponding condition. In the table, the lower bound value of the ASN.1 value range `INTEGER` subtype is referred to as ‘‘lower’’, and the corresponding upper bound value as ‘‘upper’’

If the ASN.1 type has a value range `INTEGER` subtype specification that is beyond the stated limits for **ASN1_Unsigned64** and **ASN1_Integer64**, an **ASN1_Unsigned64** will be generated if the lower bound has a positive or 0 value, otherwise an **ASN1_Integer64** will be generated. The user will be informed of the translation limitation.

`REAL` is mapped as **double** in IDL. Value ranges are ignored.

2.10.2.2 Mapping of SingleValue

Even if a type is constrained to one or more single values, then this is still a type and translated accordingly. In addition, the set of allowable values is translated as a comment. The type is generated according to the same rules in Section 2.10.2.1 on page 53, with the ‘‘lower’’ and ‘‘upper’’ bounds being the extremes of the set of allowable values. Thus constructs such as:

```
A ::= INTEGER(1|3|5|7)
```

would be translated as

```
typedef ASN1_Unsigned16 AType; // AType (1|3|5|7)
```

2.10.2.3 Mapping of ASN.1 MIN and MAX

ASN.1 supports the literals MIN and MAX in value range constraints to denote the smallest resp. the biggest value of the parent type. The use of one of these special values will cause the compiler to select the appropriate **ASN1_Integer** type as defined in Section 2.10.2.1 on page 53.

2.10.2.4 Mapping of Permitted Alphabet

If the sub-typing restricts the alphabet of a type then an IDL constant will be generated. Its value will be the allowable alphabet and its identifier is formed based on the rules described for the mapping of Single Values, with the addition of the suffix ‘‘_permittedAlphabet’’.

2.10.2.5 Mapping of INCLUDES

`INCLUDES` contained subtypes are ignored. The translator will generate an unconstrained type.

2.10.2.6 Mapping of InnerTypeConstraints

Inner Type Constraints can be applied to `SET OF`, `SEQUENCE OF`, `SET`, `SEQUENCE` and `CHOICE` types or types formed from them by tagging.

Single Type Constraint (`WITH COMPONENT`) is ignored during Specification Translation. The specified type is generated without constraints.

Multiple Type Constraints (`WITH COMPONENTS`) are used for `SET`, `SEQUENCE` and `CHOICE`. This kind of constraint contains a list of constraints on the component of types of the parent type. The inner type to which the constraint applies is determined by the identifier.

When Full Specification is used there is an implied presence constraint of `ABSENT` on all inner types which can be constrained to be absent and which is not explicitly listed.

When Partial Specification is used and the parent type is a `SET` or `SEQUENCE` type, there are no implied constraints and any inner type can be omitted from the list. When an empty Presence Constraint is used, it is equivalent to a constraint `PRESENT` for a `SET` or `SEQUENCE` component marked `OPTIONAL`. In a Partial Specification no such constraint is imposed.

When Partial Specification is used and the parent type is a `CHOICE` type, there is implied presence constraint of `PRESENT` on all inner types which are not explicitly listed.

When Multiple Type Constraints are used to define a type from another type, the translation process generates a new type. Its components depend on the Multiple Type Constraints. The rules are as follows:

1. Define an intermediate ASN.1 type as that of that parent type. The type reference of the new type is that of the constrained type. The new ASN.1 type and the parent are of identical type (`SET`, `SEQUENCE` or `CHOICE`)
2. for each NamedConstraint in TypeConstraint of Full Specification:
   - 2.1 if the parent type is a `SET` or `SEQUENCE` then:
     - 2.1.1 if ValueConstraint is not empty, PresenceConstraint is empty and if the corresponding component type is `OPTIONAL`, then the corresponding component is copied from parent type to the new ASN.1 type. The ValueConstraint is applied as ConstrainedType of the form Type Constraint for the component.
     - 2.1.2 if PresenceConstraint is not empty:
       - 2.1.2.1 if `PRESENT` is used then the corresponding component is copied from parent type to the new ASN.1 type; if the ValueConstraint in ComponentConstraint is not empty then the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component. If the component has `OPTIONAL` label, this label will be dropped from the component in the new type.
       - 2.1.2.2 if `OPTIONAL` is used then the corresponding component is copied from parent type to the new ASN.1 type; if the ValueConstraint in ComponentConstraint is not empty then the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component. If the component does not have an `OPTIONAL` label then component will be labelled `OPTIONAL`.

2.2 if the parent type is a `CHOICE`:

2.2.1 if PresenceConstraint is not empty:

2.2.1.1 if `ABSENT` is used then the corresponding component is not copied from the parent type to the new ASN.1 type; if the ValueConstraint in ComponentConstraint is not empty then the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component.

3. For each NamedConstraint in TypeConstraint of Partial Specification:

3.1 if the parent type is a `SET` or `SEQUENCE` and:

3.1.1 if PresenceConstraint is not empty:

3.1.1.1 if `PRESENT` is used then the corresponding component is copied from parent type to the new ASN.1 type; if the ValueConstraint in ComponentConstraint is not empty then the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component. If the component has `OPTIONAL` label, this label will be dropped from the component in the new type.

3.1.1.2 if `OPTIONAL` is used then the corresponding component is copied from parent type to the new ASN.1 type; if the ValueConstraint in ComponentConstraint is not empty then the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component. If the component does not have an `OPTIONAL` label then component will be labelled `OPTIONAL`.

3.1.2 if ValueConstraint is not empty then the corresponding component is copied from parent type to the new ASN.1 type and the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component.

3.1.3 if both ValueConstraint and PresenceConstraint are empty then the corresponding component is copied from parent type to the new ASN.1 type and labelled `OPTIONAL`. If the ValueConstraint in ComponentConstraint is not empty then the ValueConstraint is applied as ConstrainedType of the form TypeConstraint for the component.

3.2 if the parent type is a `CHOICE`:

3.2.1 if PresenceConstraint is not empty then copy all the components of parent type to the component:

3.2.1.1 if `ABSENT` is used then remove the corresponding component from the new ASN.1 type.

4. Finally, map the new ASN.1 type to IDL as defined for any other ASN.1 type.

When the Inner Type Constraint is applied to a base type that is defined in a different module, it may occur that the types of some of the members of the new subtype are not known in the scope of the current module. In this case they must be explicitly imported from the module in which the base type is defined[3].

3. This module may have imported the type from yet another module, but there is no need to follow a chain back to the original definition.

2.10.2.7 Examples

Example 1

```
PDU ::= SET {
    alpha INTEGER,
    beta IA5String OPTIONAL,
    gamma SEQUENCE OF Parameter,
    delta BOOLEAN
}
TestPDU ::= PDU (WITH COMPONENTS { ..., delta (FALSE),
            alpha (MIN..<0) }) -- PartialSpecification
FurtherTestPdu ::= TestPDU (WITH COMPONENTS {... ,
            beta (SIZE (5|12)) PRESENT})
```

is treated as if it were the following equivalent ASN.1 form which can be translated to IDL via the usual rules.

```
TestPDU ::= SET {
    alpha INTEGER (MIN..<0),
    beta IA5String OPTIONAL,
    gamma SEQUENCE OF Parameter OPTIONAL,
    delta BOOLEAN (FALSE)
}
FurtherTestPdu ::= SET {
    alpha INTEGER (MIN..<0) OPTIONAL,
    beta IA5String (SIZE (5|12) ,
    gamma SEQUENCE OF Parameter OPTIONAL,
    delta BOOLEAN (FALSE) OPTIONAL
}
```

Example 2

```
TestPDU ::= PDU (WITH COMPONENTS { delta (FALSE),
            alpha (MIN..<0) }) -- FullSpecification
```

would change the equivalent ASN.1 form to:

```
TestPDU ::= SET {
    alpha INTEGER (MIN..<0),
    delta BOOLEAN (FALSE)
}
```

Now apply the SetType mapping scheme for ASN.1 to IDL on TestPDU and FurtherTestPDU.

```
Z ::= CHOICE { a A, b B, c C, d D, e E}
V::= Z (WITH COMPONENTS { ..., a ABSENT,
     b ABSENT}) -- TaU & TbU must be absent
W::= Z (WITH COMPONENTS { ...,
     a PRESENT }) -- TaU must be present
X::= Z (WITH COMPONENTS { a PRESENT })
     -- TaU must be present
Y::= Z (WITH COMPONENTS { a ABSENT, b, c})
     -- TaU, TdU and TeU must be absent
```

is treated as the following equivalent ASN.1 form:

```
V ::= CHOICE { c C, d D, e E}
W ::= CHOICE { a A, b B, c C, d D, e E}
X ::= CHOICE { a A}
W ::= CHOICE { b B, c C}
```

2.11 IDL Modules for Builtin ASN.1 Types

Two modules are declared automatically. The first, stored in a file, called ASN1Types.idl, declares all the IDL types used to translate builtin ASN.1 types. This file will be imported into all other IDL modules which do not, therefore, need to redefine these IDL types. The second, stored in a file called ASN1Limits.idl, contains implementation-defined values for some ASN.1 limits. It is automatically included with the ASN1Types.idl file, and therefore never needs to be explicitly imported.

The two files are contained in Section 12.1.1 on page 167 and Section 12.1.2 on page 168.

Preliminary Specification

Part 3:
GDMO to OMG IDL Translation Algorithm

The Open Group

Chapter 3

GDMO to CORBA-IDL Translation

In this chapter, the Specification Translation process is described in terms of inputs and outputs and a rough outline of the process is given. The process will be implemented via a compiler which operates on a set of input files and results in some output files. Since IDL definitions are processed in terms of files which determine the granularity and reusability of the IDL definitions, it is necessary to specify which definitions are generated and what files they are defined in. In addition, GDMO adds some complexities since GDMO specifications use full text names, for example, CCITT Rec. X.721 (1992) | ISO/IEC 10165-2 : 1992. Since such names are used to import parts of specifications, there must be a way for the translation process to access the files containing these specifications. In addition, it is desirable to be able to associate the resulting IDL files with the original GDMO to facilitate browsing and reuse. This is achieved by providing a "nickname database" which maps from the unique registered name of the GDMO document (or relevant Object Identifier) to a short nickname suitable for use as a filename base. This nickname is used to find imported files and to control the names of the generated IDL files.

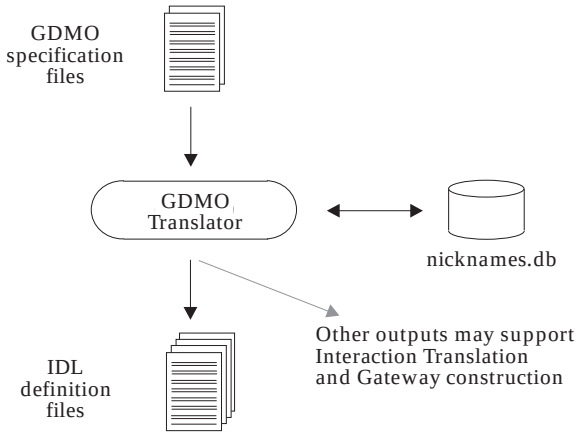

Figure 3-1 Inputs and Outputs for GDMO Specification Translation

Since nicknames will be used as the basis for naming files, generated IDL files will only be reusable in an environments where identical nickname databases are used. Therefore, it is desirable to make the nickname database as standard as possible (for example, have standard nicknames for all registered GDMO documents and their ASN.1 modules). In order to facilitate this, the following nickname selection method is recommended (but not mandatory):

- For documents, use the ITU recommendation number where it exists. For example, DMI would be called *X721*.
- For ASN.1 modules, the nickname is formed by taking the nickname of the document in which the module occurs, followed by the first three characters of the module label. For example, the Attribute module of X.721 would have the nickname *X721Att*.
- Where more than one module label in a document has the same initial three letters, append a number to the nicknames for the second and subsequent module labels to disambiguate. This means that the first module would have no numeric suffix, the second colliding module would have the suffix ‘‘1’’, the third ‘‘2’’ and so on.

As it is illegal to modify the existing contents of a standard in this context, it is assumed that the nickname always refers to the latest version of the standard. If, for any reason, parts of the original standard are modified in a revision, the last 2 digits of the revision year can be appended to the nickname.

3.1 Outline of Translation Algorithm

The algorithm for translating a GDMO specification to a CORBA-IDL specification is fully detailed in Chapter 4. It assumes that GDMO will not allow the use of CMIP version 2 to support extensible types. The basic approach is as follows:

1. Generate an OMG IDL file for each ASN.1 module that is contained in a given GDMO document and name it with the nickname that has been assigned to the module (see Chapter 2 on page 15).
2. Generate an OMG IDL file for each GDMO document and name it with the nickname assigned to the document. That file will contain a module that is named the same as the file and contains interfaces generated for each managed object class template defined in the GDMO document and other information described below.
3. Generate up to two OMG IDL files containing interfaces for handling Notifications via push and/or pull model when requested.

It should be noted that the mechanism used to handle Notifications is designed to work with OMG Event Services (see reference **COSS**). It enables the use of typed or untyped events delivered via push or pull models. This gives maximum flexibility to implementors to select the most appropriate mechanisms.

3.2 File Names and IDL Modules

The output of the above translation is a set of IDL modules and interfaces. These must be organised into files in a way which facilitates reuse and effective generation of code by the CORBA IDL compiler. Thus the Specification Translation process results in potentially many IDL files. During that process, the following rules determine the number and content of each file:

- Each file will start with a comment identifying the GDMO document from which it was generated.
- Definitions contained in IDL generated files must be enclosed within **#ifndef ... #endif** directives as shown below and will enclose mappings of GDMO template definitions.

  ```
  #ifndef _<capitalised_nickname>_IDL_
  #define _<capitalised_nickname>_IDL_
  module <nickname> {

  <idl-mapped-macro-invocations>

  };
  #endif /* _<capitalised_nickname>_IDL_ */
  ```

- For each GDMO document, generate an IDL file named **<nickname>.idl** containing an IDL module named **<nickname>**, and two IDL file named **<nickname>_N.idl** and **<nickname>_NP.idl** to contain interfaces for notifications. Here **<nickname>** is the nickname that has been assigned to the document.
- For each ASN.1 module that is contained in a given GDMO document, translate it as described in Chapter 2 on page 15.

  It is permissible for an implementation to only generate the mapping for selected classes from an input document. However, if this is done, all the appropriate superclasses must be included. In addition, the full translation must be performed before the selected classes are output in order to ensure that the name disambiguation rules are applied correctly.
- For each ASN.1 module defined in a given GDMO document, add the directive:

  ```
  #include "<module_nickname>.idl"
  ```

  to the corresponding IDL file, where <module_nickname> is the nickname that has been assigned to the defined ASN.1 module.

This approach results in many IDL files being generated. However, the _N and _NP files are generated to provide implementation choice and could all be omitted if untyped event delivery mechanisms are used. In addition, the other files need only be generated once and may be reused by many translated documents. This in turn allows more efficient code generation by the IDL compiler. Since files are of a finer granularity, documents which build on earlier definitions may extract only the portions that they use by import rather than requiring the whole referenced document to be reviewed. As a side benefit, collision between identically named ASN.1 modules may be avoided by using different nicknames for those modules. Module nicknames are not guaranteed to be unique unless they have the unique **<nickname>** of their containing GDMO document as a prefix. Another side-effect is that references to types imported from other documents need only be scoped by the module's nickname, which must be unique in the environment.

3.2.1 Standard Files for Specification Translation

The specification translation assumes the existence of a number of standard files containing base definitions and classes. These are as follows:

ASN1Types.idl	contains the base definitions for translating ASN.1 types (see Section 2.11 on page 58).
ASN1Limits.idl	contains the definitions for ASN.1 limits (see Section 12.1.2 on page 168).
OSIMgmt.idl	contains everything which is needed to use CMIS services based on the mapping defined in this document.

In addition, the following standard files, should be delivered with an implementation of the GDMO to IDL translator:

X219Rem.idl	ROSE errors
X227ACS.idl	ACSE types
X711CMI.idl	CMIP types
X501Inf.idl	Information framework

3.2.2 Example

Using the recommended nickname convention:

- the nickname for the standard X.721 ASN.1/GDMO document (see reference **GDMO**), will be *X721*
- *X721Att*, *X721Not*, *X721Par* will be the nicknames assigned to the different ASN.1 modules,

resulting in the following IDL files:

Example 3-1 File X721Att.idl

```
// Generated from X721.gdmo
// X721Att.idl file:

#ifndef _X721ATT_IDL_
#define _X721ATT_IDL_

#include <ASN1Types.idl>
#include "X711CMI.idl"
#include "X227ACS.idl"
#include "X501Inf.idl"

module X721Att {
  <exports-imports-clause-mapping>
  <ASN.1-definition-mapping-list>
};

#endif /* _X721ATT_IDL_ */
```

Example 3-2 File X721Not.idl

```
// Generated from X721.gdmo
// X721Not.idl file:

#ifndef _X721NOT_IDL_
#define _X721NOT_IDL_

#include <ASN1Types.idl>
#include "X721Att.idl"
#include "X711CMI.idl"

module X721Not {
  <exports-imports-clause-mapping>
  <ASN.1-definition-mapping-list>
};

#endif /* _X721NOT_IDL_ */
```

Example 3-3 File X721Par.idl

```
// Generated from X721.gdmo
// X721Par.idl file:

#include <ASN1Types.idl>
#ifndef _X721PAR_IDL_
#define _X721PAR_IDL_

module X721Par {
  <exports-imports-clause-mapping>
  <ASN.1-definition-mapping-list>
};

#endif /* _X721PAR_IDL_ */
```

Example 3-4 File X721.idl

```
// Generated from X721.gdmo
// X721.idl file:

#ifndef _X721_IDL_
#define _X721_IDL_

#include <OSIMgmt.idl>
#include "X721Att.idl"
#include "X721Not.idl"
#include "X721Par.idl"

module X721 {
  ..........................
};

#endif /* _X721_IDL_ */
```

Example 3-5 File X721_N.idl

```
// Generated from X721.gdmo
// X721_N.idl file:

#ifndef _X721_N_IDL_
#define _X721_N_IDL_

#include "X721Not.idl"

module X721_N {
  interface Notifications {
  ..........................
  };
};

#endif /* _X721_N_IDL_ */
```

Example 3-6 File X721_NP.idl

```
// Generated from X721.gdmo
// X721_NP.idl file:

#ifndef _X721_NP_IDL_
#define _X721_NP_IDL_

#include <OSIMgmt.idl>
#include "X721Not.idl"

module X721_NP {
  interface PullNotifications {
  ..........................
  };
};

#endif /* _X721_NP_IDL_ */
```

Chapter 4

Mapping GDMO Templates to IDL Interfaces

For every GDMO managed object class template, an IDL interface is defined which supports the operations exported by members of the corresponding managed object type. In addition, a module is defined which contain interfaces supporting notifications. Figure 4-1 describes the inheritance hierarchy of management IDL interfaces derived from MANAGED OBJECT CLASS template definitions. The base interface for all IDL management interfaces is **ManagedObject** (which implicitly inherits from **CORBA::Object** as do all IDL interfaces). No attribute or operation is defined for **ManagedObject** yet. It is provided as a place holder for attributes and operations needed to implement managed object interfaces in a generic way.

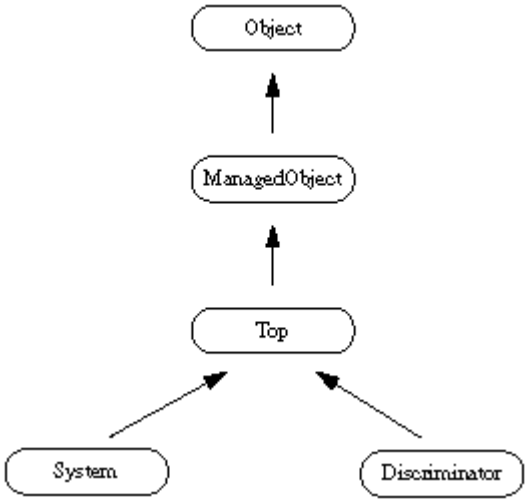

Figure 4-1 Inheritance Hierarchy of Interfaces of Managed Object Classes

,iX "managed object class"

Since all generated IDL interfaces inherit from **ManagedObject**, the necessary functionality to support CMIS request and responses can be provided through the implementation of the **OSIMgmt::ManagedObject** interface. It is envisaged that the **ManagedObject** interface will provide support for such things as scoping, filtering, and attribute groups, by declaring operations on a per object basis for use by the gateway to fulfill scoped and filtered requests. This issue is deferred to the definition of Interaction Translation and of the **ManagedObject** interface in a separate document.

GDMO identifiers are subject to the same lexical translation and disambiguation rules defined for ASN.1 identifiers in Section 2.4 on page 21. As ‘‘/’’ is a legal character in GDMO identifiers (but not in ASN.1), it will be translated as the literal string ‘‘_SLASH_’’.

GDMO template order is determined by its use inside Managed Object classes. Templates not used (for example, Name Bindings, Parameters, Attributes, Packages, etc, that are declared but are not used or not accessible from Managed Object classes in the document) are not mapped. Imported templates are considered as if they were local to the current document for lexical purposes.

With the exception of managed object classes, each template `REGISTERED AS` clause is not mapped into the IDL files generated by the specification translation.

4.1 Error Handling

Both CMIP and CORBA provide mechanisms to handle errors. CMIP provides a basic set of errors (augmented by ROSE errors) one of which, `processingFailure`, may carry additional data defined in `PARAMETER` templates with context `SPECIFIC-ERROR`. This polymorphism is handled via the `ANY DEFINED BY` clause, which in turn is mapped to the IDL **ASN1_DefinedAny** type (which is derived from the IDL **any** type). CORBA provides a two-tiered exception handling mechanism consisting of standard exceptions (which carry very little data) and user exceptions which may carry arbitrary data types as defined in the IDL. In order to preserve the data capability, user defined exceptions are provided for each of the CMIP errors with data types as translated from the CMIP module (see reference **CMIP**). This is included by all translated objects via the file OSIMgmt.idl. An initial version of the OSIMgmt.idl file is contained in Section 12.2.0 on page 169. It will further defined in the Interaction Translation document.

This solution will allow any invokers of an IDL operation to handle CMIP errors through exceptions. To enhance IDL readability, macros are provided defining the set of exceptions which each operation type might raise. A convenient side-effect of this is that `PARAMETER`s in general and `SPECIFIC-ERROR`s in particular need only be mapped to a type-declaration, so that a suitable TypeCode would be generated for use with IDL any.

4.2 Mapping Managed Object Templates to IDL

During translation from a managed object class template to an IDL interface, mandatory and conditional packages of the managed object class are distinguished only through the use of comments. The attributes, actions and notifications of conditional packages are translated in the same way as those of mandatory packages. Whether a conditional package is present or not is an implementation issue. When an attempt is made to access a conditional package that is not present, the implementation should raise the CORBA standard system exception **NO_IMPLEMENT**.

For each template in a GDMO specification, the method for translating from that template to the appropriate IDL definitions is described. This is done recursively starting from the `MANAGED OBJECT CLASS` template shown in Table 4-1.

```
<class-label> MANAGED OBJECT CLASS
    [DERIVED FROM <class-label> [, <class-label>]* ; ]
    [CHARACTERIZED BY <package-label> [, <package-label>]* ; ]
    [CONDITIONAL PACKAGES <package-label> PRESENT IF
    <condition-definition>
        [, <package-label> PRESENT IF <condition-definition> ]* ; ]
REGISTERED AS object-identifier ;
```

Table 4-1 Managed Object Class Template Structure

A managed object class, maps to a management IDL interface with the same name plus additional interfaces which support notifications. Interface **top** inherits from **ManagedObject** and #includes OSIMgmt.idl. All other management interfaces multiply inherit from the interfaces corresponding to the managed object classes in the `DERIVED FROM` clause and must #include the files where IDL interface definitions corresponding to those managed object class templates have been generated. Then for each package, `CONDITIONAL` or otherwise, do the following steps (output to management IDL interface unless otherwise stated):

- Generate IDL comments containing the name of the package and, if it is `CONDITIONAL`, the `PRESENT IF` text.
- List the names of the attributes, actions and notifications as comments.
- Copy the behaviours of the package as comments.
- Copy the attribute-group information as comments.
- Declare a set of IDL operations for each attribute in each package as described in Section 4.3 on page 73. Copy any behaviour text and/or attribute specification relating to the attribute as comments in order to include information about default-values, permitted-values, required-values, etc.
- For each Action, declare an IDL operation, as described in Section 4.5.2 on page 75. If the action has a `WITH REPLY SYNTAX` clause, then generate the IDL necessary to support the handling of multiple replies.
- For each Notification, copy the behaviour text of the corresponding `NOTIFICATION` template as a comment. In addition, generate IDL operations in the _N and _NP modules (as described in Section 4.6.2 on page 78).

The `REGISTERED AS` clauses are translated as:

```
// <class-label> ... REGISTERED AS { a b c d e f }

#pragma ID <mapped-class-label> "OSIOID:a.b.c.d.e.f"
```

This allows the object identifier to be re-used as an Interface Repository ID.

This is only done for Managed Object Class templates.

4.3 Mapping an Attribute as a Set of IDL Operations

Associated with each Attribute in a GDMO package is a property-list which identifies the access mechanisms to the attribute. These may be `GET`, `REPLACE`, `GET-REPLACE`, `ADD`, `REMOVE`, `ADD-REMOVE` or `REPLACE-WITH-DEFAULT`. In addition CMIP may raise a number of different errors (possibly with parameters given in `SPECIFIC-ERROR` parameters) such as `processingError` or `illegalValue`. These are mapped to IDL as User Exceptions so that the additional data may be carried and this is the main reason why GDMO attributes are not mapped directly to IDL attributes; access operations on IDL attributes cannot raise user exceptions. All attribute operations may raise the same set of errors defined by the macro **ATTRIBUTE_ERRORS** from OSIMgmt.idl defined in Section 4.1 on page 70. Each attribute maps to one or more of five possible IDL operations:

```
<type>  <attribute-label>Get() raises (ATTRIBUTE_ERRORS);
void    <attribute-label>Set(in <type> value)
        raises (ATTRIBUTE_ERRORS);
void    <attribute-label>Add(in <type> value)
        raises (ATTRIBUTE_ERRORS);
void    <attribute-label>Remove(in <type> value)
        raises (ATTRIBUTE_ERRORS);
<type>  <attribute-label>SetDefault()
        raises (ATTRIBUTE_ERRORS);
```

where **<type>** is the ASN.1 for the Attribute type translated to IDL via the mechanisms defined in Chapter 2 on page 15. Note that **<attribute-label>** is the attribute identifier after disambiguation (that is, it may have a suffix of the form <__n>). Note that **Add** and **Remove** operations manipulate sets and hence take values of the same type as the attribute itself.

For each attribute property in the list, IDL operations are generated according to Table 4-2. Each operation is generated at most once for each attribute in an interface.

Property	IDL Operations Required
`GET`	**<attribute-label>Get**
`REPLACE`	**<attribute-label>Set**
`GET-REPLACE`	**<attribute-label>Get and <attribute-label>Set**
`ADD`	**<attribute-label>Add**
`REMOVE`	**<attribute-label>Remove**
`ADD-REMOVE`	**<attribute-label>Add and <attribute-label>Remove**
`REPLACE-WITH-DEFAULT`	**<attribute-label>SetDefault**

Table 4-2 Mapping Attribute Properties to IDL Operations

4.4 Mapping Parameters to IDL Types

In GDMO, a PARAMETER is provided in order to resolve ANY DEFINED BY clauses in ACTIONS, NOTIFICATIONS and the CMIP Error processingFailure. The ASN.1 ANY DEFINED BY construct maps to an IDL **any** and to use this requires a suitable TypeCode to be available. Since a TypeCode is automatically generated when a type is defined in IDL, the corresponding **typedef** in IDL is simply generated. Applications that wish to use the PARAMETER clauses can then use the TypeCode to insert their data into the relevant IDL **any** field. As an additional programming aid, a comment is also generated for each parameter template.

```
<parameter-template> ::=
        <parameter-label> PARAMETER
CONTEXT <context-type> ;
<syntax-or-attribute-choice> ;
[ BEHAVIOUR <behaviour-definition-label>
        [ , <behaviour-definition-label>]* ; ]
[ REGISTERED AS object-identifier ] ;

<context-type> ::= <context-keyword> | ACTION-INFO |
        ACTION-REPLY | EVENT-INFO |
        EVENT-REPLY | SPECIFIC-ERROR
<context-keyword> ::= <type-reference>.<identifier>
<syntax-or-attribute-choice> ::= WITH SYNTAX <type-reference> |
        ATTRIBUTE <attribute-label>
```

Table 4-3 Parameter Template Production

The mapping proceeds by translating the syntax of the attribute to an IDL type. If the ATTRIBUTE construct is used, the IDL type is the type of the reference attribute. Behaviour definitions are transcribed as comments, as is the CONTEXT type information. The mapping generates a new IDL type declaration, where the new type is named after parameter label and the type is the translation of the WITH SYNTAX or ATTRIBUTE constructs.

4.4.1 Examples

Example 4-1 Mapping Parameters to IDL Types

GDMO	IDL
not-running PARAMETERS CONTEXT SPECIFIC-ERROR; WITH SYNTAX ActionModule.ServerState;	**typedef ActionModule::ServerState** **not_runningType;** **// SPECIFIC-ERROR**

4.5 Mapping Actions to IDL Operations

In GDMO objects, action templates may have parameters and replies. Where actions have a reply syntax, objects have the option of using multiple replies (that is, returning a sequence of PDUs, each of the type given in the reply syntax, containing part of the reply). The behaviour clause of an action template indicates whether multiple replies may be used. Multiple replies allow data to be returned as it becomes available and have been used for monitoring progress. Whilst these are not widely used and could easily be replaced by notifications, it was regarded as necessary to provide for this capability. To do this, an additional interface which operates in the reverse direction is necessary (that is, the object instance would be the client of the interface and the manager would be the server). To achieve this, the managed object raises an exception to indicate that it will generate multiple replies.

As an additional complexity, actions may be either confirmed or unconfirmed, and this may be selected at run-time.

```
<action-template> ::=
    <action-label> ACTION
        <behaviour-clause>
        <action-mode>
        [PARAMETERS <param1>[, <param>]* ; ]
        <action-info-syntax>
        <action-reply-syntax>
    REGISTERED AS object-identifier ;

<action-mode> ::= MODE CONFIRMED ; | empty
<action-info-syntax> ::= WITH INFORMATION SYNTAX <type-reference> ; | empty
<action-reply-syntax> ::= WITH REPLY SYNTAX <type-reference> ; | empty
```

Table 4-4 Action Template Production

4.5.1 Mapping of Action Parameters

Any parameters contained in the PARAMETERS clause of the action template are mapped as comments.

4.5.2 Mapping to an Operation on the Primary Interface

Each ACTION template is mapped to at least one operation in the management IDL interface. The name of the mandatory operation is the <action-label> of the ACTION template which may need disambiguating via the usual rules. It is possible for the name of an action to clash with an operation generated for an attribute. In this case, the attribute operation takes precedence and the name of the action is changed according to the usual rules (that is, add a suffix of the form <__n>). The operation may raise any of the CMIP errors associated with ACTION invocation (defined by the macro **ACTION_ERRORS** from OSIMgmt.idl in Section 4.1 on page 70). If the action template has a WITH INFORMATION SYNTAX clause, the operation has an in parameter with type translated from the WITH INFORMATION SYNTAX clause named **actionInfo**. If the action template has a WITH REPLY SYNTAX clause, then the action may result in multiple replies (this is determined by the text in the behaviour clause but this information is not available to the compiler). In this case, the operation has a return type which is the translation of the WITH REPLY SYNTAX clause and may raise an additional user exception named **UsingMR** to indicate if it is using multiple replies.

4.5.3 Handling Multiple Replies

The handling of multiple replies is a matter for the Interaction Translation specification. The general mechanism to handle multiple replies does not need any special treatment from the GDMO to IDL translation process except for the **UsingMR** exception raised by the operations derived from the ACTION translation (see Section 4.5.2 on page 75). If the manager is to handle multiple replies, it must detect the **UsingMR** exception and take the appropiate steps.

4.5.4 Examples

Example 1

Consider a simple case of ACTION template:

```
suspend ACTION
    MODE CONFIRMED
    PARAMETERS not-supported, not-running;
    WITH INFORMATION SYNTAX ActionModule.Suspend;
    WITH REPLY SYNTAX ActionModule.ServerState;
REGISTERED AS { 1 2 3 4 9 };
not-supported PARAMETER
    CONTEXT SPECIFIC-ERROR;
    WITH SYNTAX ActionModule.ReasonCode;
REGISTERED AS { 1 2 3 4 10 };
not-running PARAMETER
    CONTEXT SPECIFIC-ERROR;
    WITH SYNTAX ActionModule.ServerState;
REGISTERED AS { 1 2 3 4 11 };
```

This would translate in the primary interface as follows:

```
typedef ActionModule::ReasonCodeType not_supportedType; // SPECIFIC-ERROR
typedef ActionModule::ServerStateType not_runningType; // SPECIFIC-ERROR

ActionModule::ServerStateType suspend(in ActionModule::SuspendType actionInfo)
  raises (ACTION_ERRORS, UsingMR);

// PARAMETERS not-supported, not-running;
```

Example 2

If the ACTION template definition for suspend was:

```
suspend ACTION
    PARAMETERS not-supported, not-running, time-delay;
    WITH INFORMATION SYNTAX ActionModule.Suspend;
REGISTERED AS { 1 2 3 4 9 };
```

The operation definitions in the primary interface would look like this:

```
void suspend (in ActionModule::SuspendType actionInfo)
  raises (ACTION_ERRORS);

// PARAMETERS not-supported, not-running, time-delay;

oneway void suspendUnconfirmed (in ActionModule::SuspendType actionInfo);
```

4.6 Mapping Notifications to IDL Operations

The interfaces described below are contained in several separate files. This allows applications that do not wish to make use of some or all of these features to simply discard the files. It is, however, required that the files are generated.

Mapping of notifications must enable the use of typed or untyped events via the push or pull models. The mapping generates two sets of operations in the interfaces of the **<nickname>_N** and **<nickname>_NP** modules. However, due to issues surrounding the GDMO/CMIP notification mechanisms, the issue of how notification handling is setup is deferred to the Interaction Translation document. It is recommended that a comment be generated, listing the notifications on the primary interface.

CORBA event channels do not allow for operations which have a return type. In GDMO it is allowable for notifications to have reply syntaxes; however this is not believed to be used. In view of this it was decided not to map reply syntaxes for notifications. For similar reasons, no user exceptions are raised by notification operations.

```
<notification-template> ::=
    <notification-label> NOTIFICATION
        <behaviour-clause>
        [PARAMETERS <param1>[, <param>]* ; ]
        <notification-info-syntax>
        <notification-reply-syntax>
    REGISTERED AS object-identifier

<notification-info-syntax> ::= WITH INFORMATION SYNTAX
<type-reference>
        <notification-attribute-ids> ; |
        empty

<notification-attribute-ids> ::= AND ATTRIBUTE IDS
        <attribute-id> [, <attribute-id>]* |
        empty

<notification-reply-syntax> ::= WITH REPLY SYNTAX <type-reference> ; |
        empty
```

Table 4-5 Notification Template Production

To maximise the flexibility of events, two alternatives are supported. Pure untyped events are supported by the CMIP header data as well as an any to carry the data. This results in the necessary type code being generated. Alternatively, interfaces supporting fully typed events are also generated in both push and pull flavours. Note that operations supporting typed delivery of notifications are produced in two interfaces per document (one for push and one for pull). These interfaces are called Notifications and PullNotifications and are scoped within a module with the same name as the file to ensure uniqueness.

4.6.1 Mapping of Event Parameters

Any parameters contained in the PARAMETERS clause of the notification template which have a context type of EVENT-REPLY or SPECIFIC-ERROR are ignored because reply syntaxes are not supported. Any parameter having a context type of EVENT-INFO are mapped as comments.

4.6.2 Mapping to Operations in Notification Modules

To support typed events under the push model, operations are generated in the _N module. Since notification can be confirmed or unconfirmed as selected at run-time, two operations are generated, one for confirmed events and one for unconfirmed. The unconfirmed event is defined as **oneway** in IDL. These operations have the following *in* parameters for the CMIP header components:

in ASN1_ObjectIdentifier sourceObjectClass,
in X711CMI::ObjectInstanceType sourceObjectInstance,
in ASN1_ObjectIdentifier eventType,
in X711CMI::ASN1_GeneralizedTimeOpt eventTime,

followed by an *in* parameter of type translated from the WITH INFORMATION SYNTAX clause named **notifInfo**. The operations are named as **<notificaton-label>** and **<notification-label>Unconfirmed** respectively. Both operations have the return type **void**.

Similarly, the pull model is supported by two operations generated in the _NP module. These have the following *out* parameters for the CMIP header components:

out ASN1_ObjectIdentifier sourceObjectClass,
out X711CMI::ObjectInstanceType sourceObjectInstance,
out ASN1_ObjectIdentifier eventType,
out X711CMI::ASN1_GeneralizedTimeOpt eventTime,

followed by an *out* parameter of type translated from the WITH INFORMATION SYNTAX clause named **notifInfo**. The operations are named as **pull_<notification-label>** and **try_<notification-label>** and have **void** and **boolean** return types respectively. Note that with the pull model, event delivery is inherently confirmed.

To support the untyped event mechanism, a suitable IDL type will be defined in the OSIMgmt.idl file. However, definition of such a type is postponed to Interaction Translation.

4.6.3 Example

The following NOTIFICATION template:

```
objectCreation NOTIFICATION
    BEHAVIOUR objectCreationBehaviour;
    WITH INFORMATION SYNTAX Notification-ASN1Module.ObjectInfo
        AND ATTRIBUTE IDS
        sourceIndicator SourceIndicator,
        attributeList AttributeList,
        notificationIdentifier NotificationIdentifier,
        correlatedNotifications CorrelatedNotifications,
        additionalText AdditionalText,
        additionalInformation AdditionalInformation;
REGISTERED AS { joint-iso-ccitt ms(9) smi(3) part2(2) notification(10) 6}
```

would be translated as follows:

```
// in X721_N module
  void objectCreation (
    in ASN1_ObjectIdentifier sourceObjectClass,
    in X711CMI::ObjectInstanceType sourceObjectInstance,
    in ASN1_ObjectIdentifier eventType,
    in X711CMI::ASN1_GeneralizedTimeOpt eventTime,
    in X711Not::ObjectInfoType notifInfo);

  oneway void objectCreationUnconfirmed (
    in ASN1_ObjectIdentifier sourceObjectClass,
    in X711CMI::ObjectInstanceType sourceObjectInstance,
    in ASN1_ObjectIdentifier eventType,
    in X711CMI::ASN1_GeneralizedTimeOpt eventTime,
    in X711Not::ObjectInfoType notifInfo);

// in X721_NP module
  void pull_objectCreation (
    out ASN1_ObjectIdentifier sourceObjectClass,
    out X711CMI::ObjectInstanceType sourceObjectInstance,
    out ASN1_ObjectIdentifier eventType,
    out X711CMI::ASN1_GeneralizedTimeOpt eventTime,
    out X711Not::ObjectInfoType notifInfo);

  boolean try_objectCreation (
    out ASN1_ObjectIdentifier sourceObjectClass,
    out X711CMI::ObjectInstanceType sourceObjectInstance,
    out ASN1_ObjectIdentifier eventType,
    out X711CMI::ASN1_GeneralizedTimeOpt eventTime,
    out X711Not::ObjectInfoType notifInfo);
```

4.7 Resolving Inheritance Collisions

In CORBA IDL, an attribute or operation cannot be inherited from more than one interface, whereas in GDMO such multiple declaration is allowed. This can arise from both single and multiple inheritance. In order to solve issues of inheritance collision, the algorithm described here will produce the correct output.

The following steps should be applied in order to each managed object class of the input GDMO document. Before processing a class, all its ancestors in the inheritance tree must be processed, and the results of that processing taken into account:

1. Check for repetition of attribute/operation names across all ancestors in inheritance tree.
2. Revise ancestor name collisions by revising the inheritance hierarchy, if required.
3. Add only non-redundant attribute/operation methods from the current object.
4. Map the revised derived managed object class to IDL interface.

Implementation of this (or an equivalent) algorithm is optional. If a translator does not implement such an algorithm, it will not be able to successfully translate GDMO documents that cause inheritance collisions.

This algorithm was chosen because it maintains the inheritance tree as large as possible, and it also results in the generation of cleaner IDL code.

4.7.1 Examples

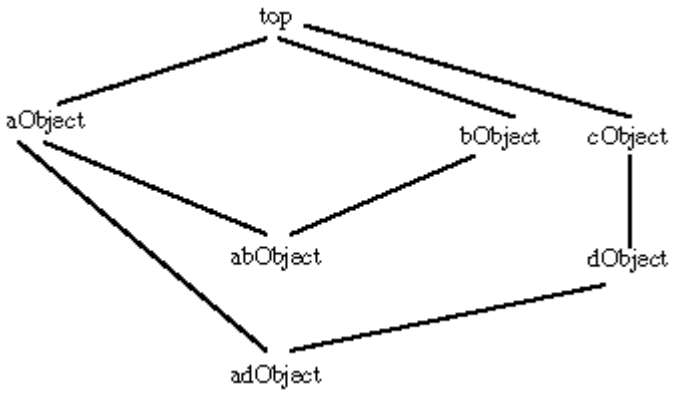

Figure 4-2 Inheritance Hierarchy Showing Name Collision

Figure 4-2 describes the inheritance hierarchy of managed object classes, where the same attributes are declared in inherited interfaces by the following GDMO:

```
aObject MANAGED OBJECT CLASS
    DERIVED FROM "dmi":top;
    CHARACTERIZED BY aObjectPkg1 PACKAGE
        ATTRIBUTES aAttr1 GET, aAttr2 GET; ;
REGISTERED AS {xxx ?};

bObject MANAGED OBJECT CLASS
    DERIVED FROM "dmi":top;
    CHARACTERIZED BY bObjectPkg1 PACKAGE
        ATTRIBUTES aAttr2 GET-REPLACE, bAttr1 GET; ;
REGISTERED AS {xxx ?};

cObject MANAGED OBJECT CLASS
    DERIVED FROM "dmi":top;
    CHARACTERIZED BY cObjectPkg1 PACKAGE
        ATTRIBUTES cAttr1 GET;;
REGISTERED AS {xxx ?};

dObject MANAGED OBJECT CLASS
    DERIVED FROM cObject;
    CHARACTERIZED BY dObjectPkg1 PACKAGE
        ATTRIBUTES aAttr1 GET-REPLACE, dAttr1 GET; ;
REGISTERED AS {xxx ?};

abObject MANAGED OBJECT CLASS
    DERIVED FROM aObject, bObject;
    CHARACTERIZED BY abObjectPkg1 PACKAGE
        ATTRIBUTES abAttr1 GET; ;
REGISTERED AS {xxx ?};

adObject MANAGED OBJECT CLASS
    DERIVED FROM aObject, dObject;
    CHARACTERIZED BY acObjectPkg1 PACKAGE
        ATTRIBUTES cAttr1 GET-REPLACE; ;
REGISTERED AS {xxx ?};

-- Assume for simplicity that all attributes have WITH SYNTAX "AttrType"
```

The IDL interfaces for aObject, bObject, cObject and dObject are:

```
interface aObject : X721::top
{
   AttrType aAttr1Get() raises (ATTRIBUTE_ERRORS);
   AttrType aAttr2Get() raises (ATTRIBUTE_ERRORS);
};

interface bObject : X721::top
{
   AttrType aAttr2Get() raises (ATTRIBUTE_ERRORS);
   void aAttr2Set(in AttrType value) raises (ATTRIBUTE_ERRORS);
   AttrType bAttr1Get() raises (ATTRIBUTE_ERRORS);};

interface cObject : X721::top
{
   AttrType cAttr1Get() raises (ATTRIBUTE_ERRORS);
};

interface dObject : cObject
{
   AttrType aAttr1Get();
   aAttr1Set(in AttrType value) raises (ATTRIBUTE_ERRORS);
   AttrType dAttr1Get() raises (ATTRIBUTE_ERRORS);
};
```

If abObject is mapped as inheriting from interfaces aObject and bObject, then there is collision on attribute aAttr2. In this case the inheritance tree is modified such that abObject is inherited only from aObject and include the attribute and operation of bObject that is not present in aObject and as well as its own attributes and operations to get:

```
interface abObject : aObject
{
   void aAttr2Set(in AttrType value) raises (ATTRIBUTE_ERRORS);
   AttrType bAttr1Get() raises (ATTRIBUTE_ERRORS);
   AttrType abAttr1Get() raises (ATTRIBUTE_ERRORS);
};
```

If adObject is mapped as inheriting from interfaces aObject and dObject, then there is a collision on attribute aAttr1. In this case the inheritance tree is modified such that adObject is inherited from aObject and cObject. In the revised object include the attribute and operation of dObject that is not present in aObject and cObject and its own attributes and operations to get:

```
interface adObject : aObject, cObject
{
   void aAttr1Set(in AttrType value) raises (ATTRIBUTE_ERRORS);
   AttrType dAttr1Get() raises (ATTRIBUTE_ERRORS);
   void cAttr1Set(in AttrType value) raises (ATTRIBUTE_ERRORS);
};
```

Figure 4-3 describes the modified inheritance hierarchy of managed object classes shown in Figure 4-2 on page 80.

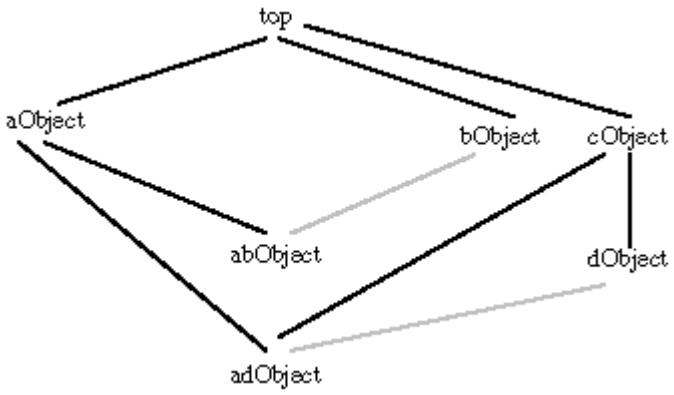

Figure 4-3 Revised Inheritance Hierarchy of Managed Object Classes

Preliminary Specification

Part 4:
OMG IDL to GDMO/ASN.1 Translation Algorithm

The Open Group

Chapter 5

OMG IDL to GDMO/ASN.1 Translation

5.1 Mapping CORBA IDL to GDMO/ASN.1

Mapping from CORBA IDL to GDMO/ASN.1 is simpler in that GDMO/ASN.1 has more expressive power than CORBA IDL.

A CORBA IDL specification consists of one or several input files. A specification may contain global declarations and a possibly empty sequence of IDL modules which themselves can contain declarations, interfaces and nested modules which allow for further nesting. This specification does not support global declarations and requires that all declaration be contained within a module.

A GDMO document consists of a sequence of GDMO templates and/or ASN.1 modules. No further formal structuring is possible.

5.1.1 Outline of the Translation Algorithm

The translation alorithm defined in this chapter applies the following translation rules. It generates:

- one or more GDMO documents per CORBA IDL specification file; each outermost IDL module in the IDL specification file will have its own GDMO document
- one MOC template per interface
- one attribute template per attribute and interface
- one action template per operation and interface
- one ASN.1 type per IDL datatype.

Nested IDL modules are unravelled to the outermost module.

No templates are generated for attribute groups, behaviour and notifications. Package templates are only generated in-line.

5.1.2 Generated GDMO/ASN.1 Documents

A separate GDMO document is generated for each outermost module of the IDL file.

The translator moves every nested IDL declaration to the GDMO document scope. All IDL datatype declarations are moved into a single ASN.1 module.

Any resulting collisions between identifiers are resolved by appending a suffix "-<n>" where <n> is an increasing integer starting with 1, as necessary within the mapped module.

The CORBA IDL input accepted by the IDL to GDMO translator may contain one or a sequence of outermost IDL modules, which may contain declarations of data types, constants, interfaces and nested modules.

- For each outermost module, a GDMO document (that is, a file) is generated. If the input contains a sequence of outermost modules, several GDMO documents will be generated in accordance with the algorithm explained below. The name of the generated GDMO document will be the same as the name of the mapped outermost IDL module.

- For each outermost module, all the data type definitions of all nested modules in the input are put into a single ASN.1 module with the nickname `<IDL-module-name>-ASN1`.
- All interface definitions of all nested modules are put into a single GDMO document (file), as explained below.
- Any cross module references definitions used within an IDL specification to be translated must be replaced in the corresponding GDMO document with the proper externalized GDMO document reference, or ASN.1 `IMPORT` clause.

5.1.3 JIDM GDMO Base Document

A single file with the new GDMO document ''JIDMbaseDocument'' with the necessary basic definitions is part of this specification:

corbaObject	A Managed Object Class used as the inheritance root for all IDL derived managed object classes. (It is a subclass of *X721:Top*).
corbaSystem	A Managed Object Class used as the local naming root for all IDL derived managed object instances. (It is a subclass of *corbaObject*).
corbaNameBinding	A Name Binding for *corbaObject* and subclasses under *corbaSystem*.
jidmRoot	An ASN.1 `OBJECT IDENTIFIER` used as root or prefix for the generated (see Section 5.1.6 on page 89).

5.1.4 Lexical Translation

The alphabet for IDL specifications is based on the ISO Latin-1 standard (reference **ISO 8859-1**) with a rich set of graphic characters. The character set of GDMO and ASN.1 is more restricted.

For the use in the generated specification itself, the rules for identifiers and constant value declarations are of major concern (see below). The permissible alphabets for strings are mapped as close as possible to the appropriate ASN.1 string types.

5.1.5 Translation of IDL Identifiers to GDMO and ASN.1 Labels

IDL identifiers must start with one alphabetic character and may consist of any sequence of any alphabetic or digit characters or the underbar (''_'') character. Identifiers that differ only in the case (upper or lower) of the letters are regarded as equal.

For GDMO and ASN.1, identifiers may consist of letters and digits as in IDL but may contain the dash (''-'') instead of the underbar. As additional requirement, GDMO and ASN.1 expect a lower case first letter for structure members and value specifications, and an upper case first letter for ASN.1 Type productions. All GDMO templates must have names beginning in lower case.

The translation algorithm replaces each underbar by a dash and applies the rule for the first letter to all identifiers.

IDL type identifiers are unique within their interfaces with respect to module naming scope. As the nested structure of modules and interfaces will be lost in the generated ASN.1 files, conflicts between these type identifiers may occur. In this case, the string ''-[i]'' is appended to the second and subsequent colliding identifiers with ''i'' incremented by one upon each usage. This is similar to the approach taken for the GDMO to CORBA IDL mapping.

For each generated GDMO document, with a given nickname, the translator must produce a text output file, named **<nickname>.ide**, which contains a line for each IDL identifier translated into corresponding definitions in that GDMO document. Each line of the **<nickname>.ide** file will

have the following three fields, using ‘‘,’’ as a field delimiter:

<IDL fully scoped identifier name>,<gdmo identifier used>,<object identifier in string dot notation>

The first and second fields are always present. The third field may not be present if no object identifier has been generated for the IDL identifier.

For ASN.1 production names defined in the ASN.1 module `<IDL-module-name>-ASN1`, the second field will be scoped as in GDMO `SPECIFIC-ERROR`.

5.1.6 Allocation of Object Identifiers

The `OBJECT IDENTIFIERS` for each of the GDMO templates are automatically allocated, given an `OBJECT IDENTIFIER` value for the IDL specification in which the definitions are contained.

Part of compiling an IDL file into GDMO will require specifying an `OBJECT IDENTIFIER` root for the IDL file. This `OBJECT IDENTFIER` root may be obtained through any valid registration authority for ASN.1 `OBJECT IDENTIFIER` values.

As an alternative, a mechanism[4] is described below which will allow a unique identifier to be easily generated. In many IDL environments, a generator of DCE uuid values (uuidgen) is available. DCE uuid values are globally unique 128 bit values. Corresponding values for `OBJECT IDENTIFIERS` based upon DCE uuid values can be generated below a common root id supplied by The Open Group:

```
xOpenJIDMrootOid OBJECT IDENTIFIER ::=
  { iso(1) member-national-body(2) bsi(826) disc(0) xopen(1050) jidm(9) }
```

The actual OID must be based on appending as the next level of the `OBJECT IDENTIFIER` a uuid generated by the invoker of the translator making use of the DCE tool **uuidgen**. It should be noted that this approach guarantees unique OIDs under The Open Group JIDM root OID without a further need for registration. In addition, once a single uuid has been generated, the user is free to allocate the resulting sub-tree as they desire:

```
jIDM<document>oid OBJECT IDENTIFIER ::= { xOpenJIDMrootOid (<uuid>) }
```

5.1.7 Use of Object Identifiers

Given an `OBJECT IDENTIFIER` value ‘‘jIDM<document>oid’’ for the IDL specification, separate `OBJECT IDENTIFIER` values can be generated for each outermost module in the IDL file. The value 0 is reserved, and each successive outermost module takes the next available number, starting with 1 for the first module in the sequence of outermost modules. For example:

```
jIDMdocMod1         OBJECT IDENTIFIER     ::= { jIDM<document>oid 1 }
  .
  .
  .
jIDMdocMod8         OBJECT IDENTIFIER     ::= { jIDM<document>oid 8 }
```

Note there is a unique `OBJECT IDENTIFIER` value for each GDMO document generated from the IDL file. Using jIDMdocMod1 as an example root for the GDMO document corresponding to the first outermost module of the jIDMdocument, it is possible to further derive the following categories, which serve as registration nodes for each category of IDL defined items within the

4. It is possible that additional mechanisms of generating unique OIDs under a common root may be defined in the future.

module. It should be noted that these assignments are fully in line with the GDMO standard (see reference **GDMO**, clause 6.4.3):

```
jIDMdocMod1ASN1Module          OBJECT IDENTIFIER ::= { jIDMdocMod1 2 }
jIDMdocMod1ManagedObjectClass  OBJECT IDENTIFIER ::= { jIDMdocMod1 3 }
jIDMdocMod1Parameter           OBJECT IDENTIFIER ::= { jIDMdocMod1 5 }
jIDMdocMod1NameBinding         OBJECT IDENTIFIER ::= { jIDMdocMod1 6 }
jIDMdocMod1Attribute           OBJECT IDENTIFIER ::= { jIDMdocMod1 7 }
jIDMdocMod1Action              OBJECT IDENTIFIER ::= { jIDMdocMod1 9 }
jIDMdocMod1IDLtype             OBJECT IDENTIFIER ::= { jIDMdocMod1 11 }
```

At a given nesting level, defined items of a given category (for example, action) are sequentially assigned integers corresponding to relative position in the specification.

The jIDMdocMod1IDLtype is supplied as a local root for assigning `OBJECT IDENTIFIER` values for each type defined in the module, for use in the ASN.1 `ANY` type (see below) as identifiers, without resorting to the free form of IDL **any** representation.

Given an interface *foo* which is the first interface in the first module, its OID value would be:

```
foo OBJECT IDENTIFIER ::= { jIDMdocMod1ManagedObjectClass 1 }
```

Given an operation *foobar* which is the second operation defined in the first module of the specification, its OID value will be:

```
foobar OBJECT IDENTIFIER ::= { jIDMdocMod1Action 2 }
```

5.1.8 Translation of Comments

Mapping of comments is optional. Because the nested modules are unravelled in the translation process, there might be some reordering of the IDL which may fail to re-order the associated comments, thus rendering the translated comments of doubtful value.

IDL uses the same syntax for comments as C++ does: either a full line of text starting from ‘‘//’’ or any portion of the input text between ‘‘/*’’ and ‘‘*/’’.

Both forms are translated into the ASN.1 comment ‘‘--’’.

5.1.9 Translation of Preprocessor Directives

IDL uses the same syntax and inventory of preprocessor directives as C++. As GDMO/ASN.1 does not support any preprocessor directives, the IDL files should be preprocessed according to the normal rules established in the CORBA specification (see reference **CORBA**).

The identifier file associated with each GDMO document must be consulted to determine the proper GDMO reference or ASN.1 Import for each of the external definitions used within the translated IDL specification file.

5.1.10 Translation of CORBA IDL

Table 5-1 on page 91 gives an overview on the mapping of CORBA IDL entities into GDMO templates.

CORBA IDL	GDMO/ASN.1
Interface (multiple inheritance, no override)	`MANAGED OBJECT (multiple inheritance)`
Operation (in, out, and in-out parameters, optional return result value, one-way option)	`ACTION WITH INFORMATION SYNTAX` `  SEQUENCE {(in, and in-out params)}` `WITH REPLY SYNTAX` `  SEQUENCE{return result value,` `  (in-out and out params)}`
Attribute (get value, set value)	`ATTRIBUTE (GET, or GET-REPLACE)`
Exception (standard and user defined with parameters)	`SPECIFIC ERRORS (with parameters carried by ROER processing failure)`

Table 5-1 Translation of IDL Definitions

5.1.10.1 Translation of IDL Interfaces

CORBA interfaces are mapped directly to Managed Object Class templates. Each IDL interface is translated into a GDMO managed object class template as in the following:

```
<interface-name>[-<n>] MANAGED OBJECT CLASS
  DERIVED FROM <super-classes>
  [CHARACTERIZED BY <interface-name>Package[-<n>] PACKAGE
  [ATTRIBUTES <list of attributes and properties>;]
  [ACTIONS <list of operation names>;]]
REGISTERED AS { <managed-object-class-oid> };
```

where `<super-classes>` are those of the interface or ‘‘jIDMbaseDocument:corbaObject’’ if none is specified.

There is no need to generate a package if an interface contains neither attributes nor operations.

The interface-name, and its corresponding inline package name, is given the ‘‘-<n>’’ suffix, where <n> increases from 1, as required to disambiguate any colliding interface names caused by the nested module unravelling.

The properties of the attributes depend on the *readonly* token of the IDL attribute. If it is present, the attribute gets the property GET, otherwise the attribute is read-write and it gets the property GET-REPLACE.

If an attribute has its base type as an IDL sequence, then the mapped ASN.1 attribute syntax will be SEQUENCE OF, thus the list of properties may not be extended by ADD-REMOVE, which is only allowed for set-valued attributes. (A set-valued attribute is one whose syntax is SET OF. CMIS has the further restriction that set-valued attributes cannot repeat values (see reference **CMIS** clause 3.1.5.9).

The GDMO properties SET TO DEFAULT, INITIAL VALUE and PERMITTED VALUE cannot be generated because IDL does offer comparable information. All translated attributes and actions must include the specific error parameter **corbaStandardException**.

5.1.10.2 Translation of IDL Attributes

Each IDL attribute declared in an IDL interface will be translated to a GDMO attribute template:

```
<attribute-name>[-<n>] ATTRIBUTE
  WITH ATTRIBUTE SYNTAX <IDL-module-name>-ASN1.<attribute-name>[-<n>];
  MATCHES FOR <matching rules>;
REGISTERED AS { <attribute-oid> };
```

where `<matching rules>` is determined based on characteristics of the attribute's base type:

`EQUALITY`	for every type
`ORDERING`	for integer, real and string types
`SUBSTRING`	for string types
`SET-COMPARISON`	cannot be used, since an IDL attribute will never be a set-valued attribute
`SET-INTERSECTION`	cannot be used, since an IDL attribute will never be a set-valued attribute.

Any collisions in attribute names within the module are resolved using sequential ''-<n>'' suffixes, where <n> increases from 1, as required to disambiguate.

5.1.10.3 Translation of IDL Operations

Each IDL attribute declared in an IDL interface will be translated into a GDMO ACTION template:

```
<operation-name>[-<n>] ACTION
    [WITH INFORMATION SYNTAX
        <IDL-module-name>-ASN1.<operation-name>[-<n>];]
    [WITH REPLY SYNTAX
        <IDL-module-name>-ASN1.<operation-name>[-<n>];]
REGISTERED AS { <action-oid> };
```

The type of an action information (reply) is a structure of type SEQUENCE which is composed of fields corresponding to all *in* and *inout* arguments (*retVal*, *inout* and *out* arguments). The name of each field the name of the corresponding formal parameter of the operation. The name of the field that represent the operation return value is **retVal**.

The exceptions raised on the IDL operations must be mapped to specific errors for the ACTION when used in the package definition.

5.1.10.4 Translation of IDL Exceptions

IDL exceptions are defined and are used in the context of a method invocation as form of error return. This sematics is different from that of GDMO notifications which spontaneously indicate an event (not only an error) from the remote side.

Thus, each IDL exception declared in an IDL interface will be translated into a GDMO `PARAMETER` template with the context of `SPECIFIC-ERROR`:

```
<exception-name>[-<n>] PARAMETER
    CONTEXT SPECIFIC-ERROR
    WITH SYNTAX <IDL-module-name>-ASN1-<exception-name>[-<n>]
REGISTERED AS { <parameter oid> };
```

The exception name is appended with a ''-<n>'' suffix, with <n> increasing from 1 as required to disambiguate naming collisions within the outermost IDL module.

The **corbaStandardException** specific error parameter, defined in the JIDM base GDMO document (see Section 5.5 on page 103) must be added as a `SPECIFIC-ERROR` parameter for all translated IDL operations and attributes.

5.2 Name Bindings

As the IDL world does not know about name bindings and introduces naming/containment hierarchies through the use of COSS:naming services, the mapping defines a flat naming hierarchy, that is, every generated GDMO MOC will be named by a common root object called **JIDM:corbaSystem** with a naming attribute defined as:

```
CORBAName ::= SEQUENCE OF SEQUENCE {
                  id GRAPHIC STRING,
                  val GRAPHIC STRING
              }
```

The name binding template is defined as follows:

```
corbaSystem-corbaObject   NAME BINDING
    SUBORDINATE OBJECT CLASS  corbaObject  AND SUBCLASSES;
    NAMED BY
        SUPERIOR OBJECT CLASS  corbaSystem  AND SUBCLASSES;
    WITH ATTRIBUTE  corbaName;
REGISTERED AS { jIDMbaseDocumentNameBinding  1 };
```

5.3 Mapping CORBA IDL Data Type Definitions

The following sections explain how CORBA IDL interface definitions can be mapped onto GDMO / ASN.1.

5.3.1 Translation of IDL Data Types

5.3.1.1 Translation of IDL Base Types

The generated common ASN.1 modules should import the common IDL base types as defined in the jIDMbaseDocument ASN1Module, as required by the translated IDL types. This will allow the translator to not include ranges each time it translates an integer and thus renders the generated code clearer.

For each IDL attribute type, operation argument type, and for each function return value type, generate an ASN.1 typed as in the following:

- The identifier of an attribute type is **<attribute-name>[-<n>]** which must start with an upper-case letter.
- The identifier of a typedef is of the form **<IDL-type-name>[-<n>]** which must start with an upper-case letter.
- The identifier for the information syntax of an action is of the form **<action-name>Info[-<n>]** which must start with an upper-case letter.
- The identifier for the reply syntax of an action is of the form **<action-name>Reply[-<n>]** which must start with an upper-case letter.
- The identifier of a exception is of the form **<IDL-exception-name>[-<n>]** which must start with an upper-case letter.
- The identifier of each constant declaration is **<constant-name>[-<n>]** which must start with a lower-case letter.

CORBA IDL	GDMO/ASN.1
integer (long, short, unsigned long, unsigned short)	`Integer (with optional subtype_spec)`
floating point (float, double)	`Real`
char **string**	`GraphicString` `GraphicString`
octet	`OCTET STRING (size constraint of 1)`
boolean	`BOOLEAN`
any	`ANY DEFINED BY or` `SEQUENCE{ typecode, anyValue }` `See discussion below.`
string	`GraphicString`
object <or any other object reference>	`ObjectInstance (X.500 Distinguished name)` `Use one of the names from the OMG name service, and convert to X.500 format`

Table 5-2 Type Mapping

5.3.1.2 Translation of IDL Type Constructors

CORBA IDL	GDMO/ASN.1
struct	SEQUENCE
union	Choice (with ASN.1 TAGs)
enum	ENUMERATED
array	SEQUENCE OF (with sizeConstraint subtype)
sequences	SEQUENCE OF (with optional SizeConstraint subtype for IDL bounds)

Table 5-3 Constructor Mapping

When translating complex IDL structures (structures containing structures, sequence ..., or sequence of sequence ...), no generation of intermediate types is required. Keep one ASN.1 type per one IDL type.

When multiple declarators are used, generate a type/constant for each declarator.

5.3.2 Examples for Constructed Types

```
module m{
   typedef struct A{
      sequence <string <10>, 15 >  u;
      long  v[10][20];
      struct B {
         long x;
         short y;} w;
      sequence<long,10>  z[3];
      union B switch (long){
         case 1: short x;
         case 2: enum C{ red, black } y;
      } q;
   };
};
```

the translation to ASN.1 is:

```
A ::= SEQUENCE {
          u SEQUENCE SIZE(15) OF GraphicString(SIZE(10) ),
          v SEQUENCE SIZE(10) OF SEQUENCE SIZE(20) OF Long,
          w SEQUENCE {
              x Long ,
              y Short },
          z SEQUENCE SIZE(3) OF SEQUENCE SIZE(10) OF Long,
          q CHOICE {
              x Short,
              y ENUMERATED { red(0),
              black(1)}
          }
      }
```

5.3.3 Provision of CORBA ANY Type in GDMO/ASN.1

The CORBA IDL **Any** type is able to contain any structured value. In addition to the value, the type of the value and its components can be determined during run time and be interpreted during access. This is similar to the ASN.1 `ANY DEFINED BY` type, but in contrast, the type information is outside of the scope of the ASN.1 specifications. For the translation, a mapping from IDL **Any** to ASN.1 `ANY DEFINED BY` is needed.

5.3.3.1 ASN.1 Syntax for CORBA ANY Type Parameters

Given the `OBJECT IDENTIFIER` allocation scheme provides an `OBJECT IDENTIFIER` for CORBA IDL Typedefs in CORBA specifications which are translated to GDMO, the generated `OBJECT IDENTIFIER` can be used to identify each IDL TypeDef. This object identifier can serve as the typeCode in an ASN.1 representation of CORBA any, based on the `ANY DEFINED BY` construct.

The resulting mapping is to the *CORBAANY* production:

```
CORBAANY ::= SEQUENCE {
                  identifiedTypeOID OBJECT IDENTIFIER,
                  identifiedTypeValue ANY DEFINED BY identifiedTypeOID
}
```

5.3.3.2 Free Form Representation of CORBA Any Parameters

For cases where CORBA `ANY` is used to carry a parameter for which there is no translated CORBA IDL module in existence to provide the identifiedTypeOID, a recursive representation of type codes and `ANY` values can be used. For such uses of CORBA `ANY`, this specification registers an ASN.1 `OBJECT IDENTIFIER VALUE`:

```
freeFormCORBAANYSyntax OBJECT IDENTIFIER ::= {jIDMbaseDocumentASN1Module 2 1}
```

for use in the *CORBAANY* production, described in Section 5.3.3.1, to signal that the identifiedTypeValue has the free form `ANY` type syntax described in Section 5.3.3.2.

The ASN.1 syntax production for a free form CORBA `ANY` value, can be used for CORBA `ANY` when the associated CORBA Type definition is not within a translated CORBA IDL specification (that is, does not have a registered `OBJECT IDENTIFIER` value for the type. Although more complex, this alternative conveys information on the contents of Any values which have types defined at run time (that is, not defined within a translated CORBA specification).

The ASN.1 syntax is specified in the *FreeCORBAAny* production in the ASN.1 module in this Chapter.

5.3.3.3 Free Form Type Code Representation in ASN.1

Conceptually, a CORBA IDL TypeCode is a recursive structure, which allows any IDL defined type to be represented. It is used to unravel the *anyValue* field in the Any type.

The syntax of Type codes is specified by the *TCKind*, *TCParm* and *TypeCode* productions in the ASN.1 module in this Chapter.

To represent recursive types (for example, `struct foo { sequence <foo> bar }`) a special TCKind value (–1) is used. Its parameter list includes an integer which indicates the number of upward nest levels required to access the recursively referenced type code.

The parameter list is populated for the various KINDs as indicated in Table 5-4 on page 98, which allows for all CORBA 2.0 valid **any** parameter types to be passed using the ASN.1 syntax for Type code.

5.3.3.4 Free Form CORBA Any Value Representation in ASN.1

The value for the Any type is represented as a nested choice structure, which can be unraveled using the type code information. The ASN.1 syntax is specified in the *CORBAAnyValue* production specified in the ASN.1 module at the end of this chapter.

KIND	**PARAMETER LIST "{...}"** *(denotes repetition of contents)*
tk-null tk-void tk-short	‘‘none‘‘ ‘‘none‘‘ ‘‘none‘‘
tk-long tk-ushort tk-ulong	‘‘none‘‘ ‘‘none‘‘ ‘‘none‘‘"
tk-float tk-double tk-boolean	‘‘none‘‘ ‘‘none‘‘ ‘‘none‘‘
tk-char tk-octet tk-any	‘‘none‘‘ ‘‘none‘‘ ‘‘none‘‘
tk-TypeCode tk-principal	‘‘"none‘‘ ‘‘none‘‘
tk-interfaceRef	string(repository ID), string (name)
tk-struct tk-except	string(repository ID), string(name), integer(count) { string(member name), TypeCode(member type) }
tk-union	string(repository ID), string(name), TypeCode(discriminant type), integer(default used), integer(count), { anyVal(label-value), string(member name), TypeCode(member type) } /*Note: type of label anyVal determined by third parameter, discriminant type */
tk-enum	string(repository ID), string(name), integer(count), { string(member name }
tk-string tk-sequence	integer(max length) /* for unbounded strings, this value is zero */ TypeCode(element type), integer(bounds)
tk-array tk-alias	TypeCode(element type), integer(length) string(repository ID), string(name), TypeCode
tk-nested	integer(number of nest levels up to referenced type code) /*use to represent recursive type definitions */

Table 5-4 Type Code Kinds and Associated Parameter Lists

5.4 Examples

5.4.1 Example 1

Assume a simple CORBA IDL specification containing the module ‘‘Example’’ with an assigned object identifier of ‘‘exo’’:

```
module example
{    /* only some data types */
    interface int1
    {
        const long c1 = 6;

        enum ExEnum { x, y, z};
        struct ExStruct{
            long x;
            boolean y;
        };

        union ExUnion switch(long){
            case 1:  boolean state;
            case 2:  ExStruct info[55] ;
        };
    };
};
```

This results in the following GDMO / ASN.1 definitions:

```
-- allocated object identifier

   exo             OBJECT IDENTIFIER ::= { xopenJIDMrootOid <x> }
   exoMod1                    OBJECT IDENTIFIER ::= {exo 1 }
   exoMod1ASN1Module          OBJECT IDENTIFIER ::= { exoMod1 2 }
   exoMod1ManagedObjectClass  OBJECT IDENTIFIER ::= { exoMod1 3 }
   exoMod1Package             OBJECT IDENTIFIER ::= { exoMod1 4 }
   exoMod1Parameter           OBJECT IDENTIFIER ::= { exoMod1 5 }
   exoMod1NameBinding         OBJECT IDENTIFIER ::= { exoMod1 6 }
   exoMod1Attribute           OBJECT IDENTIFIER ::= { exoMod1 7 }
   exoMod1AttributeGroup      OBJECT IDENTIFIER ::= { exoMod1 8 }
   exoMod1Action              OBJECT IDENTIFIER ::= { exoMod1 9 }
   exoMod1IDLtypes            OBJECT IDENTIFIER ::= { exoMod1 11 }

-- GDMO template definitions

int1  MANAGED OBJECT CLASS
      DERIVED FROM  "JIDMbaseDocument":corbaObject;
      CHARACTERIZED BY int1Package  PACKAGE;
   REGISTERED AS { exoMod1ManageObjectClass  int1(1) };

-- ASN.1 Module definitions
   example-ASN1  { exoMod1ASN1Module 1}
   DEFINITIONS ::= BEGIN

      IMPORTS  ObjectInstance  FROM  ... ;
   -- need to incorporate the common base types mapping here
      c1    INTEGER  ::= 6
      ExEnum         ::= ENUMERATED { x(0), y(1), z(2)}
      ExStruct       ::= SEQUENCE{ x INTEGER, y BOOLEAN}
      ExUnion        ::= CHOICE { state BOOLEAN,
                                  info SIZE(55) ExStruct}

END
```

5.4.2 Example 2

Assume a simple CORBA IDL specification containing the module ‘‘Example2’’ with an assigned object identifier of ‘‘exo2’’:

```
module example2
{
    const long c1 = 6;

    typedef  long ExArray[10];

    typedef sequence <long, c1> limitedSeq;

    exception reject { long reason; string  info;};

    interface int1
    {
                attribute long a;
        readonly attribute sequence<octet> b;
                attribute float  c;

        long act1 ( in ExArray a, out limitedSeq b, inout ExArray c )
               raises (reject);

    };
};
```

This results in the following GDMO / ASN.1 definitions:

```
-- allocated object identifier

     exo2             OBJECT IDENTIFIER ::= { xopenJIDMrootOid <x> }
     exo2Mod1                   OBJECT IDENTIFIER ::= { exo2 1 }
     exo2Mod1ASN1Module         OBJECT IDENTIFIER ::= { exo2Mod1 2 }
     exo2Mod1ManagedObjectClass OBJECT IDENTIFIER ::= { exo2Mod1 3 }
     exo2Mod1Package            OBJECT IDENTIFIER ::= { exo2Mod1 4 }
     exo2Mod1Parameter          OBJECT IDENTIFIER ::= { exo2Mod1 5 }
     exo2Mod1NameBinding        OBJECT IDENTIFIER ::= { exo2Mod1 6 }
     exo2Mod1Attribute          OBJECT IDENTIFIER ::= { exo2Mod1 7 }
     exo2Mod1AttributeGroup     OBJECT IDENTIFIER ::= { exo2Mod1 8 }
     exo2Mod1Action             OBJECT IDENTIFIER ::= { exo2Mod1 9 }
     exo2Mod1IDLtypes           OBJECT IDENTIFIER ::= { exo2Mod1 11 }

-- GDMO template definitions

int1  MANAGED OBJECT CLASS
     DERIVED FROM  "JIDMbaseDocument":corbaObject;
     CHARACTERIZED BY int1Package  PACKAGE
       ATTRIBUTES
             a GET-REPLACE PARAMETERS corbaStandardException,
             b GET PARAMETERS corbaStandardException,
             c GET-REPLACE PARAMETERS corbaStandardException;
     ACTIONS
             act1  PARAMETERS reject, corbaStandardException; ;;
   REGISTERED AS { exo2Mod1ManagedObjectClass  1 };

   a  ATTRIBUTE
     WITH ATTRIBUTE SYNTAX  example2-ASN1.A;
     MATCHES FOR EQUALITY, ORDERING ;
   REGISTERED AS { exo2Mod1Attribute  1 };
```

```
   b  ATTRIBUTE
     WITH ATTRIBUTE SYNTAX  example2-ASN1.B;
     MATCHES FOR EQUALITY;
   REGISTERED AS { exo2Mod1Attribute 2 };

   c  ATTRIBUTE
     WITH ATTRIBUTE SYNTAX  example2-ASN1.C;
     MATCHES FOR EQUALITY, ORDERING ;
   REGISTERED AS { exo2Mod1Attribute 3 };

   act1  ACTION
     PARAMETERS  reject  -- not sure if belongs here, as opposed to
     package template
     WITH INFORMATION SYNTAX  example2-ASN1.Act1Info;
     WITH REPLY SYNTAX  example2-ASN1.Act1Reply;
   REGISTERED AS { exo2Mod1Action 1 };

   reject  PARAMETER
     CONTEXT SPECIFIC-ERROR;
     WITH SYNTAX  example2-ASN1.Reject;
   REGISTERED AS { exo2Mod1Parameter 1 };

   example2-ASN1  { exo2Mod1ASN1Module 1 }
   DEFINITIONS ::= BEGIN
      IMPORTS ObjectInstanceType
         FROM {joint-iso-ccitt ms(9) cmip(1) modules(0) protocol(3) };
    -- need to incorporate the base type mapping here
     c1   INTEGER  ::= 6
     ExArray       ::= SEQUENCE SIZE(10) OF INTEGER
     LimitedSeq    ::= SEQUENCE SIZE(c1) OF INTEGER
     Reject        ::= SEQUENCE { reason INTEGER,
                                  info   OCTET STRING}

     A        ::= INTEGER
     B        ::= SEQUENCE OF IdlOctet
     C        ::= REAL

     Act1Info ::= SEQUENCE { a SEQUENCE OF ExArray,
                                       c ExArray}

     Act1Reply::= SEQUENCE { retVal INTEGER,
                             b LimitedSeq,
                             c ExArray}
END
```

5.5 JIDM Base GDMO Document

This section gives the full specification of the JIDM base document for use with GDMO/ASN.1 specifications which have been translated from CORBA IDL specifications.

5.5.1 Assigned X/Open JIDM Object Identifier

```
jIDMbaseDocument                  OBJECT IDENTIFIER ::=
         {iso(1) member-national-body(2) bsi(826) disc(0) xopen(1050) jidmbase(10)}

jIDMbaseDocumentASN1Module        OBJECT IDENTIFIER ::= { jIDMbaseDocument 2 }
jIDMbaseDocumentManagedObjectClass OBJECT IDENTIFIER ::= { jIDMbaseDocument 3 }
jIDMbaseDocumentPackage           OBJECT IDENTIFIER ::= { jIDMbaseDocument 4 }
jIDMbaseDocumentParameter         OBJECT IDENTIFIER ::= { jIDMbaseDocument 5 }
jIDMbaseDocumentNameBinding       OBJECT IDENTIFIER ::= { jIDMbaseDocument 6 }
jIDMbaseDocumentAttribute         OBJECT IDENTIFIER ::= { jIDMbaseDocument 7 }
jIDMbaseDocumentAction            OBJECT IDENTIFIER ::= { jIDMbaseDocument 9 }
jIDMbaseDocumentIDLtypes          OBJECT IDENTIFIER ::= { jIDMbaseDocument 11 }
```

5.5.2 JIDM Base Document Managed Object Class Template

```
corbaObject  MANAGED OBJECT CLASS
   DERIVED FROM  "DMI":top;
   CHARACTERIZED BY corbaObjectPackage
      ATTRIBUTES
         corbaName  GET;
         commonNameAttribute  GET;
      NOTIFICATIONS
   ;
REGISTERED AS { jIDMbaseDocumentManagedObjectClass  1 };

corbaSystem  MANAGED OBJECT CLASS
   DERIVED FROM  corbaObject;
   CHARACTERIZED BY corbaSystemPackage
      ATTRIBUTES
         "DMI":SystemTitle   GET
         "DMI":SystemId      GET;
   ;
REGISTERED AS { jIDMbaseDocumentManagedObjectClass  2 };
```

5.5.3 JIDM Base Document Attribute Templates

```
corbaName  ATTRIBUTE
   WITH ATTRIBUTE SYNTAX  Common.corbaName;
   MATCHES FOR EQUALITY, SUBSTRING, ORDERING;
REGISTERED AS { jIDMbaseDocumentAttribute  1 };

commonNameAttribute  ATTRIBUTE
   WITH ATTRIBUTE SYNTAX
      "Recommendation X.721:1992":Attribute-ASN1Module.SimpleNameType
   MATCHES FOR EQUALITY, SUBSTRINGS, ORDERING;
REGISTERED AS {jIDMbaseDocumenbtAttribute   2};
```

5.5.4 JIDM Base Document Name Binding Templates

The following name binding is provided for use when the IDL interface has a COS name. Other name bindings may be defined for alternative naming schemes. This is why there are two naming attributes in **corbaObject**.

```
corbaSystem-corbaObject   NAME BINDING
    SUBORDINATE OBJECT CLASS  corbaObject  AND SUBCLASSES;
    NAMED BY
        SUPERIOR OBJECT CLASS  corbaSystem  AND SUBCLASSES;
    WITH ATTRIBUTE  corbaName;
REGISTERED AS { jIDMbaseDocumentNameBinding  1 };
```

5.5.5 JIDM Base Document Parameter Templates

```
corbaStandardException   PARAMETER
    CONTEXT SPECIFIC-ERROR
    WITH SYNTAX Common.CorbaStandardException
REGISTERED AS { jIDMbaseDocumentParameter  1 } ;
```

5.5.6 JIDM Base Document ASN.1 Module

```
Common MODULE { jIDMbaseDocumentASN1module  1 }
DEFINITIONS IMPLICIT TAGS ::=
BEGIN

    corbaName ::= SEQUENCE OF SEQUENCE {
                     id  GRAPHIC STRING,
                     val GRAPHIC STRING
                  }
    Octet ::= OCTET STRING (SIZE(1))
    Long  ::= INTEGER (-2147483648..2147483647)
    ULong ::= INTEGER (0..4294967295)
    Short ::= INTEGER (-32768..32767)
    UShort::=INTEGER (0..65535)

    Completion-Status  ::= ENUMERATED {
                           COMPLETED-YES(0),
                           COMPLETED-NO(1),
                           COMPLETED-MAYBE(2) }

    CorbaStandardException  ::= SEQUENCE{
                 exceptionName IA5STRING, -- standard exception name
                 minor Ulong,  -- minor code for exception
                 completed  Completion-Status
                 }

END
```

5.5.7 ASN.1 Module for Representing CORBA ANY Type Parameters

```
CORBAAnyModule { jIDMbaseDocumentASN1module 2 }
DEFINITIONS IMPLICIT TAGS ::=
BEGIN
IMPORTS
  ObjectInstance FROM CMIP-1 {joint-iso-ccitt ms(9) cmip(1) modules(0) protocol(3) };

  CORBAANY ::= SEQUENCE {
    identifiedTypeOID OBJECT IDENTIFIER,
    identifiedTypeValue ANY DEFINED BY identifiedTypeOID
  }

  freeFormCORBAANYSyntax ::= OBJECT IDENTIFIER { jIDMbaseDocumentASN1Module 2 1 }

  -- This OBJECT IDENTIFIER value serves as identifiedTypeOID for the
  -- following free form value syntax for representing CORBAANY parameters
  -- which do not have their own OID.

  FreeCORBAAny ::= SEQUENCE {
    typeCodeField TypeCode,
    anyValueField CORBAAnyValue
  }

  TCKind ::= INTEGER {
    tk-null (0), tk-void(1), tk-short(2), tk-long(3),
    tk-ushort(4), tk-ulong(5), tk-float(6), tk-double(7),
    tk-boolean(8), tk-char(9), tk-octet(10), tk-any(11),
    tk-TypeCode(12), tk-Principal(13), tk-interfaceRef(14),
    tk-struct(15), tk-union(16), tk-enum(17), tk-string(18),
    tk-sequence(19), tk-array(20), tk-alias(21),tk-except(22),
    tk-nested(-1)
  }

  TCParm ::= CHOICE {
    tcParm-string GeneralString,
    tcParm-typeCode TypeCode,
    tcParm-integer INTEGER,
    tcParm-anyVal [3] CORBAAnyValue
  }

  TypeCode :: = SEQUENCE {
    tcKind TCKind,
    numberTCParms INTEGER,
    tcParmList SEQUENCE OF TCParm -- may be empty (if tcKind <= 13)
  }

  --the contents of the parameter list for each TCKind is specified above.

  CORBAAnyValue ::= CHOICE {
    nullVal [0] NULL,
    voidVal [1] NULL,
    intVal INTEGER, -- used for short, long, ushort and ulong IDL types
    realVal REAL,   -- used for float and double IDL types
    booleanVal BOOLEAN,
    stringVal GeneralString,
      -- used for string and char IDL types, size not constrained
      -- for char due to escape sequences
    octetVal OCTET STRING SIZE INTEGER(1),
    typeCodeVal [8] TypeCode,
    interfaceRefVal [9] ObjectInstance,
    structVal [10] SEQUENCE OF CORBAAnyValue,
    unionVal [11] SEQUENCE{ discrimVal CORBAAnyValue, valChosen CORBAAnyValue},
    enumVal INTEGER, -- position in enumeration definition, start with 1
```

```
    sequenceVal [13] SEQUENCE OF CORBAAnyValue,
    arrayVal [14] SEQUENCE OF CORBAAnyValue,
    principalVal [15] OCTET STRING, -- contents depend on implementation
    anyVal [16] SEQUENCE { typeCodeVal TypeCode,
      anyValField CORBAAnyValue } -- nested any
  }
END
```

Preliminary Specification

Part 5:
SNMP to OMG IDL Translation Algorithm

The Open Group

Chapter 6

Introduction

This part of the document describes the translation of SNMP MIB Specifications into CORBA IDL. It does not address how the messages in the SNMP protocol are converted into the messages in the CORBA environment. The translation scheme can also be applied to MIBs specified in SNMP version 1 format together with the mapping of `TRAP-TYPE` macro (described in Chapter 10 on page 153).

Figure 1-2 on page 8 describes possible scenarios using CORBA-based SNMP managers and agents. The specification of the SNMP Gateway will be addressed in the Interaction Translation specification.

The SNMP MIB definition language is built on ASN.1 and this section reuses the translation given in Chapter 2 on page 15.

The organisation of this section is as follows:

- Chapter 7 on page 111 describes the basic operation of the SNMPv2 to CORBA-IDL compiler.
- Chapter 8 on page 115 describes the mapping of ASN.1 modules to IDL file and module.
- Chapter 9 on page 121 describes the mapping of SNMP macros.

Chapter 7

SNMPv2 to CORBA-IDL Translation

7.1 Outline of the Translation Algorithm

The basic scheme for translation of a SNMP MIB specification to CORBA-IDL specification is as follows:

1. Map an SNMP Information module (SMI document) into an IDL module:
   - All interfaces, types and constant generated from a SNMP information module will be within the scope of the corresponding IDL module.
   - Declare the imported types in the information module as typedef of imported IDL type.
   - Declare two IDL interfaces, called **SnmpNotifications** and **PullSnmpNotifications**, (N in Figure 7-1 on page 113), if there is at least one `NOTIFICATION-TYPE` macro in the SNMP information module. The **SnmpNotifications** interface will be used for typed-push event communication and the **PullSnmpNotifications** interface will be used for typed-pull communication.
   - Declare a pseudo IDL interface, called **DefaultValues** (D in Figure 7-1), if there is at least one `OBJECT-TYPE` macro with `DEF-VAL` clause; Pseudo IDL interfaces generate library code similar to the specification of the CORBA::TypeCode interface.
   - Declare a pseudo IDL interface, called **TextualConventions** (TC in Figure 7-1), if there is at least one `TEXTUAL-CONVENTION` macro with `DISPLAY-HINT` clause.
2. Map each ASN.1 type into IDL type using the translation scheme defined in Chapter 2 on page 15. A complex ASN.1 data type (used to describe PDUs for SNMP) may generate more than one IDL data type.
3. Map the value-specification of `SYNTAX` clause of `TEXTUAL-CONVENTION` macro, such as `DisplayString`, into an IDL type:
   - If `DISPLAY-HINT` clause is present, define two operations within the scope of the **TextualConventions** interface for converting the typed value to string and string to typed value.
4. Map the value of the invocation of the `MODULE-IDENTITY` macros into a constant IDL literal of type string.
5. Map the value of the invocation of the `OBJECT-IDENTITY` macros into a constant IDL literal of type string.
6. Map the value of the invocation of the `OBJECT-TYPE` macros into a constant IDL literal of type **ASN1_ObjectIdentifier**.
7. Each group is mapped with one IDL interface generated for the group (G in Figure 7-1), and one IDL interface (T in Figure 7-1), generated for entries of the each of the tables in the group:
   - Non-tabular variables of a group are mapped as IDL attribute within the scope of the IDL interface for the group. Column variables of the tables are mapped as attributes within the scope of the IDL interface for the entries of the table:

— Use the descriptor of the `OBJECT-TYPE` macro of the corresponding variable as the identifier of attribute.

— Acquire the IDL type of the attribute from the `SYNTAX` clause of the `OBJECT-TYPE` macro of the corresponding variable.

— Acquire the mode of the attribute from the `MAX-ACCESS` clause of the `OBJECT-TYPE` macro of the corresponding variable.

- If a `DEFVAL` cluase is present, define an operation within the scope of **DefaultValues** interface that returns the default value in typed form.

8. Map the value of the invocation of the `NOTIFICATION-TYPE` macros into a constant IDL literal of type **ASN1_ObjectIdentifier**:
   - Map the value-specification of the `OBJECTS` clause to an IDL struct which has one item for each variable in the `OBJECT` clause; the type of the items of the struct are defined as name-index-value triplet and the type of the value is derived from the variable being mapped.
   - Define an operation within the scope of **SnmpNotifications** interface of this module for typed-push event communication. The *in* parameters of the operation are source SNMP party OID, context OID, time-stamp of the event, and the IDL struct defined for the value-specification of the OBJECTS clause. Use the descriptor of the macro as the identifier for the operation.
   - Define two operations (**try_<op>** and **pull_<op>**) within the scope of the **PullSnmpNotifications** interface of this module, where <op> is the identifier of the corresponding operation in the **SnmpNotifications** interface. The parameters of the operations are same as push-event model but as *out* parameter.
9. Assign the OIDs of macros as **RepositoryId** (using **#pragma ID** declaration) to all IDL identifiers that corresponds to macro descriptor.
10. Ignore the macros related to `MODULE-COMPLIANCE` rules.

See Chapter 9 on page 121 for the mapping of each clause of the SNMPv2 macros.

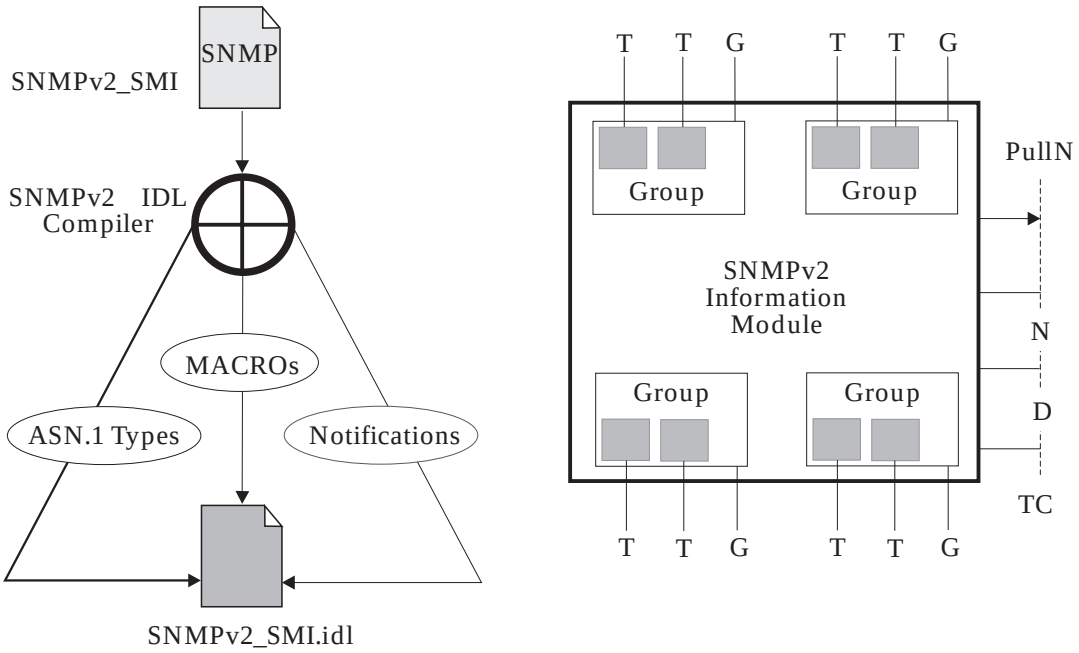

Figure 7-1 Mapping of SNMP Information Module to IDL

7.2 SNMPv2 Application-specific Type Translation

Table 7-1 describes the mapping of SNMPv2 application-specific ASN.1 types to IDL types.

For detailed description on mapping of ASN.1 type to IDL types including all primitive types, see Chapter 2 on page 15.

SNMPv2	IDL
`Integer32`	**typedef long Integer32;**
`IpAddress`	**typedef sequence<octet, 4> IpAddressType;**
`Counter32`	**typedef unsigned long Counter32Type;**
`Gauge32`	**typedef unsigned long Gauge32Type;**
`TimeTicks`	**typedef unsigned long TimeTicksType;**
`Opaque`	**typedef sequence<octet> OpaqueType;**
`NsapAddress`	**typedef ASN1_OctetString NsapAddressType;**
`Counter64`	**typedef long Counter64Type [2];**
`UInteger32`	**typedef unsigned long UInteger32Type;**

Table 7-1 Mapping of SNMP ASN.1 Types

Chapter 8

Mapping of SNMPv2 Information Modules

8.1 Lexical Translation

The lexical translation for the ASN.1 character set and identifiers follow the rules defined in the ASN.1->IDL translation document (Chapter 2 on page 15).

The suffix ‘‘Type’’ is added to all types generated from the ASN.1 types and the `SYNTAX` clause of the `OBJECT-TYPE` and `TEXTUAL-CONVENTION` macros defined in a SNMPv2 Information module.

8.2 Names and IDL Modules

8.2.1 Standard Files for Specification Translation

The specification translation assumes the existence of the following files:

ASN1Types.idl contains the base definitions for translating ASN.1 types (see Section 2.11 on page 58).

SNMPMgmt.idl contains the definition for base IDL interface for SNMP groups or Tables; it also contains the IDL types for untyped event communication and interfaces for generic typed event communication.

8.2.1.1 Contents of SNMPMgmt.idl file

The SNMPMgmt.idl file contains the definitions for the base interface for the IDL interface for SNMP groups or table-entries. The base interface is called SmiEntry and defined within the module named SNMPMgmt. The IDL interface for SmiEntry is defined in Section 12.3 on page 171.

This file also defines IDL type for untyped event communication and interfaces generic typed communications (both push and pull style). The structure for untyped event communication is defined from `NOTIFICATION PDU` format. The parameters of the operations of interfaces for typed communications are also derived from `NOTIFICATION PDU` format. The types and interfaces are defined in Section 12.3 on page 171.

8.2.2 Mapping of Module Definition

The DEFINITIONS/BEGIN statement is used to start an SNMPv2 information module in the following format:

```
<module-identifier> DEFINITIONS ::= BEGIN
    <macro-instances>
END
```

The DEFINITIONS statement is mapped to IDL as follows:

```
module <module-name> {
    <idl-mapped-macro-invocations>
};
```

where **<module-identifier>** is mapped as an ASN.1 identifier as defined in Chapter 2 on page 15. The **END** statement at the end of the information module is mapped to the closing brace and semi-colon for the IDL module.

The SNMP→IDL compiler-generated IDL file follows the following rules:

- For each SNMPv2 information module in a SNMP file, the compiler generates an IDL file. The IDL file is named based on the <module-identifier> defined in the corresponding information module.
- The information in an IDL file must be enclosed using **#ifndef/#endif** in the following way:

```
#ifndef _<module-identifier>_IDL_
#define _<module-identifier>_IDL_
module <module-identifier> {

<idl-mapped-macro-invocations>

};
#endif /* !_<module-identifier>_IDL_ */
```

8.2.3 Naming of the IDL File Output

The <module-identifier> is used to name the IDL file generated by the SNMP→IDL compiler. The file name is formed by concatenating **.idl** to the <module-identifier>.

8.3 Mapping of IMPORTing Symbols

In SNMPv2 information modules, the `IMPORTS` statement is used to reference an external object by identifying both the descriptor and the module defining the descriptor, in the following format:

```
IMPORTS <descriptor1> [, <descriptor2 ... [, descriptorn] ] FROM <module-name>
```

Descriptors which are not macro references or macro descriptors in the `IMPORTS` statement are declared within the scope of the IDL module by way of typedefs as defined in Section 2.5.4 on page 25, yielding the following IDL:

```
typedef <module-name>::<descriptor1>Type <descriptor1>Type;
typedef <module-name>::<descriptor2>Type <descriptor2>Type;
 ........
typedef <module-name>::<descriptor1>Type <descriptor1>Type;
```

Note the addition of the **Type** suffix to all types that are defined in another module. Since base ASN.1 types are assumed to be globally defined and they are not explicitly imported, the rule is always consistent.

If there are no `IMPORTS ... FROM` statements in the SNMP module, then the following include directives will be added at the beginning of the file:

```
#include <ASN1Types.idl>
#include <SNMPMgmt.idl>
```

in order to include the global ASN.1 types and base IDL interface.

Each `FROM` <module-name> will map to an include statement at the beginning of the file as follows:

```
#include <<module-name>.idl>
```

If there is at least one imported module then it is not necessary to explicitly import the standard IDL files.

8.3.1 Example

By way of example, the following definition in RFC1450:

```
SNMPv2-MIB DEFINITIONS ::= BEGIN
IMPORTS
    MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE,
    ObjectName, Integer32, Counter32, snmpModules FROM SNMPv2-SM
    TruthValue, DisplayString, TestAndIncr, TimeStamp FROM SNMPv2-T
    MODULE-COMPLIANCE, OBJECT-GROUP FROM SNMPv2-CON
    system, ifIndex, egpNeighAddr FROM RFC1213-MIB
    partyEntry FROM SNMPv2-PARTY-MIB;
    ....
END
```

maps to following definition in a file named SNMPv2_MIB.idl:

```
#ifndef SNMPv2_MIB_IDL
#define SNMPv2_MIB_IDL
#include <SNMPv2_SMI.idl>
#include <SNMPv2_TC.idl>
#include <SNMPv2_CONF.idl>
#include <RFC1213_MIB.idl>
#include <SNMPv2_PARTY_MIB.idl>

module SNMPv2_MIB {
// MODULE-IDENTITY, OBJECT-TYPE, NOTIFICATION-TYPE are ignored
typedef SNMPv2_SMI::ObjectNameType ObjectNameType;
typedef SNMPv2_SMI::Integer32Type Integer32Type;
typedef SNMPv2_SMI::Counter32Type Counter32Type;
// snmpModules is macro descriptor so it is ignored
typedef SNMPv2_TC::TruthValueType TruthValueType;
typedef SNMPv2_TC::DisplayStringType DisplayStringType;
typedef SNMPv2_TC::TestAndIncrType TestAndIncrType;
typedef SNMPv2_CONF::TimeStampType TimeStampType;
// system, ifIndex, egpNeighAddr are ignored because they are macro-descriptors
// partyEntry is ignored because its is macro-descriptor
 ........
}; /* End of SNMPv2_MIB module */
#endif /* !SNMPv2_MIB_IDL */
```

Chapter 9

SNMPv2 Information Module Macros

9.1 Macro Invocation

Within an information module, each macro invocation appears as:

```
<descriptor> <macro> <clauses> ::= <value>
```

where:

<descriptor>	corresponds to an ASN.1 identifier
<macro>	names the macro being invoked
<clauses> and <value>	depend on the definition of the macro.

Table 9-1 describes the production of an ASN.1 macro.

MacroDefinition ::= macroreference MACRO "::=" MacroSubstance
MacroSubstance ::= BEGIN MacroBody END | macroreference | Externalmacroreference
MacroBody ::= TypeProduction | ValueProduction | SupportingProductions
TypeProduction ::= TYPE NOTATION "::=" MacroAlternativeList
MacroAlternativeList ::= SymbolList
SymbolList ::= SymbolElement | SymbolList SymbolElement
SymbolElement ::= SymbolDefn | EmbeddedDefinitions

Table 9-1 Production of ASN.1 Macro Definition Notation

9.2 SNMPv2-SMI MODULE-IDENTITY Macro

The `MODULE-IDENTITY` macro is used to provide contact and revision history for each information module. It appears exactly once in every information module.

The <descriptor> of the `MODULE-IDENTITY` macro is mapped to an IDL constant, where the identifier for the literal is the <descriptor> and the IDL scoped name of the identifier for the <descriptor> is used as the value of the constant.

```
const string <descriptor> = "::<Module-Scoped-Name>::<descriptor>";
```

The value of an invocation of the `OBJECT-IDENTITY` macros is mapped into an IDL pragma.

The value-specification of the other clauses of `MODULE-IDENTITY` are mapped as block comments below the IDL literal for the `OBJECT IDENTIFIER` of the `MODULE-IDENTITY` macro. The comments for the clauses are generated in the order they are encountered.

9.2.1 Mapping of the LAST-UPDATED Clause

The `LAST-UPDATED` clause, which must be present, contains the date and time that this SNMP information module was last edited. The value-specification for this clause is mapped as part of the block-comment below the string literal for macro-descriptor:

```
/*
LAST-UPDATED : <value-specification>
*/
```

9.2.2 Mapping of the ORGANIZATION Clause

The `ORGANIZATION` clause, which must be present, contains a textual description of the organisation under whose auspices this information module was developed.

The value-specification for this clause is mapped as block-comment below the string literal for macro-descriptor:

```
/*
ORGANIZATION : <value-specification>
*/
```

9.2.3 Mapping of the CONTACT-INFO Clause

The `CONTACT-INFO` clause, which must be present, contains the name, postal address, telephone number, and electronic mail address of the person to whom technical queries concerning this information module should be sent.

The value-specification for this clause is mapped as block-comment below the string literal for macro-descriptor:

```
/*
CONTACT-INFO : <value-specification>
*/
```

9.2.4 Mapping of the DESCRIPTION Clause

The `DESCRIPTION` clause, which must be present, contains a high-level textual description of the contents of this information module:

The value-specification for this clause is mapped as block-comment below the string literal for macro-descriptor:

```
/*
DESCRIPTION : <value-specification>
*/
```

9.2.5 Mapping of the REVISION Clause

The `REVISION` clause, which need not be present, is repeatedly used to describe the revisions made to this information module, in reverse chronological order. Each instance of this clause contains the date and time of the revision.

The value-specification for each of this clause and its description is mapped as block comment below the string literal for macro-descriptor:

```
/*
REVISION : <revision-value-specification>
REVISION-DESCRIPTION : <description-value-specification>
  ....
REVISION : <revision-value-specification>
REVISION-DESCRIPTION : <description-value-specification>
*/
```

9.2.6 Mapping of the MODULE-IDENTITY Value

The value of an invocation of the `MODULE-IDENTITY` is an `OBJECT IDENTIFIER` and this value is used as authoritative registration identifier for referencing. The value of the invocation is mapped as a **#pragma** declaration for the IDL literal for the descriptor of the macro as identifier.

9.2.7 Example

Example 9-1 illustrates the mapping of an instance of SNMP `MODULE-IDENTITY` macro, called *fizbin*.

Note that the block-comments generated from individual clauses of a macro have been merged into a single block comment. This approach will be followed in the rest of the document whenever it is possible.

Example 9-1 Conversion of SNMP MODULE-IDENTITY fizbin

MODULE IDENTITY	IDL Interface
`fizbin MODULE-IDENTITY` `LAST-UPDATED "9210070433Z"` `ORGANIZATION "JIDM Task Force"` `CONTACT-INFO` `"JIDM Task Force` `Postal: X/Open Company Ltd.` `Apex Plaza, Forbury Road` `Reading, Berks, RG1 1AX, ENGLAND` `Tel: +44 118 950 8311` `E-mail: XoJIDM@xopen.org"` `DESCRIPTION` `"The MIB module for entities` `implementing the xxxx protocol."` `REVISION "9210070433Z"` `DESCRIPTION` `"Initial version of this MIB module."` `-- contact IANA for actual number` `::= { experimental 555}` `-- Assuming 555 is not used`	**module FIZ_MIB {** **const string moduleIdentity = "fizbin";** **const ASN1_ObjectIdentifier fizbin** **= "::FIZ-MIB::fizbin";** **#pragma ID fizbin "1.3.6.1.3.555";** **/*** **LAST-UPDATED: : "9210070433Z"** **ORGANIZATION: : "JIDM Task Force"** **CONTACT-INFO :** **"JIDM Task Force** **Postal: X/Open Company Ltd.** **Apex Plaza, Forbury Road** **Reading, Berks, RG1 1AX, ENGLAND** **Tel: +44 118 950 8311** **E-mail: XoJIDM@xopen.org"** **DESCRIPTION :** **The MIB module for entities** **implementing the xxxx protocol.** **REVISIONS : 9210070433Z** **REVISION-DESCRIPTION:** **Initial version of this MIB module.** ***/** **....** **}; // End of FIZ_MIB module**

9.3 SNMPv2-SMI OBJECT-IDENTITY Macro

The `OBJECT-IDENTITY` macro is used to define information about an `OBJECT IDENTIFIER` assignment.

The macro is mapped as IDL constant literal of type string. The identifier of the constant is the descriptor of the macro and the value of the constant is IDL scoped name of the constant. The value of the invocation is mapped as a **#pragma** declaration for the IDL literal for the descriptor of the macro. All other clauses are mapped within a block comment below the constant for the value of the macro.

```
const string <descriptor> = "<Module-ScopedName>::<descriptor>";
#pragma ID <descriptor> = "<OID>";
```

If the value-specification of the `STATUS` clause of an `OBJECT-IDENTITY` macro is either deprecated or obsolete, then the macro is not mapped to IDL.

9.3.1 Mapping of the DESCRIPTION Clause

The `DESCRIPTION` clause contains a high-level textual description of the object assignment and it must be present. The value-specification for this clause is mapped as block comment below the IDL constant for the value of the macro in the following form:

```
/*
DESCRIPTION : <value-specification>
*/
```

9.3.2 Mapping of the REFERENCE Clause

The `REFERENCE` clause contains a textual cross-reference to an object assignment defined in some other module and it need not be present. If the clause is present then the value-specification for this clause is mapped as block comment below the IDL constant for the value of the macro in the following form:

```
/*
REFERENCE: <value-specification>
*/
```

9.3.3 Mapping of the OBJECT-IDENTITY Value

The value of an invocation of the `OBJECT-IDENTITY` is an `OBJECT IDENTIFIER`, and this value is used as the authoritative registration identifier for referencing.

The value of the invocation is mapped as a **#pragma** declaration for the IDL literal for the descriptor of the macro as identifier.

9.3.3.1 Example

Example 9-2 shows the mapping of an `OBJECT-IDENTITY` macro, called *fizbin69* to the corresponding IDL literal.

Example 9-2 Mapping of OBJECTIDENTITY fizbin

OBJECT IDENTITY	IDL Interface
`FIZ-MIB DEFINITIONS ::= BEGIN` `  ......` `fizbinChipSets OBJECT IDENTIFIER` `  ::= { fizbin 1 }` `fizbin69 OBJECT-IDENTITY` `  STATUS current` `  DESCRIPTION` `  "The authoritative identity` `  of the Fizbin 69 chipset."` `  ::= { fizbinChipSets 1 }` `  ......` `END`	**module FIZ_MIB {** ** ......** **const string fizbinChipSets** ** = "::FIZ_MIB::fizbinChipSets";** **#pragma ID fizbinChipSets "1.3.6.1.3.555.1"** **const string fizbin69 = "::FIZ_MIB::fizbin";** **#pragma ID fizbin69 "1.3.6.1.3.555.1.1"** **/* DESCRIPTION :** ** "The authoritative identity of the** ** Fizbin 69 chipset"** ***/** ** ......** **// End of FIZ_MIB module**

9.4 SNMPv2 OBJECT-TYPE Macro

Table 9-2 describes the structure of the OBJECT-TYPE Macro. The OBJECT-TYPE macro is used to define the Table, TableEntry and the variables in SNMP information module.

```
OBJECT-TYPE MACRO ::=

BEGIN
  TYPE NOTATION ::=
    "SYNTAX" type(Syntax)
    UnitsPart
    "MAX-ACCESS" Access
    "STATUS" Status
    "DESCRIPTION" Text
    ReferPart
    IndexPart
    DefValPart
  VALUE NOTATION ::= value (VALUE ObjectName)
  UnitsPart ::= "UNITS" Text | empty
  Access ::= "not-accessible" | "read-only" | "read-write" | "read-create"
  Status ::= "current" | "deprecated" | "obsolete"
  ReferPart ::= "REFERENCE" Text | empty
  IndexPart ::= "INDEX" "{" IndexTypes "}" | "AUGMENTS" "{" Entry "}" | empty
  IndexTypes ::= IndexType | IndexTypes "," IndexType
  IndexType ::= "IMPLIED" Index | Index
  Index ::= value (Indexobject ObjectName)
  Entry ::= value (Entryobject ObjectName)
  DefValPart ::= "DEFVAL" "{" value (Defval Syntax) "}" | empty
  -- uses the NVT ASCII character set
  Text ::= """" string """"
END
```

Table 9-2 Structure of OBJECT-TYPE Macro Clauses

9.4.1 Base IDL Interface for SNMP Group or Table Entry

In RFC1442, a conceptual table is defined as follows:

> "A conceptual table has SYNTAX of the form:
>
> ```
> SEQUENCE OF <EntryType>
> ```
>
> where <EntryType> refers to the SEQUENCE type of its subordinate conceptual row".

A conceptual row has SYNTAX of the form:

```
<EntryType>
```

where <EntryType> is of type SEQUENCE. <EntryType> is defined as follows:

```
<EntryType> ::= SEQUENCE { <type1>, ... , <typeN> }
```

where there is one <type> for each subordinate object, and each <type> is of the form:

```
<descriptor> <syntax>
```

where <descriptor> is the descriptor naming a subordinate object, and <syntax> has the value of that subordinate object's SYNTAX clause, optionally omitting the sub-typing information.

Although no special syntax exists for defining an abstract group, a group object can be easily identified based on the description of the objects subordinate to the group object in the Object-Identifier hierarchy. An abstract group is considered as a special case of the conceptual table where there exists only one row for an abstract group.

The rows of a conceptual table are realised by mapping each row of the table to an instance of an object class, which is described by an IDL interface. The instances of the variables of an SNMP group object are realised by the values of the attribute of an instance object, described by an IDL interface. In other words, an individual row of a table is treated as an instance of object, and attributes of this object represent the instances of the variables in the row of an SNMP table.

Note that this mapping scheme is consistent with the modelling principle defined by IIMC for translating an SNMP document to GDMO document.

The IDL interface **SmiEntry** defines the base class for both the rows of a table and group objects. Figure 9-1 on page 129 describes the inheritance hierarchy for IDL interfaces for the rows of tables or group objects. No attribute or operation is defined for the **SmiEntry**. It will be addressed in the Interaction Translation specification. It is provided as a place holder for attributes and operations needed to implement the SNMP access to the instances of the IDL interface for **TableEntry** or **Group** objects in a generic way.

The IDL interface for **SmiEntry** is defined in a separate IDL file, called SNMPMgmt.idl, as shown in Section 12.3 on page 171.

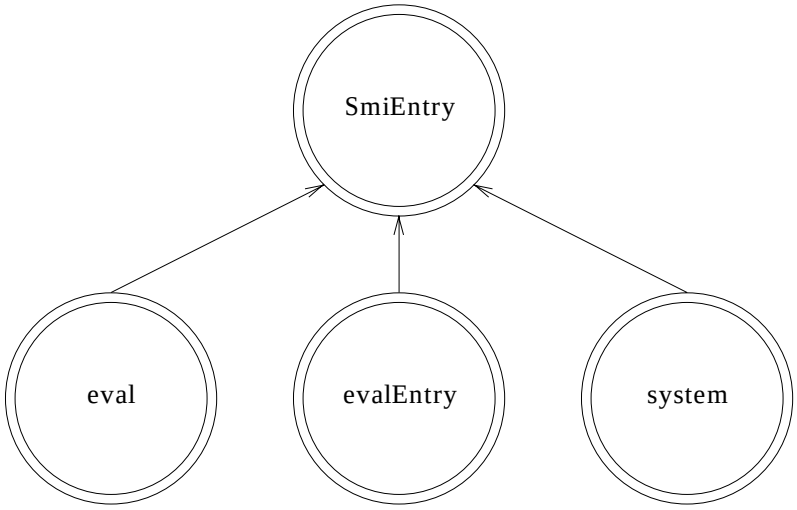

Figure 9-1 Inheritance Hierarchy of IDL Interfaces for TableEntry / Group

9.4.2 Mapping of OBJECT-TYPE Macro for Table

The `OBJECT-TYPE` macro for a table is mapped as IDL constant literal of type string. The identifier of the constant is the descriptor of the macro and the value of the constant is the IDL scoped name of the constant. The value of the invocation is mapped as **#pragma** declaration for the IDL literal for the descriptor of the macro. All other clauses are mapped within a block comment below the constant for the value of the macro.

```
const string <descriptor> = <Module-ScopedName>::<descriptor>;
#pragma ID <descriptor> = <OID>;
```

The value specification of the `DESCRIPTION` clause is copied below the IDL literal as block comment in the following form:

```
/*
DESCRIPTION: <description-value-specification>
*/
```

9.4.2.1 Example

Example 9-3 Shows the mapping of `OBJECT-TYPE` macro for a table (called evalTable) to a corresponding IDL literal. The instances of `OBJECT-TYPE` macro whose `STATUS` clause is either **deprecated** or **obsolete** are ignored.

Example 9-3 Mapping of OBJECT-TYPE Macro for Table

OBJECT IDENTITY	IDL
`FIZ-MIB DEFINITIONS ::= BEGIN` `  ......` `evalTable OBJECT-TYPE` `  SYNTAX SEQUENCE OF EvalEntry` `  MAX-ACCESS not-accessible` `  STATUS current` `  DESCRIPTION` `  "The (conceptual)` `  evaluation table."` `  ::= {eval 2}` `  .......` `END`	**const string evalTable = "::FIZ_MIB::evaltable";** **#pragma ID evalTable "1.3.6.1.3.555.2.2";** **/* DESCRIPTION :** **"The (conceptual) evaluation table."** ***/**

9.4.3 Mapping of OBJECT-TYPE Macro for Table Entry

The `OBJECT-TYPE` macros for TableEntry are mapped as IDL interfaces. The macro descriptor is used as the identifier of the IDL interface.

The attributes of the IDL interface are identified based on the descriptors defined in the ASN.1 `SEQUENCE` in the `SYNTAX` clause of the `OBJECT-TYPE` macro of the table entry.

See Section 9.4.5 on page 136 for the generation of attributes for the IDL interface from the `OBJECT-TYPE` macro for the column-variables of the table.

The value specification of the other clauses of this macro is mapped as IDL comment.

9.4.3.1 Mapping of the Macro Descriptor

The descriptor in `OBJECT-TYPE` macro for TableEntry is used as the identifier for the corresponding IDL interface. IDL interface for `OBJECT-TYPE` macro for TableEntry always inherits the **SNMPMgmt::SmiEntry** IDL interface.

9.4.3.2 Mapping of the IndexPart clause to IDL

An IndexPart clause contains either an `INDEX` clause or an `AUGMENT` clause. It may not be present at all. The `INDEX` or `AUGMENT` clause must be present if that object corresponds to conceptual row. The value-specification of this clause is mapped as comment below the declaration of the IDL interface in the following form:

```
/*
INDEX : { <value-specification> }
or
AUGMENTS : { <value-specifications> }
*/
```

9.4.3.3 Mapping of the DESCRIPTION Clause

The `DESCRIPTION` clause, which must be present, contains a textual definition of the object which provides all semantic definition necessary for implementation. The value-specification of this clause is mapped as comment below the declaration of the IDL interface in the following form:

```
/*
DESCRIPTION: <value-specification>
*/
```

9.4.3.4 Mapping of the REFERENCE Clause

The `REFERENCE` clause, which need not be present, contains a textual cross-reference to an object defined in some other information module. If this clause is present then, the value-specification for this clause is mapped as comment below the IDL interface declaration in the following form:

```
/*
REFERENCE: <value-specification>
*/
```

9.4.3.5 Mapping of the OBJECT-TYPE Value

The value of an invocation of the `OBJECT-TYPE` is an `OBJECT IDENTIFIER` and this value is used as authoritative registration identifier for referencing.

The value of the invocation of the `OBJECT-TYPE` macro for TableEntry is mapped as a **#pragma** declaration for the IDL literal for the descriptor of the macro.

9.4.3.6 Example

The following example illustrates the mapping of the Table Entry `evalEntry` to its corresponding IDL interface **evalEntry**. The interface `evalEntry` is a subtype of **SNMPMgmt::SmiEntry**. The descriptors in the ASN.1 type `EvalEntry` are used to identify the IDL attributes for the interface **evalEntry**.

Example 9-4 Mapping of OBJECT-TYPE Macro for Table Entry

OBJECT IDENTITY	IDL
FIZ-MIB DEFINITIONS ::= BEGIN ...... evalEntry OBJECT-TYPE SYNTAX EvalEntry MAX-ACCESS not-accessible STATUS current DESCRIPTION "An entry. The (conceptual row) in the evaluation table." INDEX { evalIndex } ::= {evalTable 1 } EvalEntry ::= SEQUENCE { evalIndex Integer32, evalString DisplayString, evalValue Integer32, evalStatus RowStatus } ...... END	**module FIZ_MIB {** ** ......** **interface evalEntry : SNMPMgmt::SmiEntry {** ** #pragma ID evalEntry "1.3.6.1.3.555.2.2.1"** ** /*** ** INDEX : { evalIndex }** ** DESCRIPTION:** ** An entry. The (conceptual row) in the** ** evaluation table.** ** Registered As:2.3.6.1.3.555.2.2.1** ** */** ** readonly attribute Integer32 evalIndex;** ** /***** ** DESCRIPTION :** ** The auxiliary variable used for** ** identifying instances of the** ** columnar objects in the evaluation** ** table.** ** */** ** readonly attribute DisplayString evalString;** ** #pragma ID evalString "1.33.6.1.33.555.2.2.1.2"** ** /***** ** DESCRIPTION :** ** The Index Number of first unassigned** ** entry in the evaluation table.** ** */** ** readonly attribute Integer32 evalValue;** ** #pragma ID evalValue "1.33.6.1.3.555.2.2.1.3"** ** /***** ** DESCRIPTION :** ** The value when eval string was last executed.** ** DEFVAL : 0** ** */** ** readonly attribute RowStatusType evalStatus;** ** #pragma ID evalStatus "1.3.6.1.3.555.2.2.1.4"** ** /***** ** DESCRIPTION :** ** The status column used for creating,** ** modifying, and deleting instances of** ** the columnar objects in the evaluation table.** ** */** **}; // End of evalEntry interface** **/* pseudo */** **interface DefaultValues {** **// DEFVAL : 0** ** Integer32 evalValue();** **// DEFVAL : active** ** RowStatusType evalStatus();** **};** **}; // End of FIZ_MIB module**

9.4.4 Mapping of SNMP Group

Although no special syntax exists for defining an abstract group, a group object can be easily identified based on the description of the objects subordinate to the group object in the Object-Identifier hierarchy. An abstract group is considered as a special case of the conceptual table where there exists only one row for an abstract group.

If an `OBJECT IDENTIFIER` assignment in the following form:

```
<descriptor> OBJECT IDENTIFIER ::= ObjectIdentifierValue
```

is deduced to be a group from its subordinate objects in the Object-Identifier hierarchy, then the assignment is implicitly mapped to `OBJECT-TYPE` macro first before mapping to an IDL interface. The clauses for implicit `OBJECT-TYPE` macro is defined as follows:

```
<descriptor> OBJECT-TYPE
    SYNTAX <cap-descriptor>Entry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION ""
    -- Reference, INDEX and DEFVAL are not present
::= ObjectIdentifierValue
```

The ASN.1 syntax for <cap-descriptor>Entry is deduced based on the non-tabular objects below the group object in object-identifier hierarchy:

```
<cap-descriptor>Entry  ::= SEQUENCE {
    <descriptor> <syntax> -- one line for each non-tabular
                          -- object in the group
}
```

where <descriptor> is the descriptor for a non-tabular subordinate object, and <syntax> has the value-specification of that subordinate object's `SYNTAX` clause (omitting the sub-typing information).

One IDL interface is generated for each SNMP group in an SNMP information module. The descriptor of the `OBJECT IDENTIFIER` assignment for a group is used as the identifier for the IDL interface. The value of the `OBJECT IDENTIFIER` assignment is mapped as a **#pragma** declaration for the identifier of the IDL interface. The non-tabular objects of a SNMP group are mapped as attributes of the corresponding IDL interface. Only those non-tabular objects whose value-specification `STATUS` clause is *current* are mapped as attributes.

See Section 9.4.5 on page 136 for the generation of attributes from the `OBJECT-TYPE` macro for the non-tabular subordinate objects of the group object.

9.4.4.1 Example 1

Example 9-5 describes the mapping of the group `eval` to its corresponding IDL interface **eval**. The interface `eval` inherits the base interface, called **SNMPMgmt::SmiEntry**. The variables of the `eval` group are represented as the IDL attribute in the interface **eval**. Note that the evalTable is not mapped as IDL attribute of the `eval` interface.

Example 9-5 Conversion of Group eval

SNMP Groups	IDL Interface
FIZ-MIB DEFINITIONS ::= BEGIN ..... eval OBJECT IDENTIFIER = {fizbin 2 } -- eval group has object called evalSlot evalSlot OBJECT-TYPE SYNTAX INTEGER MAX-ACCESS read-only STATUS current DESCRIPTION "" ::= {eval 1} evalTable OBJECT-TYPE SYNTAX SEQUENCE OF EvalEntry MAX-ACCESS not-accessible STATUS current DESCRIPTION "The (conceptual) evaluation table." ::= {eval 2} ..... END	**module FIZ_MIB {** **.....** **interface eval : SNMPMgmt::SmiEntry {** **#pragma ID eval "1.3.6.1.3.555.2"** **/*** **DESCRIPTION:** **INDEX:** ***/** **readonly attribute ASN1_Integer evalSlot;** **#pragma ID evalSlot "1.3.6.1.3.555.2.1"** **/*** **DESCRIPTION :** **The Index Number of first unassigned** **entry in the evaluation table** **.....** ***/** **// Ignore evalTable even if it is a** **// subordinate object to eval group** **// because it is a tabular object.** **};** **const evaTable evalTable = "::FIZ_MIB::evaltable";** **#pragma ID evalTable "1.3.6.1.3.555.2.2"** **/*** **DESCRIPTION :** **"The (conceptual) evaluation table."** ***/**

9.4.4.2 Example 2

Example 9-6 describes the mapping of the SNMP group `system` to IDL.

Example 9-6 Conversion of Group system

SNMP Groups	IDL Interface
`RFC1213-MIB DEFINITIONS ::= BEGIN` `  ......` `system OBJECT IDENTIFIER ::= {mib-2 1}` `  ......` `END` `_`	**module RFC1213_MIB {** ** ......** **interface system : SNMPMgmt::SmiEntry {** ** #pragma ID system "1.3.6.1.2.1.1"** ** /*** ** DESCRIPTION:** ** INDEX:** ** */** ** readonly attribute DisplayStringType sysDescr;** ** #pragma ID sysDescr "1.3.6.1.2.1.1.1"** ** /*** ** DESCRIPTION :** ** */** ** readonly attribute ASN1_ObjectIdentifier sysObjectID;** ** #pragma ID sysObjectID "1.3.6.1.2.1.1.2"** ** /*** ** DESCRIPTION :** ** */** ** readonly attribute TimeTicksType sysUpTime;** ** #pragma ID sysUpTime "1.3.6.1.2.1.1.3"** ** /*** ** DESCRIPTION :** ** */** ** attribute DisplayStringType sysName;** ** #pragma ID sysName "1.3.6.1.2.1.1.4"** ** /*** ** DESCRIPTION :** ** */** ** attribute DisplayStringType sysLocation;** ** #pragma ID sysLocation "1.3.6.1.2.1.1.5"** ** /*** ** DESCRIPTION :** ** */** ** attribute DisplayStringType sysServices;** ** #pragma ID sysServices "1.3.6.1.2.1.1.6"** ** /*** ** DESCRIPTION :** ** */** **};** ** ......** **}; // End of RFC1213_MIB**

9.4.5 Mapping of OBJECT-TYPE Macro for Variables

For each `OBJECT-TYPE` macro, that defines a leaf-object in the ASN1_ObjectIdentifier hierarchy, an IDL attribute is defined within the scope of the IDL interface for the corresponding parent object. The instances of `OBJECT-TYPE` macro whose STATUS clause is either *deprecated* or obsolete are not mapped to IDL.

Note that in the GDMO-to-IDL translation, all attributes are mapped as IDL operations, whereas in the SNMPv2-to-IDL translation variables are mapped as attributes. The main reason for mapping GDMO attributes to IDL operations is to support specific-errors for attributes. In mapping SNMPv2 to IDL such cases do not arise, and the IDL **NO_IMPLEMENT** exception is sufficient to take care of most of the situations.

SNMP errors are mapped to CORBA exceptions as follows:

SNMP Error	CORBA Exception
`noError`	**NO_EXCEPTION**
`inconsistentValue`	**BAD_PARAM**
`resourceUnavailable`	**NO_RESOURCE**
`authorizationError`	**NO_PERMISSION**

Table 9-3 Mapping SNMP Errors to CORBA Exceptions

The other SNMP errors are not related to MIB variable access and will not occur as they will be intercepted by the gateway.

9.4.5.1 Mapping of the Macro Descriptor

The macro-descriptor in `OBJECT-TYPE` macro is mapped as identifier for the corresponding IDL attribute.

9.4.5.2 Mapping of the SYNTAX Clause

The `SYNTAX` clause, which must be present, defines the abstract data-structure corresponding to that object. The data structure must be one of the alternatives defined in the ObjectSyntax `CHOICE`. Full ASN.1 subtyping is allowed.

The value-specification of the `SYNTAX` clause is mapped as *type* of the corresponding attribute.

If the value of the `SYNTAX` clause is defined to be an ASN.1 subtype, then the subtype definition is first used to define a new ASN.1 type and then the new ASN.1 type is used as type of attribute. The new ASN.1 type is mapped as IDL type within the scope of the interface, and the identifier for the new ASN.1 type is formed by capitalizing the first character of the descriptor for the `OBJECT-TYPE` macro. During the conversion from ASN.1 to IDL, a **Type** suffix will be added to the IDL type identifier.

Table 9-4 on page 137 describes the mapping of ASN.1 subtype in the SYNTAX clause.

SNMP OBJECT-TYPE Macro	IDL Attributes
`evalString OBJECT-TYPE` `  SYNTAX OCTET STRING (SIZE (0..255))` `  MAX-ACCESS read-create` `  STATUS current` `  DESCRIPTION` `  "The string to evaluate.` `  evaluation table."` `  ::= {evalEntry 2 }`	**typedef sequence<octet, 256> EvalStringType;** **// The above typedef is derived from** **// the following ASN.1 type** **// EvalString ::= OCTET STRING (SIZE (0..255))** **attribute EvalStringType evalString;**

Table 9-4 Mapping ASN.1 Subtype in OBJECT-TYPE Macro SYNTAX Clause

9.4.5.3 Mapping of the MAX-ACCESS Clause

The `MAX-ACCESS` clause, which must be present, defines whether it makes ‘‘protocol-sense’’ to **read**, **write** or **create** an instance of the object.

The mapping of the value-specification of the `MAX-ACCESS` clause of a variable is described in Table 9-5.

SNMP MAX-ACCESS	IDL Attribute Mode
`read-only`	**readonly**
`read-write`	**<empty>**
`read-create`	**<empty>**
`not-accessible`	**<comment>**

Table 9-5 Mapping MAX-ACCESS Clause of OBJECT-TYPE Macro

9.4.5.4 Mapping of the UNITS Clause

The `UNITS` clause, which need not be present, contains a textual definition of the units associated with that object. If this clause is present then the value-specification for this clause is mapped as IDL comment below the attribute in the following form:

```
/*
UNITS : <value-specification>
*/
```

9.4.5.5 Mapping of the STATUS Clause

The `STATUS` clause, which must be present, indicates whether this definition is current or historic. Possible values of `STATUS` clause are *current*, *deprecated* or *obsolete*. Only those `OBJECT-TYPE` macros whose value-specification for `STATUS` clause is current are mapped.

9.4.5.6 Mapping of the DESCRIPTION Clause

The `DESCRIPTION` clause, which must be present, contains a textual definition of the object which provides all semantic definition necessary for implementation. The value-specification for this clause is mapped as comment below the corresponding IDL attribute in the following form:

```
/*
DESCRIPTION: <value-specification>
*/
```

9.4.5.7 Mapping of the REFERENCE Clause

The REFERENCE clause, which need not be present, contains a textual cross-reference to an object defined in some other information module. If this clause is present then, the value-specification for this clause is mapped as comment below the corresponding IDL attribute in the following form:

```
/*
REFERENCE: <value-specification>
*/
```

9.4.5.8 Mapping of the IndexPart Clause

This clause is not applicable for column-variables in a table or a non-tabular object for a group.

9.4.5.9 Mapping of the DEFVAL Clause

The DEFVAL clause, which need not be present, defines an acceptable default value which may be used at the discretion of an SNMPv2 entity acting as agent role when an object instance is created. The value of the DEFVAL clause must correspond to the SYNTAX clause for the object.

If this clause is present then an operation is defined within the scope of the **DefaultValues** pseudo IDL interface in the following form:

```
/* pseudo */
interface DefaultValues {
   // DEFVAL: <value-specification>
   <Type> <macro-descriptor>();
};
```

The value-specification of this clause is mapped as a comment before the operation declaration. <Type> is derived from the mapping of SYNTAX clause of the macro. IDL pseudo interfaces are mapped as a library API (like the **CORBA::TypeCode** interface) in contrast to the client and server side skeletons.

9.4.5.10 Mapping of the OBJECT-TYPE Value

The value of an invocation of the OBJECT-TYPE is an OBJECT IDENTIFIER and this value is used as the authoritative registration identifier for referencing.

The value of the invocation of the OBJECT-TYPE macro for a variable is mapped as a **#pragma** declaration for the identifier of the corresponding attribute.

Example

Example 9-7 illustrates the mapping of two Object-Types, called evalIndex and evalString, that represent the columnar variable in the table evalTable.

Example 9-7 Conversion of OBJECT-TYPE macro

SNMP OBJECT-TYPE Macro	IDL Attributes
`5evalIndex OBJECT-TYPE` `  SYNTAX Integer32` `  MAX-ACCESS not-accessible` `  STATUS current` `  DESCRIPTION` `  "The auxiliary variable used` `  for identifying instances of` `  the columnar objects in the` `  evaluation table."` `  ::= {evalEntry 1}` `evalString OBJECT-TYPE` `  SYNTAX DisplayString` `  MAX-ACCESS read-create` `  STATUS current` `  DEFVAL "InitialString"` `  DESCRIPTION` `  "The string to evaluate.` `  evaluation table."` `  ::= {evalEntry 2 }`	**readonly attribute Integer32Type evalIndex;** **#pragma ID evalIndex "1.3.6.1.3.555.2.2.1.1"** **/*** **DESCRIPTION :** **The auxiliary variable used** **for identifying instances of** **the columnar objects in the** **evaluation table.** ***/** **readonly attribute DisplayStringType evalString;** **#pragma ID evalString "1.3.6.1.3.555.2.2.1.2"** **/*** **DESCRIPTION :** **The Index Number of first unassigned entry** **in the evaluation table.** ***/** **/* pseudo */** **interface DefaultValues {** **// DEFVAL: "Initial String"** **DisplayStringType evalString();** **};**

9.5 SNMPv2-SMI NOTIFICATION-TYPE Macro

The NOTIFICATION-TYPE macro is used to define the information contained within an unsolicited transmission of management information. Table 9-6 shows the structure of the NOTIFICATION-TYPE macro.

```
NOTIFICATION-TYPE MACRO ::=
BEGIN
    TYPE NOTATION ::=
        ObjectsPart
        "STATUS" Status
        "DESCRIPTION" Text
        ReferPart
    VALUE NOTATION ::= value (VALUE OBJECT IDENTIFIER)
    ObjectsPart ::= "OBJECTS" "{" Objects "}" | empty
    Objects ::= Object | Objects "," Object
    Object ::= value (Name ObjectName)
    Status ::= "current" | "deprecated" | "obsolete"
    ReferPart ::= "REFERENCE" Text | empty
    -- uses the NVT ASCII character set
    Text ::= """" string """"
END
```

Table 9-6 Structure of NOTIFICATION-TYPE Macro

The instances of NOTIFICATION-TYPE macro whose STATUS clause is either *deprecated* or *obsolete* are ignored.

SNMPv1 traps are mapped as described in Chapter 10 on page 153.

9.5.1 Mapping of the OBJECTS clause

The OBJECTS clause, which need not be present, defines the ordered sequence of MIB objects which are contained within every instances of notification.

If the ObjectsPart clause is present then the value-specification of this clause is mapped to an IDL struct (to be used as event-data) as follows:

- Define a new IDL struct (based on the <name, index, value> triplet) for each of the variable in the value-specification of the clause. The identifier of the new struct type is derived by capitalising the first character of the variable and then appending **Type** to it. The type of name and index elements are **ASN1_ObjectIdentifier**, and the type of the value is derived from the value of the SYNTAX clause of the corresponding object.
  - — If the same variable is defined in more than one NOTIFICATION-TYPE macro within same SNMPv2 module, then ignore this step for this variable when it is defined for the second and subsequent time.
- Define an IDL struct where the identifier of elements of the struct are defined based on the name of the object in the value specification of the clause. The types of the elements of the IDL struct are based on IDL type defined for the corresponding object (as defined above). The identifier for the new IDL type is formed using following rules:
  - — capitalise the first character of the descriptor for the NOTIFICATION-TYPE macro
  - — append "Type" string to the capitalised macro-descriptor.

The new IDL types (as described above) are defined within the scope of the corresponding IDL module.

The main advantage of mapping the `OBJECTS` clause to an IDL struct is that the information about the objects can be used in the `NOTIFICATION-TYPE` macro as event information. This event information can be exchanged both as typed or untyped event data based on the OMG Event Services Specification (see reference **ESS**).

9.5.2 Mapping of the DESCRIPTION Clause

The `DESCRIPTION` clause, which must be present, contains a textual definition of the notification which provides all semantic definition necessary for implementation. The value-specification for this clause is mapped as a comment above the IDL operations (both push and pull modes), in the following form:

```
/*
DESCRIPTION: <value-specification>
*/
```

9.5.3 Mapping of the REFERENCE Clause

The `REFERENCE` clause, which need not be present, contains a textual cross-reference to an notification defined in some other module. The value-specification for this clause is mapped as comment above the IDL operations (both push and pull modes) in the following form:

```
/*
REFERENCE: <value-specification>
*/
```

9.5.4 Mapping of the NOTIFICATION-TYPE Value

The value of an invocation of the `NOTIFICATION-TYPE` is an `OBJECT IDENTIFIER` and this value is used as the authoritative registration identifier for referencing.

The value of the invocation of the `NOTIFICATION-TYPE` macro is used to derive the ID for the **#pragma** declaration for the identifier of the corresponding operations within the interface for both push and pull style event communication.

For **push** operations in the **SnmpNotifcation** interface (for push-style communication), the **#pragma** ID is derived from the OID of `NOTIFICATION-TYPE` macro by appending **::push** to OIDvalue.

For **pull** operations in the **PullSnmpNotifcation** interface (for pull-style communication), the **#pragma** ID is derived from the OID of `NOTIFICATION-TYPE` macro by appending **::pull** to the OID value.

For **try** operations in **PullSnmpNotifcation** interface (for pull-style communication), the **#pragma** ID is derived from the OID of the `NOTIFICATION-TYPE` macro by appending **::try** to the OID value.

9.5.5 Generation of Operation for Typed-Push Event Communication

Define an operation within the scope of **SnmpNotifications** interface of this module for typed-push event communication. The mandatory *in* parameters of the operation are *source SNMP*, *party OID*, *context OID* and *#time-stamp"* (of the event). See the example in Section 9.5.6.1 on page 143 for the types of these parameters. The optional *in* parameter of the operation is the IDL struct defined for the value-specification of the `OBJECTS` clause, as described in Section 9.5.1 on page 140. If the `OBJECT` clause is not present, then this parameter is not generated. The descriptor of the macro is used as the identifier for the operation. The return value of the operation is of type void.

9.5.6 Generation of Operation for Typed-Pull Event Communication

Define two operations (**try_<op>** and **pull_<op>**) within the scope of the **PullSnmpNotifications** interface of this module, where <op> is the identifier of the corresponding operation in the **SnmpNotifications** interface. The *in* parameter of the **push** operation will be converted to an *out* parameter. The return value of **try_<op>** is boolean and **pull_<op>** is void. See the ''Typed Pull Model'' section of the OMG Event Services Specification (see reference **ESS**) for details about defining an interface for typed-pull event communication.

9.5.6.1 Example

Consider the mapping of an SNMPv2 Notification-Type macro, called `linkUp`, into an IDL interface. The SNMP definition is as follows:

```
SNMPv2-MIB DEFINITIONS ::= BEGIN
 ......
coldStart NOTIFICATION-TYPE
  STATUS  current
  DESCRIPTION
  "A coldStart trap signifies that the SNMPv2 entity, acting in an agent
  role, is reinitializing
  itself such that its configuration may be altered."
  ::= { snmpTraps 1 }

warmStart NOTIFICATION-TYPE
  STATUS  current
  DESCRIPTION
  "A warmStart trap signifies that the SNMPv2 entity, acting in an agent
  role, is reinitializing
  itself such that its configuration is unaltered."
  ::= { snmpTraps 2 }

linkDown NOTIFICATION-TYPE
  OBJECTS { ifIndex }
  STATUS  current
  DESCRIPTION
  "A linkDown trap signifies that the SNMPv2 entity, acting in an agent
  role, recognises a failure
  in one of the communication links represented in its configuration."
  ::= { snmpTraps 3 }

linkUp NOTIFICATION-TYPE
  OBJECTS { ifIndex }
  STATUS current
  DESCRIPTION
  "A linkUp trap signifies that the SNMPv2 entity, acting in an agent
  role, recognises that one of
  the communication links represented in its configuration has come up."
  ::= { snmpTraps 4 }
 ......
END
```

The following IDL will be generated. Note that an IDL type called **LinkUpType** is defined. The element of the struct is defined based on the IDL type **IfIndexType**, generated from the object *ifIndex*. The type of ''var_value'' item in **IfIndexType** is obtained from the value-specification of the `SYNTAX` clause of the `OBJECT-TYPE` macro, called ''ifIndex''. Only one **IfIndexType** is generated, even though ifIndex object is present in both linkUp and linkDown notification macros. For coldStart and warmStart macros, no IDL type is generated since the `OBJECT` clause

is not present in the macro and as such no parameter is defined in the corresponding operations.

```
module SNMPv2_MIB {
 ......
IfIndexType  {
   ASN1_ObjectIdentifier var_name; // Only Instance Info part
   ASN1_ObjectIdentifier var_index; // Only Instance Info part
   ASN1_INTEGER var_value;
};

struct LinkDownType {
   IfIndexType ifIndex;
};

struct LinkUpType {
   IfIndexType ifIndex;
};

interface SnmpNotifications { // for typed-push event communication
   /*DESCRIPTION:
      "A coldStart trap signifies that the SNMPv2 entity,
      acting in an agent role, is reinitializing itself
      such that its configuration may be altered"
   */
   void coldStart (
      in ASN1_ObjectIdentifier src_party,
      in ASN1_ObjectIdentifier snmp_context,
      in SNMPv2TC::TimeStampType event_time
   );
   #pragma ID coldStart "1.3.6.1.6.3.1.1.5.1: :push"

   /* DESCRIPTION:
      "A warmStart trap signifies that the SNMPv2 entity,
      acting in an agent role, is reinitializing itself
      such that its configuration is unaltered."
   */
   void warmStart (
      in ASN1_ObjectIdentifier src_party,
      in ASN1_ObjectIdentifier snmp_context,
      in SNMPv2TC::TimeStampType event_time
   );
   #pragma ID coldStart "1.3.6.1.6.3.1.1.5.2: :push"

   /* DESCRIPTION:
      "A linkDown trap signifies that the SNMPv2 entity,
      acting in an agent role, recognizes a failure in
      one of the communication links represented in
      its configuration."
   */
   void linkdown (
      in ASN1_ObjectIdentifier src_party,
      in ASN1_ObjectIdentifier snmp_context,
      in SNMPv2TC::TimeStampType event_time,
      in LinkDownType notification_info
   );
   #pragma ID linkDown "1.3.6.1.6.3.1.1.5.3: :push"

   /*
   DESCRIPTION:
```

```
            "A linkUp trap signifies that the SNMPv2 entity,
            acting in an agent role, recognises that one of
            the communication links represented in its
            configuration has come up."
        */
        void linkUp (
            in ASN1_ObjectIdentifier src_party,
            in ASN1_ObjectIdentifier snmp_context,
            in SNMPv2TC::TimeStampType event_time,
            in LinkUpType notification_info
        );
        #pragma ID linkUp "1.3.6.1.6.3.1.1.5.4: :push"
    };

    interface PullSnmpNotifications { // for typed-pull event communication
        void pull_coldStart (
            out ASN1_ObjectIdentifier src_party,
            out ASN1_ObjectIdentifier snmp_context,
            out SNMPv2TC::TimeStampType event_time
        );
        #pragma ID pull_coldStart "1.3.6.1.6.3.1.1.5.1: :pull"

        boolean try_coldStart (
            out ASN1_ObjectIdentifier src_party,
            out ASN1_ObjectIdentifier snmp_context,
            out SNMPv2TC::TimeStampType event_time
        );
        #pragma ID try_coldStart "1.3.6.1.6.3.1.1.5.1: :try"

        void pull_warmStart (
            out ASN1_ObjectIdentifier src_party,
            out ASN1_ObjectIdentifier snmp_context,
            out SNMPv2TC::TimeStampType event_time,
        );
        #pragma ID pull_warmStart "1.3.6.1.6.3.1.1.5.2: :pull"

        boolean try_warmStart (
            out ASN1_ObjectIdentifier src_party,
            out ASN1_ObjectIdentifier snmp_context,
            out SNMPv2TC::TimeStampType event_time
        );
        #pragma ID try_warmStart "1.3.6.1.6.3.1.1.5.2: :try"

        void pull_linkdown (
            out ASN1_ObjectIdentifier src_party,
            out ASN1_ObjectIdentifier snmp_context,
            out SNMPv2TC::TimeStampType event_time,
            out LinkDownType notification_info
        );
        #pragma ID pull_linkDown "1.3.6.1.6.3.1.1.5.3: :pull"

        boolean try_linkdown (
            out ASN1_ObjectIdentifier src_party,
            out ASN1_ObjectIdentifier snmp_context,
            out SNMPv2TC::TimeStampType event_time,
            out LinkDownType notification_info
        );
        #pragma ID try_linkDown "1.3.6.1.6.3.1.1.5.3: :try"
```

```
    void pull_linkUp (
       out ASN1_ObjectIdentifier src_party,
       out ASN1_ObjectIdentifier snmp_context,
       out SNMPv2TC::TimeStampType event_time,
       out LinkUpType notification_info
    );
    #pragma ID pull_linkUp "1.3.6.1.6.3.1.1.5.4: :pull"

    boolean try_linkUp (
       out ASN1_ObjectIdentifier src_party,
       out ASN1_ObjectIdentifier snmp_context,
       out SNMPv2TC::TimeStampType event_time,
       out LinkUpType notification_info
    );
    #pragma ID try_linkUp "1.3.6.1.6.3.1.1.5.4: :try"
}
```

9.5.7 Operation Signatures for Typed-push/Typed-pull

Section 12.3 on page 171 also defines the operation signatures for the typed-push and typed-pull event communication. The operation signature for typed-push communication is **linkUp**( ) which is defined within the scope of the **SnmpNotifications** interface. The operation signatures for typed-pull communication are **pull_linkUp**( ) and **try_linkup**( ) which are defined within the scope of the **PullSnmpNotifications** interface. Note that there will be only one interface for typed-push event communication and only one for typed-pull event communication, for each SNMP information module. During the connection set-up for event channel, the fully-scoped IDL interface name (that is, **<moduleName>::SnmpNotifications** and **<moduleName>::PullSnmpNotifications**) would be passed.

9.6 SNMPv2 TEXTUAL-CONVENTION Macros

The TEXTUAL-CONVENTION macro is used to convey the syntax and semantics associated with a textual convention. Only those TEXTUAL-CONVENTION macros are considered for mapping whose STATUS clause value specification is current. Table 9-7 describes the structure of the TEXTUAL-CONVENTION macro.

```
TEXTUAL-CONVENTION MACRO ::=
BEGIN
    TYPE NOTATION ::=
        DisplayPart
        "STATUS" Status
        "DESCRIPTION" Text
        ReferPart
        "SYNTAX" type(Syntax)
    VALUE NOTATION ::= value (VALUE Syntax)
    DisplayPart ::= "DISPLAY-HINT" Text | empty
    Status ::= "current" | "deprecated" | "obsolete"
    ReferPart ::= "REFERENCE" Text | empty
    -- uses the NVT ASCII character set
    Text ::= """" string """"
END
```

Table 9-7 Structure of TEXTUAL CONVENTION Macro Clauses

Based on SYNTAX clause, an IDL type is defined for each TEXTUAL-CONVENTION macro. The identifier of the IDL type for an instance of TEXTUAL-CONVENTION macro is the corresponding identifier for the descriptor for the macro. If a DISPLAY-HINT clause is present, or the SYNTAX is an enumerated INTEGER value, then two operations are declared to convert the value of an object to a displayable string and the displayable string to the typed value.

9.6.1 Mapping of the SYNTAX Clause

The SYNTAX clause, which must be present, defines the abstract data-structure corresponding to the textual convention. The data structure must be one of the alternatives defined in the ObjectSyntax CHOICE. Full ASN.1 subtyping is allowed.

The type defined in the SYNTAX clause is first mapped to an ASN.1 type, where the identifier of the ASN.1 type is the descriptor for the TEXTUAL-CONVENTION macro, and then converted to IDL by using the ASN1 to IDL translation scheme defined in Chapter 2 on page 15. The IDL type is defined within the scope of the IDL module for the SNMPv2 module for Textual-Convention. Note that as the ASN.1 to IDL translation scheme is used, a **Type** suffix will be added to the ASN.1 type name in order to generate the identifier for the IDL type.

If the value specification clause is an enumerated INTEGER value, then two operations will be declared (as described in Section 9.6.2 on page 148) within the scope of TextualConvention interface for mapping the integer value to an enumerated value and vice-versa.

9.6.2 Mapping of the DISPLAY-HINT Clause

The `DISPLAY-HINT` clause, which need not be present, gives a hint as to how the value of an instance of an object with the syntax defined using this textual convention might be displayed. If this clause is present then two operations are declared to convert the value of an object to a displayable string and the displayable string to the typed value. The value-specification for this clause is mapped as IDL block comment above the operation declarations:

```
/* pseudo */
interface TextualConventions {
   /*
   DISPLAY-HINT : <value-specification>.
   DESCRIPTION: <value-specification> of DESCRIPTION clause
   */
   string <macro-descriptor>ToString (in <Type> Value);
   <Type> <macro-descriptor>FromString (in string str);
};
```

<Type> is derived from the `SYNTAX` clause as defined in Section 9.6.1 on page 147.

9.6.3 Mapping of the STATUS Clause

The `STATUS` clause, which must be present, indicates whether this definition is current or historic. The value-specification of this clause is used to determine if an IDL type will be defined for this macro. Only those `TEXTUAL-CONVENTION` macro whose value-specification is current is mapped to IDL. All others are ignored.

9.6.4 Mapping of the DESCRIPTION Clause

The `DESCRIPTION` clause, which must be present, contains a textual definition of the textual convention which provides all semantic definition necessary for implementation. The value-specification for this clause is mapped as IDL comment below the IDL type in the following form:

```
/*
DESCRIPTION : <value-specification>
*/
```

If the `DISPLAY-HINT` clause is present, the value-specification of this is also declared as a block comment before the declaration of the operations for the `DISPLAY-HINT` clause.

9.6.5 Mapping of the REFERENCE Clause

The `REFERENCE` clause, which need not be present, contains a textual cross-reference to a related item defined in some other published work. The value-specification for this clause is mapped as IDL comment below the IDL type in the following form:

```
/*
REFERENCE : <value-specification>
*/
```

9.6.6 Example 1

Example 9-8 illustrates the mapping of a TEXTUAL-CONVENTION macro, called *DisplayString*, into the associated IDL type. The IDL type **DisplayStringType** is mapped from the following ASN.1 type:

```
DisplayString ::= OCTET STRING (SIZE (0..255))
```

Example 9-8 Conversion of SNMP TextualConvention DisplayString

Textual Convention	IDL Interface
`DisplayString ::= TEXTUAL-CONVENTION` `  DISPLAY-HINT "255a"` `  STATUS current` `  DESCRIPTION` `  "Represents textual information` `  taken from the NVT ASCII character` `  set, as defined in pages 4, 10-11` `  of RFC 854. Any object defined` `  using this syntax may not exceed` `  255 characters in length."` `  SYNTAX OCTET STRING (SIZE (0..255))`	**// Following ASN.1 type is derived** **// from SYNTAX clause of DisplayString** **// and mapped to IDL type** **// DisplayString ::= OCTET STRING (SIZE (0..255))** **typedef sequence<octet, 256> DisplayStringType;** **/*** **DISPLAY-HINT = 255a** **DESCRIPTION:** **"Represents textual information** **taken from the NVT ASCII character** **set, as defined in pages 4, 10-11** **of RFC 854. Any object defined** **using this syntax may not exceed** **255 characters in length."** ***/** **/* pseudo */** **interface TextualConventions {** **/*** **DISPLAY-HINT = 255a** **DESCRIPTION:** **"Represents textual information** **taken from the NVT ASCII character** **set, as defined in pages 4, 10-11** **of RFC 854. Any object defined** **using this syntax may not exceed** **255 characters in length."** ***/** **string DisplayStringToString(in DisplayStringType value);** **DisplayStringType DisplayStringFromString(in string str);** **};**

9.6.7 Example 2

Example 9-9 illustrates the mapping of a TEXTUAL-CONVENTION macro, called *TruthValue*, into an IDL type. The IDL type **TruthValueType** is mapped from the following ASN.1 type:

```
TruthValue ::= INTEGER { true (1) , false (2) }
```

Example 9-9 Conversion of SNMP TextualConvention

Textual Convention	IDL Interface
`TruthValue ::= TEXTUAL-CONVENTION` `  STATUS current` `  DESCRIPTION` `  "Represents a boolean value."` `  SYNTAX INTEGER { true (1) , false (2) }`	**// Following ASN.1 type is derived from SYNTAX** **// clause of TruthValue and mapped to IDL type** **// TruthValue ::= INTEGER { true (1) , false (2) }** **typedef ASN1_Integer TruthValueType;** **const TruthValueType true = 1;** **const TruthValueType false = 2;** **const string TruthValue_NameNumberList =** **" true (1) , false (2) ";** **/*** **DESCRIPTION = "Represents a boolean value."** ***/** **/* pseudo */** **interface TextualConventions {** **/*** **DESCRIPTION = Represents a boolean value."** ***/** **string TruthValueToString(in TruthValueType value);** **TruthValueType TruthValueFromString(in string str);** **};**

9.6.8 Example 3

Example 9-10 also illustrates the mapping of a `TEXTUAL-CONVENTION` macro, called *TimeStamp*, into the IDL type **TimeStampType**.

Example 9-10 Conversion of SNMP TextualConvention

Textual Convention	IDL Interface
`TimeStamp ::= TEXTUAL-CONVENTION` `  STATUS current` `  DESCRIPTION` `  ""` `  SYNTAX TimeTicks`	**// Following ASN.1 type is derived from SYNTAX clause** **// of TimeStamp and mapped to IDL type** **// TimeStamp ::= TimeTicks** **typedef TimeTicksType TimeStampType;** **/*** **DESCRIPTION** **"The value of MIB-II's sysUpTime object at which a specific occurrence happened. The specific occurrence must be defined in description of any object defined using this type."** ***/**

9.7 SNMPv2 MODULE-COMPLIANCE Macros

The `MODULE-COMPLIANCE` macro is used to convey a minimum set of requirements with respect to implementation of one or more MIB modules.

Since IDL interfaces do not specify implementation requirements, the macros related to `MODULE-COMPLIANCE` macro in a SNMPv2 information module are not mapped to IDL.

Chapter 10

Mapping of SNMPv1 Traps

10.1 SNMPv1 Traps

Although the mapping of tables for SNMPv2 can also be applied to the SNMPv1 specs, the same can not be said for SNMPv1 traps. The SNMP-to-IDL mapping addresses the MIB specification written for SNMPv2.

SNMPv1 traps definitions are different from SNMPv2 Notification in specification as well as in the PDU structure. In SNMPv1, traps are defined using TRAP-TYPE macros (see Table 10-1) which are described in RFC1215.

```
ourcompany OBJECT IDENTIFIER ::= { enterprises 9999 }
myAlarm TRAP-TYPE
    ENTERPRISE ourcompany
    VARIABLES { alarmReason }
    DESCRIPTION ""
    ::= 1
```

Table 10-1 TRAP-TYPE Macro in SNMPv1

The SNMPv1Trap.idl file (see Section 12.4 on page 172) is defined to support conversion of traps in SNMPv1 to OMG Event Services specification and vice-versa. This file describes data types (**SNMPv1_TrapInfo**) and interfaces (**SNMPv1_Notification** and **PullSNMPv1_Notification**) to support conversion of SNMPv1 traps to untyped events in the CORBA domain.

The data types are defined based on the data structure of Trap PDU in SNMPv1 and the TRAP-TYPE macro defined in RFC1215. A CORBA-based object implementation can ignore the first two parameters of **SNMPv1_TrapInfo** (*agent_ip_address* and *community_name*) of the event information. In that case, the CORBA / SNMP gateway must set them before forwarding an event in the CORBA domain as a trap in the SNMP domain.

10.2 Mapping of TRAP-TYPE Macro in SNMPv1

The mapping of `TRAP-TYPE` macro for SNMPv1 specification to IDL operation is similar to the mapping of `NOTIFICATION-TYPE` macro in SNMPv2 specification. In both cases, a set of operations are generated within the scope of the **SnmpNotifications** and the **PullSnmpNotifications** interfaces. The `VARIABLES` clause is mapped in the same way as the `OBJECTS` clause in the `NOTIFICATION-TYPE` macro. Since the `TRAP-TYPE` macro does not assign any OID to the trap definition, an OID is created based on the `ENTERPRISE` clause and the integer value of the macro. The other difference is that the parameters of the IDL operation are generated based on the SNMPv1 Trap PDU as described in Chapter 10 on page 153.

10.2.1 Deriving Repository ID of IDL Operations for Traps

It is necessary to assign a repository ID for IDL operations for Traps using **#pragma ID**, such that it is possible to recover the information about the enterprise and the specific trap type in the SNMP domain from a trap originated in the CORBA domain.

The repository ID is derived based on the value of the `ENTERPRISE` clause of the `TRAP-TYPE` macro and the integer value of the invocation of the macro. The repository ID is formed using the following format.

"<enterprise-oid>.Traps.<trap-type-macro-value>::<push|pull|try>"

10.2.2 Mapping of TRAP-TYPE Macros for Generic Traps

In order to support SNMPv1 generic traps using typed interfaces for event communication in the CORBA domain, a set of operations are defined based on the usage examples in section 2.2.2 of RFC1215.

10.2.3 Example: Generic Traps

The following example illustrates the mapping of the `TRAP-TYPE` macros (as defined in RFC1215) for generic traps. Although the Repository ID for the generic traps are defined based on Section 10.2.1, it is necessary to convert the enterprise oid to the sysObjectID (as described in section 2.1.1 of RFC1215) at the gateway. Thus, a CORBA/SNMP gateway has to detect if a trap in the SNMP domain or event in the CORBA domain is based on a generic trap before forwarding it to the other domain. Since the `ENTERPRISE` clause of a generic trap is always RFC1213.snmp (1.3.6.1.2.1.11), it is easy to detect if an event is based on a generic trap by looking at the Repository ID of the operations in the CORBA domain. Similarly if the generic-trap in an SNMPv1 PDU is one of the generic trap types, it can easily generate the Repository ID of the corresponding IDL operation from the SNMPv1 Trap-PDU.

```
SNMPv1-GenericTraps  DEFINITIONS ::= BEGIN
 ....
coldStart NOTIFICATION-TYPE
    ENTERPRISE snmp
    DESCRIPTION
    "A coldStart trap signifies that the
        SNMPv2 entity, acting in an agent role,
        is reinitializing itself such that its
        configuration may be altered."
    ::= 0

warmStart NOTIFICATION-TYPE
    ENTERPRISE snmp
    DESCRIPTION
    "A warmStart trap signifies that the
        SNMPv2 entity, acting in an agent role,
        is reinitializing itself such that its
        configuration is unaltered."
    ::= 1

linkDown NOTIFICATION-TYPE
    ENTERPRISE snmp
    VARIABLES { ifIndex }
    DESCRIPTION
    "A linkDown trap signifies that the
        SNMPv2 entity, acting in an agent role,
        recognizes a failure in one of the
        communication links represented in its
        configuration."
    ::= 2

linkUp NOTIFICATION-TYPE
    ENTERPRISE snmp
    VARIABLES { ifIndex }
    DESCRIPTION
    "A linkUp trap signifies that the
        SNMPv2 entity, acting in an agent role,
        recognizes that one of the communication
        links represented in its configuration
        has come up."
    ::= 3

    ....
END
```

maps to the following IDL:

```
// File name is SNMPv1_GenericTraps.idl
module SNMPv1_GenericTraps
{
 ..............
  struct IfIndexType {
    ASN1_ObjectIdentifier var_name;
      // IDL Scoped-Name of the object
    ASN1_ObjectIdentifier var_index;
      // Only Instance Info part
    ASN1_INTEGER var_value;
  };

  struct LinkDownType {
    IfIndexType ifIndex;
  };

  struct LinkUpType {
    IfIndexType ifIndex;
  };

  interface SnmpNotifications {
    // for typed-push event communication
    void coldStart (
      in string agent_ip_address,
      in string community_name,
      in ASN1_ObjectIdentifier snmp_context,
      in RFC1155_SMI::TimeTicksType event_time
    );
    #pragma ID coldStart "1.3.6.1.2.1.11.Traps.0::push"

    void warmStart (
      in string agent_ip_address,
      in string community_name,
      in RFC1155_SMI::TimeTicksType event_time
    );
    #pragma ID warmStart "1.3.6.1.2.1.11.Traps.1::push"

    void linkDown (
      in string agent_ip_address,
      in string community_name,
      in RFC1155_SMI::TimeTicksType event_time,
      in LinkDownType notification_info
    );
    #pragma ID linkDown "1.3.6.1.2.1.11.Traps.2::push"

    void linkUp (
      in string agent_ip_address,
      in string community_name,
      in RFC1155_SMI::TimeTicksType event_time,
      in LinkUpType notification_info
    );
    #pragma ID linkUp "1.3.6.1.2.1.11.Traps.3::push"
  };

  interface PullSnmpNotifications {
    // for typed-pull event communication
    void pull_coldStart (
      out string agent_ip_address,
```

```
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time
  );
  #pragma ID pull_coldStart "1.3.6.1.2.1.11.Traps.0::pull"

  boolean try_coldStart (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time
  );
  #pragma ID try_coldStart "1.3.6.1.2.1.11.Traps.0::try"

  void pull_warmStart (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time
  );
  #pragma ID pull_warmStart "1.3.6.1.2.1.11.Traps.1::pull"

  boolean try_warmStart (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time
  );
  #pragma ID try_warmStart "1.3.6.1.2.1.11.Traps.1::try"

  void pull_linkdown (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time,
    out LinkDownType notification_info
  );
  #pragma ID pull_linkDown "1.3.6.1.2.1.11.Traps.2::pull"

  boolean try_linkdown (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time,
    out LinkDownType notification_info
  );
  #pragma ID try_linkDown "1.3.6.1.2.1.11.Traps.2::try"

  void pull_linkUp (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time,
    out LinkUpType notification_info
  );
  #pragma ID pull_linkUp "1.3.6.1.2.1.11.Traps.3::pull"

  boolean try_linkUp (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time,
    out LinkUpType notification_info
  );
  #pragma ID try_linkUp "1.3.6.1.2.1.11.Traps.3::try"
};
```

}; // End of module SNMPv2_MIB

10.2.4 Example: Enterprise-specific Trap

The following example illustrates the mapping of the enterprise-specific traps in SNMPv1 information modules.

```
SNMPv1-TrapExamples  DEFINITIONS ::= BEGIN
  ....
  ourcompany OBJECT IDENTIFIER
        ::= { enterprises 9999 }
  myAlarm TRAP-TYPE -- notice that they use
                    -- a TRAP-TYPE macro
    ENTERPRISE ourcompany
  VARIABLES { alarmReason }
  DESCRIPTION ""
        ::= 1 -- Repository ID of this trap
              -- is derived by concatenating
              -- the OID in ENTERPRISE clause,
              -- "Traps" string,
              -- and the Id number
END
```

maps to:

```
module SNMPv1_TrapExamples
{
 typedef sequence<octet, 255> DisplayString;
 struct AlarmReasonType {
       // Derived from variables in Object Clause
  ASN1_ObjectIdentifier var_name;
       // IDL Scoped-Name of the object
  ASN1_ObjectIdentifier var_index;
       // Only Instance Info part
  DisplayString var_value;
 };
 struct MyAlarmType {
       // Derived from variables in Object Clause
  AlarmReasonType alarmReason;
 };

 interface SnmpNotifications {
       // for typed-push event communication
  void myAlarm (
    in string agent_ip_address,
    in string community_name,
    in RFC1155_SMI::TimeTicksType event_time,
    in MyAlarmType notification_info
  );
  #pragma ID myAlarm "1.3.6.1.4.1.11.9999.Traps.1::push"
 };

 interface PullSnmpNotifications {
  // follow the SNMPv2 notification
  // macro mapping for operations in
  // PullSnmpNotifications interface.
  void pull_myAlarm (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time,
    out MyAlarmType notification_info
  );
  #pragma ID pull_myAlarm "1.3.6.1.4.1.11.9999.Traps.1::pull"

    boolean try_myAlarm (
    out string agent_ip_address,
    out string community_name,
    out RFC1155_SMI::TimeTicksType event_time,
    out MyAlarmType notification_info
  );
  #pragma ID try_myAlarm "1.3.6.1.4.1.11.9999.Traps.1::try"
 };
}; // End of SNMPv1_TrapExamples module
```

Preliminary Specification

Part 6:
OMG IDL to SNMP Translation Algorithm

The Open Group

Chapter 11

OMG IDL to SNMP Translation

During the development of this specification, it was felt that there was no requirement for translating from OMG IDL to SNMP. Accordingly, this part of the document is intentionally left blank.

If it is subsequently decided that such a translation requirement does exist after all, and suitable text is available, this part of the document may be completed in a future release.

Preliminary Specification

Part 7:
IDL Modules and Examples

The Open Group

Chapter 12

Basic Definitions

This part of the document provides informative examples of the application of the algorithms defined within the document.

In the case of any discrepancies between the examples and the specification of the algorithms, the specification is to be regarded as definitive.

ASN1Types.idl and ASN1Limits.idl are required by all implementations. OSIMgmt.idl is specific to OSI implementations, and the SNMP files SNMPMgmt.idl and SNMPv1Trap.idl are both specific to SNMP implementations.

12.1 Basic IDL Definitions

The following sections in this chapter illustrate the basic IDL definitions.

12.1.1 ASN1Types.idl File

```
//
// ASN1Types.idl
//

#ifndef _ASN1TYPES_IDL_
#define _ASN1TYPES_IDL_

// ASN.1 base types

// Null type
typedef char                ASN1_Null;
const ASN1_Null             ASN1_NullValue = '\x00';

typedef boolean             ASN1_Boolean;

// unsigned integers
typedef unsigned short      ASN1_Unsigned16;
typedef unsigned long       ASN1_Unsigned;
typedef unsigned long       ASN1_Unsigned64[2];

// integers
typedef short               ASN1_Integer16;
typedef long                ASN1_Integer;
typedef long                ASN1_Integer64[2];

// others
typedef double              ASN1_Real;
typedef sequence<octet>     ASN1_BitString; // PIDL defined
typedef sequence<octet>     ASN1_OctetString;
typedef string              ASN1_ObjectIdentifier;
typedef any                 ASN1_Any;
typedef any                 ASN1_DefinedAny;

struct ASN1_External {
    ASN1_ObjectIdentifier  syntax;
    ASN1_DefinedAny        data_value;  // by syntax
```

```
};

// ASN.1 strings (which may not contain binary zeros)

typedef string                  ASN1_IA5String;
typedef string                  ASN1_NumericString;
typedef string                  ASN1_PrintableString;
typedef string                  ASN1_TeletexString;
typedef string                  ASN1_T61String;
typedef string                  ASN1_VideotexString;
typedef string                  ASN1_VisibleString;
typedef string                  ASN1_ISO646String;

// PIDL defined
typedef ASN1_VisibleString              ASN1_GeneralizedTime;
typedef ASN1_VisibleString              ASN1_UTCTime;

// ASN.1 strings of octets (which may contain binary zeros)

typedef sequence<octet>                 ASN1_GeneralString;
typedef sequence<octet>                 ASN1_GraphicString;

// ASN.1 strings of wide characters (which may contain binary zeros)

typedef sequence<unsigned short>        ASN1_BMPString;
typedef sequence<unsigned long>         ASN1_UniversalString;

typedef ASN1_GraphicString              ASN1_ObjectDescriptor;

// define constants for ASN.1 Real infinity

#include<ASN1Limits.idl>

#endif /*  _ASN1TYPES_IDL_ */
```

12.1.2 ASN1Limits.idl File

```
//
// ASN1Limits.idl
//

#ifndef _ASN1LIMITS_IDL_
#define _ASN1LIMITS_IDL_

// Substitute <MAX> and <MIN> by the max and min (lowest negative) float values your
// machine can hold for IDL interfaces.

const  ASN1_Real  plus_infinity   = <MAX>;
const  ASN1_Real  minus_infinity = <MIN>;

#endif /*  _ASN1LIMITS_IDL_ */
```

12.2 OSIMgmt.idl File

```
//
// OSIMgmt.idl
//

#ifndef _OSIMGMT_IDL_
#define _OSIMGMT_IDL_


// include all needed data types

#include <ASN1Types.idl>          // base ASN1 types
#include <X711CMI.idl>            // the types defined in the CMIP ASN.1 module

module OSIMgmt
{

 // OSIMgmt::exceptions
 // Corba User exceptions based on ROSE and CMIS.

 // ROSE originated exceptions

 exception ROSEerror                    { X711CMI::InvokeProblemType          errorInfo; };

 // CMIS originated exceptions

 exception AccessDenied                 { };
 exception ClassInstanceConflict        { X711CMI::BaseManagedObjectIdType  errorInfo; };
 exception ComplexityLimitation         { X711CMI::ComplexityLimitationType errorInfo; };
 exception GetListError                 { X711CMI::GetListErrorType         errorInfo; };
 exception InvalidArgumentValue         { X711CMI::InvalidArgumentValueType errorInfo; };
 exception InvalidFilter                { X711CMI::CMISFilterType           errorInfo; };
 exception InvalidScope                 { X711CMI::ScopeType                errorInfo; };
 exception InvalidObjectInstance        { X711CMI::ObjectInstanceType       errorInfo; };
 exception NoSuchAction                 { X711CMI::NoSuchActionType         errorInfo; };
 exception NoSuchArgument               { X711CMI::NoSuchArgumentType       errorInfo; };
 exception NoSuchAttribute              { X711CMI::AttributeIdType          errorInfo; };
 exception NoSuchObjectClass            { X711CMI::ObjectClassType          errorInfo; };
 exception NoSuchObjectInstance         { X711CMI::ObjectInstanceType       errorInfo; };
 exception NoSuchReferenceObject        { X711CMI::ObjectInstanceType       errorInfo; };
 exception ProcessingFailure            { X711CMI::ProcessingFailureType    errorInfo; };
 exception SetListError                 { X711CMI::SetListErrorType         errorInfo; };
 exception SyncNotSupported             { X711CMI::CMISSyncType             errorInfo; };

 //
 // ManagedObject
 // the base interface for all generated managed object class interfaces
 // Its visible attributes and methods will be defined during the
 // Interaction Translation specification phase of the NMF-X/Open JIDM working
 // group.
 //

 interface ManagedObject
 {

   // to be defined in the Interaction Translation Specification

 };
```

```
  // exception for multiple replies to actions

  exception UsingMR{ /* to de defined in Interaction Translation */ };

};

// macros for use in the 'raises' clause of interface methods:

#define ACTION_ERRORS OSIMgmt::ROSEerror, \
   OSIMgmt::AccessDenied, OSIMgmt::ClassInstanceConflict, \
   OSIMgmt::ComplexityLimitation, OSIMgmt::InvalidScope, \
   OSIMgmt::InvalidArgumentValue, OSIMgmt::InvalidFilter, \
   OSIMgmt::NoSuchAction, OSIMgmt::NoSuchArgument, \
   OSIMgmt::NoSuchObjectClass, OSIMgmt::NoSuchObjectInstance, \
   OSIMgmt::ProcessingFailure, OSIMgmt::SyncNotSupported

#define ATTRIBUTE_ERRORS  OSIMgmt::ROSEerror, \
   OSIMgmt::AccessDenied, OSIMgmt::ClassInstanceConflict, \
   OSIMgmt::ComplexityLimitation, OSIMgmt::GetListError, \
   OSIMgmt::SetListError, OSIMgmt::InvalidScope, \
   OSIMgmt::InvalidFilter, OSIMgmt::NoSuchObjectClass, \
   OSIMgmt::NoSuchObjectInstance, OSIMgmt::ProcessingFailure, \
   OSIMgmt::SyncNotSupported

#define UsingMR  OSIMgmt::UsingMR

#endif /* _OSIMGMT_IDL_ */
```

12.3 SNMPMgmt.idl File

The IDL interface **SmiEntry** is stored in a file called SNMPMgmt.idl:

```
//
// SNMPMgmt.idl
//

#ifndef _SNMPMGMT_IDL_
#define _SNMPMGMT_IDL_

#include <SNMPv2_TC.idl>

module SNMPMgmt
{
 interface SmiEntry
 {
  // Note that no attribute or operations are defined.
 };

 struct SNMPv2_NotificationInfo {
  // to be sent when using untyped event channel
  ASN1_ObjectIdentifier src_party;
  ASN1_ObjectIdentifier snmp_context;
  ASN1_ObjectIdentifier event_type; // Repository ID of event
  SNMPv2TC::TimeStampType event_time; any notification_info;
 };

 interface SNMPv2_Notification {
  void snmp_notification (
    in ASN1_ObjectIdentifier src_party,
    in ASN1_ObjectIdentifier snmp_context,
    in ASN1_ObjectIdentifier event_type, // Repository ID of event
    in SNMPv2TC::TimeStampType event_time,
    in any notification_info
  );
 };

 interface PullSNMPv2_Notification {
  boolean try_snmp_notification (
    out ASN1_ObjectIdentifier src_party,
    out ASN1_ObjectIdentifier snmp_context,
    out ASN1_ObjectIdentifier event_type,
    out SNMPv2TC::TimeStampType event_time,
    out any notification_info
  );

  void pull_snmp_notification (
    out ASN1_ObjectIdentifier src_party,
    out ASN1_ObjectIdentifier snmp_context,
    out ASN1_ObjectIdentifier event_type,
    out SNMPv2TC::TimeStampType event_time,
    out any notification_info
  );
 };
}; /* End of Module */
#endif /* !_SNMPMgmt_idl_ */
```

12.4 SNMPv1Trap.idl File

The following file is defined in order to support SNMPv1 Traps:

```
//
// SNMPv1Trap.idl
//

#ifndef _SNMPv1Trap_idl_
#define _SNMPv1Trap_idl_
#include <SNMPMgmt.idl>
#include <RFC1155_SMI.idl>

module SNMPMv1Trap
{
  struct SNMPv1_TrapInfo { // to be sent when using untyped event channel
    string agent_ip_address;
    string community_name;
    ASN1_ObjectIdentifier event_type; // <enterprise-oid>.Traps.trap-id>
    RFC1155_SMI::TimeTicksType event_time;
    any notification_info;
  };

/*
  Example: (in the form of ASN.1 value in string)
  SNMPv1_TrapInfo : {
    agent_ip_address 999.00.60.14,
    community_name public,
    event_type 1.3.6.1.4.1.3.1.1,
    event_time 0,
    notification_info LinkUpType : {
      ifIndex {
        var_name "::RFC1213_MIB::ifEntry::ifIndex",
        var_index "1",
        var_value 1
      }
    }
  }
*/

  interface SNMPv1_Notifications {
    void snmpv1_trap(
      in string agent_ip_address,
      in string community_name,
      in ASN1_ObjectIdentifier event_type,
      in RFC1155_SMI::TimeTicksType event_time,
      in any notification_info
    );
  };

  interface PullSNMPv1_Notifications {
    boolean try_snmpv1_trap(
      out string agent_ip_address,
      out string community_name,
      out ASN1_ObjectIdentifier event_type,
      out RFC1155_SMI::TimeTicksType event_time,
      out any notification_info
    );
```

```
    void pull_snmpv1_trap(
      out string agent_ip_address,
      out string community_name,
      out ASN1_ObjectIdentifier event_type,
      out RFC1155_SMI::TimeTicksType event_time,
      out any notification_info
    );
  };
}; /* End of SNMPMv1Trap Module */
#endif /* !_SNMPv1Trap_idl_ */
```

Chapter 13

Translation of X.721 and X.722 Modules

It is intended that example translations of important GDMO specifications will be included when this document is revised to become a CAE Specification.

As an interim measure, examples may be posted on the World Wide Web, in the Distributed Systems Management area, on URL:

http://www.opengroup.org

Chapter 14

Mapping of SNMPv2 RFC Modules

14.1 Mapping of SNMPv2-SMI (RFC1442)

SNMPv2-SMI Module (RFC1442) is mapped to a file named **SNMPv2_SMI.idl**.

This **SNMPv2_SMI.idl** file contains the following IDL types and interfaces as translated from RFC 1442:

```
#ifndef _SNMPv2_SMI_idl
#define _SNMPv2_SMI_idl
#include <ASN1Types.idl>
#include <SNMPMgmt.idl>

module SNMPv2_SMI {

typedef long                    Integer32Type;
typedef unsigned long           UInteger32Type;

typedef sequence<octet, 4>      IpAddressType;
typedef unsigned long           Counter32Type;
typedef long                    Counter64Type[2];
typedef unsigned long           Gauge32Type;
typedef unsigned long           TimeTicksType;
typedef sequence<octet>         OpaqueType;
typedef ASN1_OctetString        NsapAddressType;

typedef ASN1_ObjectIdentifier   ObjectNameType;

enum SimpleSyntaxChoice {
   integerValueChoice,
   stringValueChoice,
   objectIDValueChoice,
   bitValueChoice
};

union SimpleSyntaxType switch(SimpleSyntaxChoice) {
   case integerValueChoice : ASN1_INTEGER integerValue;
   case stringValueChoice : ASN1_OctetString stringValue;
   case objectIDValueChoice : ASN1_ObjectIdentifier objectIDValue;
   case bitValueChoice : ASN1_BitString bitValue;
};

enum ApplicationSyntaxChoice {
   ipAddressValueChoice,
   counterValueChoice,
   gaugeValueChoice,
   timeticksValueChoice,
   arbitaryValueChoice,
   nsapAddressValueChoice,
   bigCounterValueChoice,
```

```
    unsignedIntegerValue
};

union ApplicationSyntaxType switch(ApplicationSyntaxChoice) {
    case ipAddressValueChoice : IpAddressType ipAddressValue;
    case counterValueChoice : Counter32Type counterValue;
    case gaugeValueChoice : Gauge32Type gaugeValue;
    case timeticksValueChoice : TimeTicksType timeticksValue;
    case arbitaryValueChoice : OpaqueType arbitaryValue;
    case nsapAddressValueChoice : NsapAddressType nsapAddressValue;
    case bigCounterValueChoice : Counter64Type bigCounterValue;
    case unsignedIntegerValueChoice : UInteger32Type unsignedIntegerValue;
};

enum ObjectSyntaxChoice {
    simpleChoice,
    applicationWideChoice
};

union ObjectSyntaxType switch(ObjectSyntaxChoice) {
    case simpleChice : SimpleSyntaxType simple;
    case applicationWideChoice : ApplicationSyntaxType application_wide;
};

}; /* End of SNMPv2_SMI Module.*/
```

14.2 Mapping of SNMPv2-TC (RFC1443)

SNMPv2-TC Module (RFC1443) is mapped to a file named **SNMPv2_TC.idl**, which contains the following IDL types and interfaces as translated from RFC 1443:

```
#ifndef _SNMPv2_TC_idl
#define _SNMPv2_TC_idl
#include <SNMPv2_SMI.idl>

module SNMPv2_TC {

typedef SNMPv2_SMI::ObjectSyntaxType ObjectSyntaxType;
typedef SNMPv2_SMI::Integer32Type Integer32Type;
typedef SNMPv2_SMI::TimeTicksType TimeTicksType;

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
// DisplayString ::= OCTET STRING (SIZE (0..255))
typedef sequence<octet, 256> DisplayStringType;

/*
DISPLAY-HINT: 255a
DESCRIPTION:
   Represents textual information taken from the NVT
   ASCII character set, as defined in pages 4, 10-11
   of RFC 854. Any object defined using this syntax
   may not exceed 255 characters in length.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
//    PhysAddress ::= OCTET STRING
typedef ASN1_OctetString PhysAddressType;

/*
DISPLAY-HINT: 1x:
DESCRIPTION: Represents a media or physical-level address.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
// MacAddress ::= OCTET STRING (SIZE (6))
typedef sequence<octet, 6> MacAddressType;

/*
DISPLAY-HINT: 1x:
DESCRIPTION:
   Represents an 802 MAC address represented in the
   canonical order defined by IEEE 802.1a, taht is,
   as if it were transmitted least significant bit
   first, even though 802.5 (in contrast to other
   802.x protocols) requires MAC addresses to be
   transmitted most significant bit first.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
// TruthValue ::= INTEGER { true (1) , false (2) }
```

```
typedef ASN1_INTEGER TruthValueType;

/*
DESCRIPTION: Represents a boolean value.
*/
const TruthValueType      true = 1;
const TruthValueType      false = 2;
const string TruthValue_NameNumberList = "true (1) , false (2)";

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
// TestAndIncr ::= INTEGER (0..2147483647)
typedef ASN1_INTEGER TestAndIncrType;

/*
DESCRIPTION:
   Represents integer-valued information used for
   atomic operations.
   See RFC 1443 for detailed description.
const string TestAndIncr_ValueRange = "0..2147483647";
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
//    AutonomousType ::=OBJECT IDENTIFIER
typedef ASN1_ObjectIdentifier AutonomousTypeType;

/*
DESCRIPTION:
   Represents an independently extensible type
   identification value. It may, for example,
   indicate a particular sub-tree with further MIB
   definitions, or define a particular type of
   protocol or hardware.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
//    InstancePointer ::= OBJECT IDENTIFIER
typedef ASN1_ObjectIdentifier InstancePointerType;

/*
DESCRIPTION:
   A pointer to a specific instance of a conceptual
   row of a MIB table in the managed device. By
   convention, it is the name of the particular
   instance of the first columnar object in the
   conceptual row.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
// TimeStamp = TimeTicks
typedef TimeTicksType TimeStampType;

/*
DESCRIPTION:
   The value of MIB-IIs sysUpTime object at which a
```

```
   specific occurrence happened. The specific
   occurrence must be defined in description of
   any object defined using this type.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
//     TimeInterval ::= INTEGER (0..2147483647)
typedef ASN1_INTEGER TimeIntervalType;

/*
DESCRIPTION: A period of time, measured in units of 0.01 seconds.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
// RowStatus ::= INTEGER {active(1), notInService(2),
//              notReady(3), createAndGo(4),
//              createAndWait(5), destroy(6) }
typedef ASN1_INTEGER RowStatusType;
const RowStatusType active = 1;
const RowStatusType notInService = 2;
const RowStatusType notReady = 3;
const RowStatusType createAndGo = 4;
const RowStatusType createAndWait = 5;
const RowStatusType destroy = 6;
const string RowStatus_NamedNumberList = ""
"active (1), notInService (2), notReady (3),
createAndGo (4), createAndWait (5), destroy (6)";

/*
DESCRIPTION:
   The RowStatus textual convention is used to manage
   the creation and deletion of conceptual
   rows, and is used as the value of the SYNTAX /clause
   for the status column of a conceptual row
   See RFC 1443 for detailed description.
*/

// Following ASN.1 type is derived from SYNTAX clause
// and mapped to IDL type
//     DateAndTime ::= OCTET STRING (SIZE (8 | 11))
typedef sequence<octet, 11> DateAndTimeType;

/*
DISPLAY-HINT: 2d-1d-1d,1d:1d:1d.1d,1a1d:1d
DESCRIPTION:
   A date-time specification.
   See RFC 1443 for detailed description.
   For example, Tuesday May 26, 1992 at
   1:30:15 PM EDT would be displayed as:
      1992-5-26,13:30:15.0,-4:0
   Note that if only local time is known, then
   timezone information (fields 8-10) is not present.
*/

/* pseudo */
interface TextualConventions {
```

```
    /*
    DISPLAY-HINT: 255a
    DESCRIPTION:
        Represents textual information taken from
        the NVT ASCII character set, as defined in
        pages 4, 10-11 of RFC 854. Any object defined
        using this syntax may not exceed 255
        characters in length.
    */
    string DisplayStringToString(in DisplayStringType value);
    DisplayStringType DisplayStringFromString(in string str);

    /*
    DISPLAY-HINT: 1x
    DESCRIPTION: Represents a media or physical-level address.
    */
    string PhysAddressToString(in PhysAddressType value);
    PhysAddressType PhysAddressFromString(in string str);

    /*
    DISPLAY-HINT: 1x
    DESCRIPTION: ....
    */
    string MacAddressToString(in MacAddressType value);
    MacAddressType MacAddressFromString(in string str);

    /*
    DESCRIPTION: Represents a boolean value.
    */
    string TruthValueToString(in TruthValueType value);
    TruthValueType TruthValueFromString(in string str);

    /*
    DESCRIPTION:
        The RowStatus textual convention is used to manage
        the creation and deletion of conceptual rows,
        and is used as the value of the SYNTAX /clause for
        the status column of a conceptual row.
        See RFC 1443 for detailed description.
    */
    string RowStatusToString(in RowStatusType value);
    RowStatusType RowStatusFromString(in string str);

    /*
    DISPLAY-HINT: 2d-1d-1d,1d:1d:1d.1d,1a1d:1d
    DESCRIPTION:
        A date-time specification. See RFC 1443 for detailed
        description. For example, Tuesday May 26, 1992 at
        1:30:15 PM EDT would be displayed as:
            1992-5-26,13:30:15.0,-4:0
        Note that if only local time is known, then timezone
        information (fields 8-10) is not present.
    */
    string DateAndTimeToString(in DateAndTimeType value);
    DateAndTimeType DateAndTimeFromString(in string str);

};  // End of TextualConvention interface
```

```
};  /* End of SNMPv2_TC Module.*/
#endif /* _SNMPv2_TC_idl */
```

Preliminary Specification

Part 8:
Object Model Comparison

The Open Group

Chapter 15

Introduction

15.1 Scope and Purpose

The publication by the Network Management Forum (NMF) of network-management agreements contained in OMNI*Point* (see reference **OMNI**) has focused industry attention on the issues of effective development of conformant network-management products. In developing the Open System Interconnection (OSI) management standards on which OMNI*Point* is based, the International Organization for Standardisation (ISO) and the International Telegraph and Telephone Consultative Committee (CCITT, now known as ITU-T) pioneered the specification of an object-oriented approach to modeling the resources that are to be managed.

Over the last few years the information technology industry has been rapidly developing object-oriented software development environments (languages, library tools, databases, etc.) for the realisation of object-oriented designs in computer systems. It is therefore natural for implementors of OSI management to expect to use these environments in the development of OSI-conformant network-management products. The fundamental enabling factor in realising this expectation is the compatibility of the underlying object models, so an evaluation of the OSI Management Model with those used in software development environments is required.

Such a comparison begs the question of which model the OSI Management Model should be compared against. In particular, the following have identified user needs for comparison:

- The Object Management Group (OMG), since it has forged an industry consensus in this respect, offers an important candidate.
- Another ISO / CCITT effort — Open Distributed Processing (ODP) — has wider scope than OMG, but provides a reference model and framework to assist comparisons of OMG and OSI management. There are expectations that most, if not all, of OMG will find its way into ODP related standards in the near future.

With regard to OMG, comparisons to the following have been developed:

ODP	in that OMG can be placed into the scope of the ODP reference model so that standardisation efforts involving OMG can be best directed.
Internet Management	in that OMG tools might be used to implement managers of embedded SNMP agents.

This **Object Model Comparison** report sets out to compare the three object models developed by:

- ISO / CCITT OSI management (SMI, CMIP)
- OMG CORBA for object-oriented software development (CORBA)
- Internet Management (SNMP).

Internet Management is included in the comparison, but the analysis in this report concentrates on comparing OSI management and OMG CORBA.

The definitions in this document make reference to concepts defined in the ISO/CCITT ODP reference model (see references **ODP93-2** and **ODP93-3**). This aids efforts to relate the three object models, using concepts from the ODP perspective.

The following comparison is, in itself, a necessary stage leading towards the larger objective to capitalise on the synergistic aspects of the three models, the synergistic aspects of the work that went into their respective development, and the maximum possible synergistic goals of the three groups. As will be shown, there is overwhelming agreement between OSI management and OMG models, but there are also a number of significant differences. Some are complementary and some are conflicting. Further JIDM documents are planned which will describe approaches for reconciling these differences.

The objective of this report is to provide a comparison of the three models by documenting the similarities, the complementary differences, and the conflicting differences;

Some knowledge of the OSI Management and OMG models is assumed. Readers are referred to the referenced documents **SMI**, **CMIP** and **OOM** for further information on the respective models.

15.2 Document Structure

The structure of this **Object Model Comparison** report is as follows:

- Chapter 16 provides a detailed comparison of the object models.
- Chapter 17 provides a summary of the similarities and differences.
- Chapter 18 briefly outlines the scope of work being undertaken to reconcile the models.
- Chapter 19 offers some concluding remarks.

Chapter 16

Comparison of Object Models

This chapter gives a detailed comparison of the various aspects of the OSI (SMI, CMIP), OMG (CORBA) and Internet (SNMP) management models, under the following headings:

- goals
- interfaces
- elements of objects
- object specification
- object taxonomy
- object reference
- object selection.

Each section begins with a glossary of basic concepts (incorporating relevant ODP terms). This is followed by a matrix comparison of the way the basic concepts are interpreted in the two models, and the section is then concluded with comparative analysis.

Trying to provide a glossary of generally-agreed concepts in object-oriented modeling is a difficult task, and the outcome is inevitably contentious. However, the works of many authorities on the subject have been consulted including Wegner (see reference **Weg90**), Meyer (see reference **Mey88**), Goldberg (see reference **Gold89**), Rumbaugh et al (see reference **rumb91**), Booch (see reference **Boo91**) and Kent (see reference **Kent91**). It is believed that the concepts defined here provide a reasonable basis on which to conduct the comparison.

16.1 Goals of the Models

Each model has been developed to meet specific business needs. These are summarized in the following sub-section.

16.1.1 Comparison

16.1.1.1 Intended Use

OSI Mgmt	Network Management.
OMG	Object-oriented distributed systems development.
Internet Mgmt	Management of the Internet and TCP / IP-based networked devices.

16.1.1.2 Interoperability/Portability

OSI Mgmt	At the syntactic and semantic level. No code portability, communications interoperability.
OMG	At the syntactic and semantic level. Code portability. CORBA 2.0 provides interoperability.
Internet Mgmt	At the syntactic and semantic level. No code portability. Communications interoperability.

16.1.1.3 User Advantage

OSI Mgmt	Management of heterogeneous network components.
OMG	Transparency to applications of underlying heterogeneous platforms.
Internet Mgmt	Management of heterogeneous inter-networked devices.

16.1.1.4 Re-usable Components

OSI Mgmt	Library /catalogue of management information.
OMG	Interface Type library.
Internet Mgmt	Management Information Base (MIB) specifications.

16.1.2 Analysis

Portability	Portability for OSI and Internet management is provided by the the XMP (see reference **XMP**) and XOM (see reference **XOM**) interfaces.
Interoperability	There are two aspects to component interoperability: agreement on the syntax and agreement on the semantics of information interchange. Achieving interoperability requires conformance to both. The strategic objective of each group is identical — to promote syntactic and semantic interoperability between heterogeneous computing components. The models differ only in their approaches to specifying such interoperability and their choices of the interface points at which the interoperability is to exist. For OSI Management and OMG, as can be seen in Section 16.2 on page 192, far from conflicting, these choices are complementary.

Re-usability

A common theme running through the statement of objectives of the OSI management and OMG models is that of component re-usability with emphasis on type/object libraries, portability of applications, and heterogeneous management components. Ever since the identification of the software crisis (see reference **Naur68**) some quarter of a century ago, the re-use of software components has been promoted as a solution (see reference **Krug92**).

16.2 Interfaces

As was pointed out in the previous section, the groups have chosen different positions for their interfaces — OSI Management has chosen a communications interface between computer systems, whereas OMG has chosen a programmatic interface between components within a computer system. These choices have dictated a different style of interface but, nonetheless, a number of important concepts apply to all three.

16.2.1 Concepts

Interface — The boundary at which a prescribed specification is supported. The reference model for ODP supports multiple object interfaces and viewpoints. An ODP Interface type is a predicate that characterises an interface. An interface signature can be described using an interface definition language. Interfaces can have behaviour specifications described using formal description techniques.

The three models compared in this document describe the following interface types:

- OSI — managed object communication interface, operations and notifications
- OMG — programmatic interface between invoker and object
- SNMP — *get* and *set* operations upon instrumentation variables.

Open Interface — An interface that supports a published protocol and at which conformance may be tested.

Protocol — A set of syntactic and semantic rules for exchanging information.

Carriage Protocol — A protocol to which invocations, notifications, and replies are added to form a complete protocol.

Interface Concurrency — The property of an interface such that it will accept protocol elements at any time.

Exception — A condition that precludes normal execution.

16.2.2 Comparison

16.2.2.1 Interface Type

OSI Mgmt — Communications interface between a managing-system and a managed-system; an agent, resident in the managed system, mediates between the managing system and a managed object.

OMG — Programmatic interface between an invoker and object. The interface includes signature only.

Internet Mgmt — Communications interface between a Network Management Station (manager) and Device (agent). The agent supports one or more MIBs.

16.2.2.2 Carriage Protocol

OSI Mgmt	The syntax and semantics of the carriage protocol are defined in ISO/IEC 9596-1 (see reference **CMIP**).
OMG	Independent of carriage protocol. Interoperability dictates one or more protocols.
Internet Mgmt	The syntax and semantics of the carriage protocol are defined in RFC 1157 (see reference **SNMPv2**).

16.2.2.3 Open Interface

OSI Mgmt	ISO/IEC 9596-1 is an international standard and many managed-object specifications are publicly available, for example, OMNI*Point* (see reference **(OMNI**).
OMG	IDL is specified in referenced document **CORBA**. Standard interfaces have been defined using IDL by object service specifications (see reference **COS**).
Internet Mgmt	RFC 1157 (SNMPv1) is an Internet standard; RFC 1448 (SNMPv2) is a proposed draft Internet standard. Many MIB specifications are publicly available for SNMPv1, with on-line access provided via the Internet.

16.2.2.4 Protocol Model

OSI Mgmt	The remote operations protocol model is one of non-blocking message-passing with normal and exception reply-types. A single initiating message may result in multiple replies.
OMG	The protocol model has two execution semantic styles: at most once (blocking with exceptions); best effort (non blocking, no guarantee of delivery). Delivery of events is not guaranteed by COSS event services (see reference **ESS**).
Internet Mgmt	The protocol model is primarily manager-initiated message-passing, without guaranteed delivery. A single initiating message results in (at most) a single reply.

16.2.2.5 Interface Concurrency

OSI Mgmt	A management association which supports operations and transfer of event reports requires interactions to be processed concurrently. Support for confirmed event reports requires interface concurrency.
OMG	Not precluded by the object model.
Internet Mgmt	Polling driven, manager controls concurrency. Traps are not confirmed.

16.2.3 Analysis

Interface Types

The choices of interface made by the OSI management and OMG groups reflects their different preoccupations: OSI chose network management communications, while OMG chose software development. These choices are synergistic: object-oriented network-management systems have to be implemented, and implementors will require object-oriented software tools.

Protocol Model

The use of OMG client stubs implies a blocking procedure-call model. However implementation support for a dynamic invocation interface is defined, which allows deferred synchronous operation. OMG implementations can be designed to employ callbacks to the original invoking client application for return of operation results, thus permitting asynchronous operations. However, some implementations of CORBA do not provide a callback location as part of receiving an operation invocation. Thus to be safe, a CORBA operation which is designed to be followed by callback operations should provide an input parameter containing an ObjectReference upon which to invoke the callback.

OSI Management has, through the use of ROSE (see reference **XAP-ROSE**), chosen a non-blocking message-passing model. Exception handling mechanisms will have to be aligned.

Protocol Specification

There are three aspects to the specification of a protocol: the rules for exchanging information; the semantics of the information; and the syntax of the information. OSI Management has chosen to specify all three aspects, whilst OMG in CORBA 1 has chosen to specify no protocol for interoperability. However CORBA 1 does specify a language mapping to C. Other language mappings are defined by the OMG (including C++, SmallTalk).

CORBA 2.0 provides interoperability between different ORB implementations, and protocols have been specified for use between ORBs.

16.3 Characteristics of Objects

This section deals with the most fundamental aspects of the object models — the objects themselves, what they are, and what they are about. To compare the characteristics of objects, notwithstanding the notion of encapsulation, it is necessary first to consider the abstractions underlying the internal nature of objects including their state, behaviour, and methods. From there, the attributes appearing at the interface of objects and the signals crossing the interface can be compared.

In ODP terms, all objects have a *type*, which is a predicate that determines if an object is of that type. The extension of all the object instances of a given type is called the *class* derived by that *type*. Instances of a class are generated to adhere to the specifications of a *class template*, which specifies the *behaviour* of the object. Objects are characterised by the interfaces they support. In ODP, an object can support multiple interfaces, bound to a common state.

16.3.1 Concepts

Object	A collection of behaviours that share the same state.
State	A stable internal condition of an object.
Behaviour	A change in the state of an object.
Attribute	A set of data values (that is, a data-type) and qualifiers over those data values; the data values are built from literals.
Literal	A set of data values (that is, a data-type) which, by convention, have fixed semantics (for example, integer, boolean).
Encapsulation	A property of an object such that its state can only be accessed by means of prescribed signals across the object boundary or by means of prescribed relationships between objects.
Operation	A behaviour of an object. It is normally initiated by the reception of an invocation.
Event	An autonomous behaviour of an object. It may be signalled by the emission of a notification.
Invocation	A signal, possibly parameterised, requesting that an operation be performed by an object.
Notification	A signal, possibly parameterised, indicating that an event has occurred.
Reply	A signal, possibly parameterised, indicating a response to an invocation or notification; replies are often classified as normal or exceptions.
Parameter	A data value whose syntax conforms to that of a prescribed protocol.
Method	The implementation of an operation.
Qualifier	An additional semantic applied to an attribute (for example, concerning accessibility or default values).

16.3.2 Comparison

16.3.2.1 Description

OSI Mgmt

A *managed object* is described by the Management Framework (see reference **ISO/IEC7498-4**) as *''a view of a resource that may be managed through the use of OSI Management protocols''*. A managed object enforces encapsulation, and it is characterised by the attributes it makes available, its behaviour, and the invocations and notifications that cross its boundary. A managed object can optionally present itself as if it were a superclass (*allomorphism*) of that of which it is instantiated.

OMG

An *object* is described as *''a package of data and code used to implement a computational construct or to model an application entity''* (see reference **OOM**). An OMG object enforces encapsulation and is characterised by the attributes it makes available, its behaviour, and the signals that cross its boundary. An object may have more than one interface, but only inheritance is explicitly provided as a mechanism in IDL to specify multiple interfaces.

Internet Mgmt

As described by RFC1155 (see reference **ISMI**), *''managed objects are accessed via a virtual information store, termed MIB ... Each type of object has a name, a syntax, and an encoding''* (see reference **ISMIV2**). Internet objects represent individual variables which can be manipulated using SNMP protocol; the Internet SMI provides no mechanism for collecting together all variables which together might encapsulate a resource. Also, in Internet objects, the notion of interface inheritance is not supported.

16.3.2.2 Object Operations

OSI Mgmt

An operation may be initiated by receiving an invocation. A confirmed operation will generate one or more replies; a non-confirmed operation will not. Exceptions are signalled by means of exception replies. Invocations and replies may be parameterised.

OMG

An operation is initiated either by a generated stub or by the dynamic invocation interface. The normal execution of an operation results in a reply (and the return of control to the invoker). An exception is indicated by means of an exception signal and the transfer of control to an exception handler. Invocations, replies, and exception signals may be parameterised.

Internet Mgmt

An operation is initiated by the receiving of a request. Each request generates a single response (although SNMPv2 **Get-Bulk** provides a *self-repeating* operator for conceptual table traversal). Exceptions are signalled as responses. Requests and responses may be parameterised.

16.3.2.3 Object Events

OSI Mgmt

Behaviour unconnected with operations may be signalled by means of a notification. A confirmed notification requires a reply; a non-confirmed notification does not. Notifications and replies may be parameterised

OMG

Objects which generate events invoke an operation on the event notification service (see reference **ESS**). Event distribution is via push model or pull model. Event notification service does not indicate

failure/success of delivery to Invoker. No standard event filtering is yet defined (other than by grouping originating objects into channels). The receiver of typed events indicates support of each event.

Internet Mgmt — Significant events unconnected with operations may be signalled by means of a non-confirmed *Trap* notification (intended for infrequent generation by managed devices) or a confirmed *Inform-Request* (intended for use between managers). Notifications are specified as part of the carriage protocol, and are unrelated to object type definitions. Event philosophy has been characterised as *trap-directed polling*.

16.3.2.4 Behaviour

OSI Mgmt — Behaviour can be in any language (natural or formal), but is most commonly written in English. This is normative information and can be subjected to conformance testing. This would normally require an intermediate step to represent the behaviour in a testable form.

OMG — Side effects are permitted but are not specified in IDL. Basic OMG IDL has no formal way to specify what event types may be generated (event generation is independent of interfaces an object supports). A Receiver only knows which channel an event came from (channels can have multiple object sources) but event data could be defined to include originating *objectReference*.

Internet Mgmt — Textual specification may be included in the description clause of an object type definition, describing the behaviour of an individual variable. This may include description of an entire conceptual table or table row, but otherwise does not specify the behaviour of encapsulated resources. Notification behaviour specifications are unrelated to object type behaviour specifications.

16.3.2.5 Attributes

OSI Mgmt — An attribute is characterised by a set of data values and qualifiers which specify matching rules, range restrictions, initial and default values, and access restrictions. Attributes can also be manipulated by specifying an attribute group name that identifies one or more attributes.

OMG — An attribute is characterised by an identifier, a type, and an optional *read-only* qualifier. There is no mechanism for grouping attributes.

Internet Mgmt — An attribute is embodied by an object type specification, which identifies the object name, syntax, description, access, status, and default value (if any). Syntax may include range and/or value restrictions. There is no mechanism for grouping attributes.

16.3.2.6 Attribute Operations

OSI Mgmt — Built-in operations on attributes are **get attribute**, **replace attribute** and **replace attribute with default**. Set-valued attributes have **add-member** and **remove-member**. Built-in operations support manipulation of multiple attributes or attribute groups within a single operation. A built-in operation supports the cancellation of the **get** built-in operation. Arbitrary exceptions (specific errors) may be specified on attribute operations.

OMG — Built-in operations on attributes are **set attribute** and **get attribute**. Built-in operations can only get/set the value of a single attribute within one invocation. Only standard exceptions are supported on attribute operations.

Internet Mgmt — Built-in operations on attributes (object types) are **get attribute** and **set attribute**. Aggregate conceptual tables and table rows are supported by the ASN.1 construct `SEQUENCE [OF]`. Built-in operations are **get-next** (SNMPv1) and **get-bulk** (SNMPv2). However, these operations serve only to navigate through conceptual tables; individual object types are the only accessible elements. Built-in operations support manipulation of multiple attributes within a single operation. Some standard exceptions are supported on attribute operations.

16.3.2.7 Object Life-cycle Operations

OSI Mgmt — Built-in operations to agent for create and delete of object instances. Creator may specify name or may rely on agent to supply name. It has no standard protocol to move object instances between systems.

OMG — No built-in operation for instantiation of object interfaces by client demand. However, factory objects may be defined which allow clients to create application specific object interfaces. COSS (see reference **ESS**) life-cycle services specify deletion of objects, as well as copy/move of object implementations.

Internet Mgmt — No direct support for life cycle operations. SNMP v2 has notion of special variable in table rows which allows for a **set** operation to make a new table row available.

16.3.2.8 Attribute Behaviour

OSI Mgmt — The behaviour of attribute-oriented operations is defined. The interaction between attribute operations is left to the package definition.

OMG — **get** and **set** operations are defined for attributes. Behaviour specification is only available through textual comments.

Internet Mgmt — The behaviour of attribute-oriented operations is defined; interaction between operations is not.

16.3.2.9 Data Types

OSI Mgmt — All primitive ASN.1 built-in types are supported, including `BOOLEAN; INTEGER; BIT STRING; OCTET STRING; NULL; ANY DEFINED BY, CharacterString; OBJECT IDENTIFIER; REAL; ENUMERATED;` and `Generalized Time`. The following constructed types are supported: `SET, SET OF, SEQUENCE, SEQUENCE OF, CHOICE`.

OMG — Primitive types are integer, floating point, character, octet, boolean, and ANY (type descriptor, data fields). Built in access methods exist for ANY type. Aggregate attribute values and operation parameters are supported by the collection types, which are Sequences (variable length ordered list); struct, Union, Enum, Array (fixed size). In IDL, recursion within constructed Type declarations is only allowed if there is an intervening Sequence.

Internet Mgmt — The following primitive ASN.1 built-in types are supported by SNMPv1: `INTEGER, OCTET STRING, NULL, OBJECT IDENTIFIER`. The only ASN.1 constructed type permitted is `SEQUENCE [OF]`, used to generate either lists or tables. New application-wide types may be defined, but only if they resolve into one of the above implicitly defined ASN.1 types. RFC1155 (see reference **ISMI**) defines the following application-wide types: NetworkAddress, IpAddress, Counter, Gauge, TimeTicks, Opaque. RFC1442 (see reference **ISMIV2**) specifies revisions for SNMPv2, including addition of `BIT STRING`, `unsigned INTEGER`, and 64-bit Counters.

16.3.2.10 Encapsulation

OSI Mgmt — Supported.

OMG — Supported.

Internet Mgmt — Not supported (except where encapsulating any individual variable).

16.3.2.11 Object Reference Data Type

OSI Mgmt — ObjectInstance (Distinguished Name), used extensively as communication interface parameter. X.500 syntax used.

OMG — Object reference (type name Object). Interface operations may employ the object reference type for their parameters. Its syntax is opaque to the application using it.

Internet Mgmt — Object type identifier, appended to an object instance identifier (row index) for objects contained within a conceptual table.

16.3.2.12 Interface Type References

OSI Mgmt — Managed object class, attribute id, notification id and action id are all specified with a globally unique OBJECT IDENTIFIER value, as well as a document specific textual ASN.1 reference label. Textual reference labels are case sensitive.

OMG — Interface type has global reference id through Module Names. Attributes, types and operations are referred to using scoped names defined within the interface specification. IDL identifiers are case insensitive.

Internet Model — Object type is specified with globally unique OBJECT IDENTIFIER value, as well as document-specific textual ASN.1 reference label. SNMPv2 also specifies globally unique OBJECT IDENTIFIER values for Notifications, Module Identities, Groups, and Compliance Modules.

16.3.3 Analysis

Description — At first sight, the descriptions of objects in the OSI management and OMG models appear different, the OSI Managed object being a ... *view of a resource* ... and the OMG object being ... *a package of code and data* .... In fact, both object models enforce strict encapsulation: objects are characterised by how they appear to the outside world, not how they are constructed internally. With the exception of notifications and the support of built-in operations to support manipulating multiple attributes with a single invocation, differences between the models are surprisingly few. In

practice, objects from both models will be constructed of code and data and be run on computer processors.

From the ODP perspective managers and agents can be modelled as ODP objects. However, OSI management, OMG and Internet management each limit specification of an object to that of a performer of operations. Invoker behaviour is not specified in OSI management, OMG, or Internet management.

Operations

Both the OSI and OMG models support the notion of an extensible set of operation types, although particular objects are only required to support a fixed set of operation types. The mechanism for invoking such operations is defined by the underlying protocol.

There is a need to define a generic OMG factory object interface to support the OSI managed object **M-CREATE** operation for managed objects mapped to CORBA interfaces.

Events

The major difference between the OMG and OSI management models is the way of specifying autonomous events.

In OSI, management events are behaviour unconnected with operations; they reflect the active, autonomous nature of managed objects. Typically, event notifications are caused by alarm conditions (for example, loss of signal) in the resource represented by a managed object. The event notifications which an object may emit are specified as part of the Managed Object definition. An event report management function determines which event notifications are forwarded as event reports to designated destinations.

The notion of events has been accommodated in the OMG object services. The main difference between OMG events is that they are specified as being supported by the receiver of the events. In effect, OMG event services models events as *operations in the other direction*.

Attributes

In OSI management and OMG (unlike Internet management), the models differentiate between attributes and objects:

- there are distinct type systems for objects and attributes
- objects ontologically precede attributes (that is, attributes can only be accessed within the context of an objects)
- there are built-in operations to manipulate attributes.

This is in-line with popular programming models such as Eiffel (see reference **Mey88**) and C++ (see reference **Strou87**), but is at variance with models such as Smalltalk (see reference **Gol89**), where everything is an object. Both the OMG and OSI Models support aggregate attribute types.

OSI Management allows specification of specific user-defined errors on attribute operations, and defines operations to manipulate members of set-valued attributes (*add, remove*). If OSI management attributes are mapped to OMG attributes, then only CORBA standard errors can be used, resulting in a loss of information. In addition, there would be no way of using CORBA operations to add or remove members of set-valued attributes. An alternative would be to map some OSI managed object attributes to custom CORBA IDL operations, in cases where user defined

	exceptions or set member operations are required.
Encapsulation	Both the OSI and OMG models have similar notions of encapsulations: the state of an object can only be accessed by means of signals across the object boundary or by means of relationships between objects. However, the OSI management interface definition includes, in addition to operations the object can perform, the notifications that an object may emit.

16.4 Object Specification and Instantiation

16.4.1 Concepts

Specification Binding	The irrevocable merging of specifications.
Object Instantiation	The process by which instances of objects are created.

16.4.2 Comparison

16.4.2.1 Attribute Specification

OSI Mgmt	Attributes and qualifiers are specified independently of managed-object classes.
OMG	Attributes and qualifiers are defined only within scope of Object interface types.
Internet Mgmt	Attributes (object types) are specified individually, not within any *object class* context. Object types may be aggregated into conceptual tables, but the same object type may not appear in multiple tables. SNMPv2 Textual Conventions may be used to specify behaviour and syntax to be reused by multiple object types.

16.4.2.2 Binding

OSI Mgmt	Specifications of attributes, operations, and events may be bound into packages by means of a package specification. The package specification may specify additional constraints over the attributes (for example, for example, range restrictions) and over operations and events (for example, for example, constraining variable-type parameters); it also specifies the semantics of the package as a whole (for example, the meaning of a statistics package). IDD Package specifications themselves may be bound into a managed-object class by means of a managed-object-class specification. These bindings are qualified as either mandatory (a managed-object instance must reflect the specifications contained in the package) or conditional (a managed-object instance may reflect the specifications contained in the package).
OMG	Specifications of attributes and operations are combined into interfaces. Interfaces may be further combined into modules. Optionality is not supported for interface specifications. However, attribute and parameter values can include *null* choices, and there is a standard NOT_IMPLEMENTED exception type defined for CORBA. The OMG Event Service has events defined as operations of an interface which operates in the reverse direction (that is, events are raised by the Object invoking an operation). Thus, a managed object can define one or more interface containing all the events which it can generate. The manager needs to support that interface.
Internet Mgmt	Individual object types may be combined into conceptual table rows, but no additional constraints or semantics are imposed there. In SNMPv2, individual object types may be combined into *Object Groups* which specify the semantics of the group as a whole, but do not otherwise impose

additional constraints. Also in SNMPv2, individual object groups and types may be referenced by Module Compliance macros which specify access and/or value constraints permitted for conformance to a given MIB module. Object type status does not indicate presence/absence, its values are **current** and **obsolete**. A *Module Compliance* macro specifies object groups as mandatory or otherwise for the purpose of conformance.

16.4.2.3 Object Instantiation

OSI Mgmt — Instantiation may be from what alreadty exists, by local means, or by means of a CMIP create operation.

OMG — Instantiation may be from what already exists, or by means of an operation invocation on an application specific *factory* interface. No standard factory interfaces are defined for CORBA.

Internet Mgmt — Instantiation may be from what already exists, by local means, or by means of a set operation involving a conceptual table row status variable.

16.4.2.4 Behaviour Specification

OSI Mgmt — Unstructured natural language behaviour clause, with a recommendation to state pre-conditions, post-conditions, and invariants.

OMG — Unstructured natural language, using IDL comments.

Internet Mgmt — Unstructured natural language.

16.4.2.5 Specification Tools

OSI Mgmt — GDMO templates.

OMG — IDL.

Internet Mgmt — ASN.1 macros (`OBJECT-TYPE, NOTIFICATION, MODULE-IDENTITY, MODULE-COMPLIANCE, OBJECT-GROUP`).

16.4.3 Analysis

Specification techniques

The major difference in specification techniques is between the OSI Management piecemeal approach and the more monolithic approach taken by OMG. The OSI Management approach of separate specifications bound into packages which are subsequently bound into object classes, is claimed to improve the potential for re-use of specifications. Conditionally-bound packages are a form of late binding. They meet OSI Management's need for controlled variation in managed-object class specifications, but if used liberally they can cause ambiguity for implementors (see reference **MPR**). In the OSI Management model, the union of the object-class identification with the identification of the packages present serves as the indicator of a particular managed object's characteristics. For implementors, packages complicate the type hierarchy.

The OMG specification technique is more monolithic over the type and does not admit late binding (it specifies that ''*the object type is an absolute indication of the characteristics of a type instance*''). Re-use is achieved by inheritance (possibly multiple) of interface definitions. However, there is

a standard exception (NOT IMPLEMENTED) which must be dealt with by invokers when an object implementation does not support an operation defined in its interface.

Conditional packages can lead to situations where all members of a class may not be the same. Usage of null values for attributes/parameters to simulate optionality does not change this.

Behavioural Specification

The behavioural aspects of managed objects are specified using natural language (see reference **SMI**), although it is recommended that such specifications should be expressed in terms of pre-conditions, post-conditions, and invariants. There have been attempts to use formal techniques for the specification of the behaviour particular managed-object classes (see reference **SMI**). The OMG also proposes the use of natural language to specify the behavioral aspects of objects.

ODP behaviour specifications will use Formal Description Techniques (FDTs). Dynamic type checking may use assertions about behaviour.

Specification Tools

OSI Management provides two tools for formally specifying the syntax of managed objects and the associated messages:

- a template language (see reference **GDMO**) for defining the characteristics managed objects
- a data-definition language (see reference **ASN.1**) for defining the data types associated with attributes and parameters.

The OMG CORBA 1.0 model does not prescribe a language for defining the syntax of the associated messages. For IDL (see reference **CORBA**), a language similar to C++ is used to define the programmatic interface signature.

The OMG model defines objects by defining interface operations using in IDL. The word *object* in CORBA is equivalent to *object interface* in ODP.

16.5 Object Taxonomy

A taxonomy permits reasoning about real-world things by grouping instances of those things into classes or types and by relating the different classes or types. In day-to-day discourse, the terms class and type are used synonymously, but in the world of object-oriented systems, classes and types are different. In ODP, a *class* is the extension of (that is, the set of instances which satisfy) a *type*.

16.5.1 Concepts

Object Type	A predicate over the characteristics of an object. Objects are said to *conform to a type*.
Object Subtype	An additional predicate over the characteristics of a supertype.
Object Class	The set of objects which are of the same type. Objects in a class can have one or more compatible templates used for the generation and management of objects. Every type defines a class; however, the converse is not necessarily true. In ODP, the term *class* is used to represent the set of instances that satisfy a type, and a *template* represents the object specifications.
Object Sub-class	A modification to the template of a superclass.
Inheritance	A mechanism for deriving the specification of a descendant from the specification of an ancestor, such that in addition to its own characteristics, a descendant acquires all (strict inheritance), or part, of the characteristics of its ancestor. A descendant may have a single immediate ancestor (single inheritance) or multiple immediate ancestors (multiple inheritance). The super/sub relationship resulting from single inheritance may be represented by a tree; that resulting from multiple inheritance may be represented by a directed, acyclic graph.

At first glance, the subtyping and subclassing relationships in the OMG and OSI models appear similar. They both involve the development of new types or classes by means of inheritance and by the addition of new predicates or template modifications. The difference is that the subtyping relationship is one of substitutability, whereas the subclassing relationship is one of implementation sharing (see reference **Lalon91**). Thus an instance of a subtype may be substituted for an instance of a supertype without any change in behaviour visible at the object interface. A subclass, by comparison, shares the implementation of its superclass, and an instance of a subclass cannot always be reliably substituted for an instance of a superclass.

16.5.2 Comparison

16.5.2.1 Object Class

OSI Mgmt	The Management Information Model (see reference **SMI**) classifies managed objects into classes. Class specifications are documented by templates.
OMG	The OMG model specifies objects by their interface. There is no direct concept of Object Class in OMG. However an object template can be derived from the inheritance graph of interface definitions.
Internet Mgmt	The Internet SMI (see references **ISMI** and **ISMIV2**) defines object types which classify individual variables.

16.5.2.2 Taxonomy

OSI Mgmt	Managed-object classes are related in quasi type-hierarchy by means of strict, multiple inheritance. The root of the hierarchy is the managed-object class *TOP*, which specifies attributes for specifying static properties common to all classes (for example, the actual Managed object class, which of the conditional packages are instantiated, and which classes the instance may respond to requests as). It is legal to inherit from two class definitions which include the same action or attribute, since the properties of the common named operations are merged (for example, **get** and **get-replace** become **get-replace** in the derived class).
OMG	Object types are related in a type hierarchy of supported interfaces by means of multiple inheritance. The root of the hierarchy is the object type *object*. It is currently illegal to inherit from two interfaces with the same operation or attribute name, or to redefine an operation or attribute name in the derived interface.
Internet Mgmt	Object types are related in a naming hierarchy that reflects conceptual table and object group organisation. The root of the naming hierarchy is the OBJECT IDENTIFIER assigned to the entire MIB. There is no support for inheritance of any type, although SNMPv2 conceptual tables may augment existing conceptual tables specified elsewhere.

16.5.2.3 Type System

OSI Mgmt	The managed-object types are distinct from other types (for example, attributes, actions, notifications) in the model.
OMG	Object interface types are integrated into a single type-hierarchy for all the types in the model.
Internet Mgmt	Object types are distinct from each other, and from other types in the model.

16.5.3 Analysis

The taxonomies used in the OSI and OMG models are largely similar. They are both specified in terms of object interfaces, and both use a multiple-inheritance discipline. The OSI Management Model specifies a number of attributes in TOP (*objectClass* and *nameBinding*). Thus all managed objects contain instances of those attributes.

The OMG model support the notion of substitutability by means of subtyping (that is, support for each inherited interface is required when supporting a derived interface). The OSI Management Model supports a similar notion by means of a limited subtyping capability termed compatibility (see reference **SMI**).

Representing GDMO managed object inheritance in the case of repeated attributes or actions in the inherited packages will require special treatment when translating to OMG CORBA IDL.

16.6 Object Reference

Understanding the term *object reference* requires special care. Confusion and overlap exists between the identity of an object and its name. Informally, the identity of an object is that which distinguishes it from all others (see reference **Khosh86**), and a name is a mnemonic handle used to refer to an object (see reference **Hauze86**). Thus, the identity of a person could be the molecules that make up that person (although the molecules keep changing) and the name of a person could be the symbols that appear on their birth certificate. Identity (that is, object reference) is difficult to define in formal sense, but intuitively it is *''something enduring about an object''* (see reference **Kent91**). A name on the other hand is easier to define, being a token used to refer to an object. However, names of objects can change with time, and the same name could apply to many objects over time.

Both *object references* and *names* refer to objects, but generally *object references* have no structure and are implemented as long bit-strings to facilitate effective machine processing. Their main use is to *''resolve predicates of sameness and equality''* (see reference **Kent91**). *Names*, on the other hand, are normally symbolic and mnemonic and oriented towards human consumption.

16.6.1 Concepts

It is important to have a formal framework for object reference and object naming within a distributed system. The following definitions are offered:

Object Identity — A unique property of an object that unambiguously distinguishes it from all other objects in the universe of discourse for all time.

Object Reference — A unique token used to unambiguously refer to an object during its lifetime. Object references are assigned at object creation time and are never subsequently re-assigned. Thus, the mapping function of object reference to object, if it is valid, is both single-valued and singular (1:1) during the life-time of the object, and invalid afterwards.

Name — A token used to refer to an object. Different objects may be assigned the same name and the object may be assigned different names. Names may be re-assigned during the lifetime of the object. Thus the mapping function of name to object, if it is valid, may be both multi-valued and non-singular, and may change with time.

Access Transparency — A reference to an object does not expose/define the carriage protocol to that object.

Location Transparency — A reference to an object does not expose its location.

Location Independence — A reference to an object remains valid after the object changes location.

16.6.2 Comparison

16.6.2.1 Object Reference

OSI Mgmt — There is no specific concept of an object reference for a managed object other than the object instance (distinguished name).

OMG — In CORBA, Object Reference is an opaque type (that is, it is not externally defined and thus has no particular format). Objects may have more than one reference. Objects are assigned object references at creation. Object references are local entities and may not be globally unique.

Internet Mgmt — An object's reference is its name (that is, ASN.1 OBJECT IDENTIFIER). All agents use identical names for objects of a given types. In SNMP V2, the community field (or whatever it is now called) can be used to select the particular Agent which will serve as the scope for the object references passed in the protocol operations.

16.6.2.2 Name

OSI Mgmt — A managed object must have a single distinguished name, comprised of a sequence of Relative Distinguished Names (RDNs), corresponding to a name scope hierarchy of instance containment (not necessarily physical containment). Each RDN is a single, mandatory attribute/value pair associated with the managed object. The RDN attribute value is assigned at object creation-time and may not be changed during the lifetime of the object, although it may be reused afterwards.

OMG — Objects may have an arbitrary number of names. The scope is a name context. Names are distinct from objects — objects need not have names. There is mapping between name values and objects. The mapping may be changed at any time.

Internet Mgmt — An object instance name is formed by concatenating the OBJECT IDENTIFIER assigned to the object type, with either the value zero (for singular objects), or one or more object values representing conceptual table row indices. Index object types do not need to be part of the conceptual table row. Any given conceptual row always has the same index value(s), although that row may be conceptually created and deleted many times over.

16.6.2.3 Naming Model

OSI Mgmt — A hierarchically-qualified set of naming contexts, based on object containment, is prescribed to ensure object names are unique and unambiguous within an agent system's local context (local distinguished name), as well as in a global context (distinguished name). Aliases are not permitted.

OMG — The Name Service provides a directed, potentially cyclic, graph of naming contexts. This may have many roots. All names are mapped to an *ObjectRef* (possibly of another naming context). Thus an object may have many names and a single point in the graph may be reached via many routes.

Internet Mgmt — A hierarchically-qualified set of naming contexts, based on the object type registration tree, provides names that are locally unique and unambiguous within a single agent. To provide global uniqueness, object instance names must be used in conjunction with the agent's network address. Aliases are not permitted.

16.6.2.4 Access Transparency

OSI Mgmt	Unsupported. CMIP as defined in ISO / IEC 9596-2 (see reference **CMIP**) is required. CMIP may run over Internet Protocol (IP), LLC I, or OSI lower layers.
OMG	Supported. ORBs may use whatever protocol is most appropriate. ORBs may employ Internet Protocol (IP), or OSI lower layers. CORBA interoperability requires support of a standard protocol.
Internet Mgmt	Unsupported. SNMPv1 or SNMPv2 as defined by referenced documents **SNMP** and **SNMP** is required. SNMP may run over a wide variety of Internet and non-Internet transports, as described by RFC1449.

16.6.2.5 Location Transparency

OSI Mgmt	Not supported when objects are named within the scope of the System object. For OSI management, OSI protocol machines are naturally named within the scope of the open system they are part of, and thus location transparency is meaningless. However, not all managed objects have to be named within the scope of the System managed object, so location transparency is not precluded.
OMG	Supported in CORBA. Lifecycle services include **Move** operations.
Internet Mgmt	Not supported.

16.6.2.6 Location Independence

OSI Mgmt	Not supported for managed objects named within the scope of *System*.
OMG	Supported in CORBA.
Internet Mgmt	Not supported.

16.6.3 Analysis

At first sight, the object reference schemes seem very different. In the OMG model, identity is determined by an opaque object reference value, while in the OSI Management model it is determined by name. However, if the user were to forbid the renaming or moving of managed objects (which will largely be the case in practice) and were to carefully manage the re-use of managed-object names, the two mechanisms are broadly equivalent, since managed-object names are unique and unambiguous, and aliases are not permitted.

16.7 Object Selection and Address Resolution

16.7.1 Concepts

Selection	The mechanism by which object instances are selected to receive invocations.
Associative Selection	A reference constructed from predicates over the characteristics of objects. Such a reference can resolve to zero or more objects.
Address Resolution	The mechanism by which a name is mapped to an address (see reference **Jacqm90**).

16.7.2 Comparison

16.7.2.1 Direct Selection

OSI Mgmt	By name.
OMG	In OMG, an Object Reference is an address, that is, it is all that is necessary to connect to that object, regardless of its current location. ORB interoperability is provided in CORBA 2.0 (see reference **CORBA**), which includes resolving object references between different ORB implementations.
Internet Mgmt	By name.

16.7.2.2 Associative Selection

OSI Mgmt	Support of associative references by means of filters is optionally supported. The syntax and semantics of filters are defined. The scope is the whole, or some sub-tree, of the tree of managed objects. With CMIP, the invocation is made directly on the scope and filter — the agent implementation sorts out the targets.
OMG	Associative reference to a set of names could be done by traversal of the naming graph, or by a Trader Service. A standard OMG mechanism is not yet defined. Commands addressed to a set of objects via an associative reference would require a two phase mechanism: one which resolves to a set of objects via constraint matching on properties of the object's supported services; and the other which iterates over the objects invoking the operation on it. Clearly a scoping and filtering service could be written, but none appears specifically in the current OMG Roadmap (at the time of publication). OMG has a Relationship Service, which may also provide a different mechanism. The OMG Object Services activity might *fast-track* a Trader Service (such as ODP trader) which provides associative references employing constraint matching on properties of services supported by object instances. Support for such associative references is permitted but not required. A standard OMG trader has not yet been specified, and it is expected that the ODP trader can be used.
Internet Mgmt	SNMP **get-next** and **get-bulk** operators allow reference to a conceptual table entry which is lexicographically *after* a named entry. The scope is the agent's network address, the entire registration tree, and

lexicographic ordering within that tree. However, the receiver of these operation responses must do any necessary *filtering out* of unwanted variables. RFC1445 (see reference **IADM**) defines *MIB Views* which are subsets of all instances of all object types.

16.7.2.3 Address Resolution

OSI Mgmt — The model does not prescribe a mechanism for address resolution, but it could be provided by the Directory (see reference **DIR**).

OMG — The ORB provides address resolution. Mapping from a name to an object reference is done by the Name Service. An ORB resolves only a single target, and it is an application concern to sort out the targets.

Internet Mgmt — The model does not prescribe a mechanism for address resolution, but it could be provided by any directory service, including the Internet Domain Name Server.

16.7.3 Analysis

Both OSI and OMG models permit similar features with respect to object selection. The OSI model permits a direct selection of a single object by name, and the OMG model permits it by object identifier. Both provide associative selection over some subset of objects that may resolve to zero or more objects. In both cases, the responsibility for address resolution is delegated.

The OSI Management model defines specific syntax and semantics of the supporting mechanism. The OMG model does not.

ODP is specifying the syntax and semantics of a Trader service, which is expected to be used for associated selection of OMG objects.

Chapter 17

Summary of Similarities and Differences

This chapter gives a summary and analysis of the similarities and differences between the OSI, OMG and Internet Management Models.

17.1 Summary

17.1.1 Interoperability and Portability

OSI Mgmt	CMIP allows different implementations to interoperate through specification of communications interface. However application portability is not provided.
OMG	Application portability is provided by specification of programmatic interface. CORBA 2.0 provides a common protocol for different implementations to interoperate.
Internet Mgmt	SNMP allows different implementations to interoperate through specification of communications interface. However application portability is not provided.

17.1.2 Re-usable Components

OSI Mgmt	Library /catalogue of management information.
OMG	Type library (interface repository).
Internet Mgmt	MIBs containing object type definitions.

17.1.3 Encapsulation

OSI Mgmt	Supported.
OMG	Supported.
Internet Mgmt	Not supported.

17.1.4 Object Operations

OSI Mgmt	Supported.
OMG	Supported. No built-in create service provided.
Internet Mgmt	Not supported.

17.1.5 Behaviour

OSI Mgmt	Supported.
OMG	Supported.
I0ernet Mgmt	Supported.

17.1.6 Attributes and Attribute Operations

OSI Mgmt	Supported. Specific exceptions can be specified.
OMG	Supported. Specific exceptions cannot be specified.
Internet Mgmt	Supported.

17.1.7 Taxonomy

OSI Mgmt	Managed object class type-hierarchy/graph supported by multiple inheritance. Attributes, actions, and notifications may be repeated in derived classes, with additional properties.
OMG	Interface Type hierarchy/graph supported by multiple inheritance. Attributes and operations cannot be repeated in a derived interface specification.
Internet Mgmt	No inheritance supported.

17.1.8 Direct Selection

OSI Mgmt	By global or local distinguished name.
OMG	By object reference. The form of object references is ORB-specific.
Internet Mgmt	By object name (OID + index).

17.1.9 Intended Use

OSI Mgmt	Distributed network management.
OMG	Object-oriented software development.
Internet Mgmt	Management of inter-networked devices.

17.1.10 Interface Type

OSI Mgmt	Communications.
OMG	Programmatic.
Internet Mgmt	Communications.

17.1.11 Interface Concurrency

OSI Mgmt	Support required for confirmed events and operations/notifications on the same association.
OMG	Not required, but concurrency is implicit in the model, especially where deferred synchronous calls are made (as in the DII).
Internet Mgmt	Not precluded.

17.1.12 Protocol Model

OSI Mgmt	Remote operations (invoke, returnResult, returnError, and reject messages).
OMG	Procedure call. CORBA2 defines protocol(s) for interoperability.
Internet Mgmt	Message passing, requests have agent responses, traps are sent by agent and are unconfirmed.

17.1.13 Multiple Replies

OSI Mgmt	Supported explicitly by CMIP.
OMG	Although not specified explicitly in CORBA, multiple replies can be achieved by passing a reference on the call and implementing a call-back style of programming, by defining pairs or sets of definitions going in both directions. The service may invoke many calls on the reference to pass results and the return on the original request.
Internet Mgmt	Not supported (although the SNMPv2 **get-bulk** operator provides a self-repeating request which provokes several responses).

17.1.14 Object Events

OSI Mgmt	Supported by CMIP. Notifications are specified as part of managed object definition.
OMG	There is now a standard models of events — the OMG Event Service. Notifications are not specified as part of the sender's interface definition. The event receiver registers the event receipt interface. Typed events are specified as part of a receiver's interface definition. Fine filtering of events is not supported.
Internet Mgmt	Provides limited event support in the form of Traps and Inform-Requests. The philosophy is typically characterised as *trap-directed polling*.

17.1.15 Late Binding

OSI Mgmt	Some degree of optionality is handled in GDMO via conditional binding of a package to a managed object instance of a class defined with conditional packages.
OMG	In OMG, the implementation of a particular object instance is determined when the instance is created, by the implementation repository. It is even possible to change implementations, although the semantics are currently unclear. Attribute and parameter values may be defined with discriminated choice types including a NULL alternative.

	IDL has no direct equivalent of managed object class. However, simply supporting an interface does not actually require an implementation of all operations. An object is perfectly at liberty to implement a function simply by raising an exception. Clearly this is not covered in the specification part.
Internet Mgmt	Not formally supported, except through compliance specifications.

17.1.16 Associated Selection

OSI Mgmt	Supported through scoping and filtering on the managed object instance tree.
OMG	Permitted. Trader services will enable this in the future.
Internet Mgmt	Very limited support.

17.1.17 Associated Selection Scope

OSI Mgmt	Sub-trees of the tree of managed object instances.
OMG	Eventually will have an arbitrary scope when using a Trader for two phase selection.
Internet Mgmt	Agent's entire MIB registration tree, or a subset (MIB View) of that tree.

17.1.18 Specification

OSI Mgmt	Piecemeal, in that Attributes, Notifications, Actions, and Parameters are defined first, then combined into packages, which are then combined into managed object class definitions.
OMG	Object specification in OMG is at the granularity of interfaces. Via inheritance, an object may support many interfaces, so it is not really monolithic. Operations and attributes specified separately could be achieved by putting these individual declarations in their own interface and combining them as required.
Internet Mgmt	Largely monolithic. There is little specification re-useuse except as provided by SNMPv2 (Textual Conventions and Table Augmentation).

17.1.19 Specification Tools

OSI Mgmt	GDMO.
OMG	IDL.
Internet Mgmt	ASN.1 macros.

17.1.20 Specification Formality

OSI Mgmt	Syntax.
OMG	Syntax.
Internet Mgmt	Syntax.

17.2 Analysis of Similarities and Differences

Up to and including Section 17.1.8, there is an encouraging degree of agreement between the fundamental aspects of the OSI and OMG models. The basic notions of objects, object taxonomy, attributes, operations, state, behaviour, and encapsulation are virtually identical. This is important in realising our goal of using object-oriented software development systems to implement OSI-conformant network-management products. The closer the two models are, the less incidental code has to be written to fit the specification (that is, OSI management) model to the implementation (that is, OMG) model. This will not only result in less code, but will also improve accuracy and robustness of implementation.

The OSI and OMG models differ:

- in four major aspects:
  - — support for events
  - — support for multiple replies
  - — support for attribute groups
  - — late binding.
- in a number of incidental aspects, such as specification techniques and associative references
- in two complementary aspects:
  - — interface type
  - — intended use.

Therefore, the OSI management and OMG models only fundamentally differ in late binding (possibly) and multiple replies. The latter is implementable and is a common style for event driven systems, for example, X-windows (DII even enables callback style of programming). The possibility of interworking is strengthened by existence of OMG Object Services.

The following chapter outlines the approach to reconciling the differences between these object models.

Chapter 18

Reconciling the Models

In the previous chapters of this **Object Model Comparison** report, it has been established that, in spite of some differences, the OSI and OMG models are fundamentally in alignment.

This chapter explores how some of the differences can be exploited, and how the remainder can be reconciled.

18.1 Changing the Models

In principle, the models could be reconciled by demanding changes to one or the other, in order to simplify the mappings between them.

The JIDM working group decided that such an approach is not practical, in that large investments have been made based on each of the existing models. It was assumed that mappings between specifications in each model are constrained to the existing definitions of each model.

However, it is anticipated that this JIDM work might be used to suggest enhancements in future revisions to the models, to aid mappings of specifications between each other.

18.2 Exploiting the Differences

Two of the differences between the models (the intended use and interface type) are, as has been pointed out earlier, complementary. Network management systems have to be implemented, ideally using object-oriented software-development tools.

The system aspects of the OMG model can be represented as an invoker invokes operations on a performer (one or more objects), using an abstract procedure-call protocol (abstract in the sense that the syntactical details of what is sent on the wire are not specified). The OSI and Internet Management models exhibit similar invoker/performer characteristics but using a precisely-specified message-passing protocol. Conceptually, the models could be merged, with management protocol intervening between OMG programmatic interface service boundaries.

This approach would satisfy both models. The OMG model would be unaware of the intervening Management protocol, and the end-systems would conform to MGMT protocols. Of course it not quite that simple. Difficult issues such as support for asynchronous operation of the OSI Management model with the potentially synchronous operation of the OMG model would have to be resolved.

Asynchronous OSI MGMT versus synchronous OMG is not a problem if an OMG implementation supports concurrency and/or deferred synchronous calls.

18.3 Reconciling the Differences

Having exploited the complementary differences, it is next necessary to reconcile fundamental and incidental differences. There are a number of approaches:

1. align the models
2. provide run-time mediation between implementations of the models
3. provide notational mapping tools.

The following sections explore each of these approaches.

18.3.1 Model Subset Alignment

Ideally, all of the conflicting differences between the models would be resolved by complete alignment. However, this is impractical. Each model is the result of many man-years of unrelated consensus-building, and it is unlikely that agreements to align the models can be quickly reached. Nevertheless, attempting to develop more compatible subsets of the two models is feasible.

The OMG is restructuring its documentation (see reference **OOM**) to reflect the notion of a core part and a number of component parts, and it is envisaged that profiles based upon this structure will be defined for various applications.

OSI Management has, in its turn, a similar concept. Standards define kernel functionality to guarantee a basic level of interoperability, and a number of additional functional units to permit extended capabilities. Indeed, OMNI*Point* is itself a set of agreements based on OSI standards.

This has the practical disadvantage that there are a multitude of existing definitions of managed objects using all of the glory of GDMO. Profiling would not allow interoperability for systems which must implement heavier object definitions than allowed by the profile.

In particular, since recursive attribute SYNTAX structures cause problems for translation to OMG IDL, they should be used with caution in Managed Object definitions. Such recursive structures may require special consideration for mapping to CORBA IDL, and may in some cases require manual intervention in the translation process.

18.3.2 Run-time Mediation

Run-time mediation essentially requires the development of incidental software to match the differences between the specification (for example, OSI Management) model to the implementation (for example, OMG) model. In particular software needs to be developed to handle notifications, late binding, and multiple replies. It is anticipated that agreed approaches to developing such software will be developed.

The IIMC work, which deals with OSI Management and Internet Management coexistence, has used the term *proxy* to describe devices which perform such run-time mediation. The proxy needs to translate service requests and messages from one domain format to that of another.

Mechanisms for run-time mediation, in general, will depend on tools for notation translation. It is likely that to work properly, any run-time mediation mechanism will require knowledge of the base notation in both the original notation form and the translated notation form. With such knowledge of the notation translation, the run-time mediation mechanisms could dynamically translate messages and service invocations from either form to the other. Thus, the dynamic message translation process may work in either direction, although one notation was chosen as the base and the other notation was derived by notation translation.

18.3.3 Notation Translation Tools

Considerable investment has been made in the notational tools:

- GDMO, used by OSI Management
- SNMP Macros, used by Internet Management
- IDL, used by the OMG.

This investment takes the form of syntax checkers, data-structure generators, and ASN.1 compilers, and also published specifications giving definitions of object types. Alignment of the tools is thus impractical. However, there is good reason to believe that automatic, or at least semi-automatic, translation between any two models is feasible.

The IIMC has specified translation algorithms between Internet Management and OSI Management formal definitions. Their experience has proved that the concept is practical. However the static notational mappings are not invertible, that is, taking a translated definition through the reverse algorithm does not bring back the original definition. Thus, care must be taken as to which notation is selected as the base. However, once a translation of notation from the base to the derived notation is developed, it may be used by the run-time mediation process to convert services / messages from one domain into those of the other.

Chapter 19

Conclusions

It has been shown that the object model defined by the OMG for object-oriented software development and that defined by OSI Management for network management are fundamentally aligned. There are, however, a number of major, and a number of incidental, differences; some are complementary and some are conflicting. In order for implementors of network management products to be able to use development tools and run-time components meeting OMG specifications, the conflicting differences will have to be reconciled. Methods are proposed to achieve this: the development of run-time mediation software; and the development of software to translate between the notational tools.

Glossary

Abstract Syntax Notation One
(ASN.1) A notation defined in the OSI standards that allows data to be described in a machine-independent fashion.

CCITT
Consultative Committee of International Telegraph and Telephone — an international committee whose membership is composed of government postal, telephone and telegraph agencies (PTTs). Now known by the name ITU-T (International Telcommunications Union - Telecommunications Standardisation Sector).

CMIP
Common Management Information Protocol, defined in ISO / IEC 9596-1: 1991.

CMIS
Common Management Information Service, defined in ISO / IEC 9596: 1991.

CORBA
Common Object Request Broker Architecture.

GDMO
Guidelines for the Definition of Managed Objects: OSI Structure of Management Information, Part 4.

IDL
Interface Definition Language. Since there are several versions / sources of such languages in existence, this specification makes clear it uses the OMG CORBA IDL.

ISO
International Organization for Standardization. A standards organisation with the membership composed of the standards organisations from each participating country. OSI working groups generate the OSI Protocol Suite standards.

ITU-T
International Telcommunications Union - Telecommunications Standardisation Sector — an international committee whose membership is composed of government postal, telephone and telegraph agencies (PTTs). Formerly known as the CCITT (Consultative Committee of International Telegraph and Telephone).

MIB
Management Information Base.

NMF
Network Management Forum.

ODP
Open Distributed Processing: ITU-T Recommendations X.902 and X.903.

OMG
Object Management Group.

RFC
OMG organization's Request for Comments process, whereby a proposer of some technology which the OMG has not (yet) itself proposed submits it in specification form to the OMG ''for comment'', with the objective of adoption of that technology by the OMG. Compare this with

the OMG RFP process.

RFP
OMG organization's Request for Proposals process, whereby the OMG issues a requirements document for some technology, and calls for proposals which answer all or part of the issued requirement. Compare this with the OMG RFC process.

Open Systems Interconnection
(OSI) A set of ISO standards that define a network architecture based on a 7-layer model for communication between open systems. OSI management standards define Configuration, Fault, Performance, Security and Accounting Management.

Protocol Data Unit
The data unit exchanged by peer protocol entities. Examples are APDU (Application Layer), PPDU (Presentation Layer) and SPDU (Session Layer).

PIDL
Pseudo Interface Definition Language.

Simple Network Management Protocol
(SNMP) A protocol for managing IPS networks.

TMN
Telecommunications Management Network.

Index