ITU-Telecommunication Standardization Sector **Contribution _____ (WP4/4)**

STUDY GROUP **4**

Geneva, Switzerland, ? – ? January, 2001

Questions: 14/4, 15/4, 19/4

Title: Draft Rec. *"TMN Guidelines for Defining CORBA Managed Objects"*

Source: Editors

Contact:

Keith Allen
SBC Technology Resources
USA
Tel: +1 512 372 5741
Fax: +1 512 372 5791
E-mail: kallen@tri.sbc.com

Lakshmi Raman
Teraburst
USA
Tel: +1 408 541 1155 x322
Fax: +1 408 541 0439
E-mail: lraman@teraburst.com

ABSTRACT

This draft new recommendation specifies guidelines for defining CORBA-based interfaces to software objects representing manageable resources in a TMN. It covers information modeling guidelines, rules for translating models from GDMO, and IDL style conventions. It also provides an IDL module defining data types, superclasses, and notifications to be used in CORBA-based information model specifications.

INTERNATIONAL TELECOMMUNICATION UNION

TELECOMMUNICATION STANDARDIZATION SECTOR
STUDY PERIOD 1997 - 2000

COM 4-xxx-E
August 2000
Original: English

Question: 14/4

STUDY GROUP 4 – CONTRIBUTION ____

SOURCE*: EDITORs

TITLE: DRAFT NEW RECOMMENDATION X.780: TMN Guidelines for Defining CORBA Managed Objects

---

Summary
This draft new Recommendation specifies guidelines for defining CORBA-based interfaces to software objects representing manageable resources in a TMN. It covers information modeling guidelines, rules for translating models from GDMO, and IDL style conventions. It also provides an IDL module defining data types, superclasses, and notifications to be used in CORBA-based information model specifications.

Source
ITU-T Recommendation X.780 was developed by ITU-T Study Group 4 (1997-2000) and was approved under the WTSC Resolution 1 procedure on the **xx of xx xx**.

Keywords
Common Object Request Broker Architecture (CORBA), Interface Definition Language (IDL), Guidelines for the Definition of Managed Objects (GDMO), Distributed Processing, TMN Interfaces, Managed Objects, Abstract Syntax Notation One (ASN.1)

Foreword

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.
The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.
The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1.
In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.
As of the date of approval of this Recommendation, the ITU had/had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

Table Of Contents

Table Of Figures

Table Of Tables

Recommendation X.780

TMN Guidelines for Defining CORBA Managed Objects

(2001)

1 Scope

The TMN architecture defined in Recommendation M.3010 –2000 introduces concepts from distributed processing and includes the use of multiple management protocols. The initial TMN interface specifications for intra- and inter-administration interfaces were developed using the Guidelines for the Definition of Managed objects (GDMO) notation from OSI Systems Management with Common Management Information Protocol (CMIP) as the protocol. The inter-administration interface (X) included both CMIP and CORBA GIOP/IIOP as possible choices at the application layer.

CORBA, a distributed processing technology, is being considered for use in the TMN communication architecture primarily due to its acceptance by the Information Technology industry. This acceptance is expected to enhance the availability of CORBA-based interfaces due to better development tools and wide-spread expertise in developing CORBA-based interfaces. This technology, developed by the Object Management Group (OMG), is also being considered by multiple industries. Specifications using this technology provide support for standard application programming interfaces (APIs) and language bindings to programming languages, and they also facilitate software portability. The interoperability solutions offered by the object request broker combined with the Inter-ORB protocol address interoperability between client and server. While CMIP and information models provide solutions for interoperability between manager and agent systems, CORBA defines inter-object interactions where the objects may be distributed.

1.1 Purpose

Several groups are developing network management specifications that use CORBA modeling techniques with IDL as the notation along with CORBA services. The scope of this standard is to define guidelines suitable for use in the specification of interoperable CORBA-based network management interfaces. Previous standards for CORBA-based network management interfaces have mainly focused on TMN "X" interfaces, which are interfaces between administrations (carriers). The demands placed on these interfaces are different from those used "inside" an administration, "Q" interfaces. The scope of this Recommendation covers all interfaces in the TMN where CORBA may be used. It is expected that not all capabilities and models defined here are required in all TMN interfaces. This implies that the framework can be used for

interfaces between management systems at all levels of abstractions (inter and intra-administration) as well as between management systems and network elements.

ITU-T Recommendation Q.816[1] defines a set of services that are required for CORBA-based TMN interfaces. This Recommendation defines guidelines for specifying information models written in CORBA IDL to which the services are applicable. It also provides rules for translating existing GDMO models to IDL. Finally, it defines some base IDL code for use by all CORBA-based TMN information models. The combination of this Recommendation and Q.816 form a *framework* for defining and implementing CORBA-based TMN interfaces.

Use of a common framework on telecommunications management interfaces has several advantages. Some examples are: facilitating reuse of models that are developed to meet the generic requirements of telecommunications; profiling CORBA services for use by the telecommunications industry; easing the definition of new services for TMN; reusing the semantics of the existing rich set of models; and harmonizing the modeling approach across groups using a single source similar to Recommendations X.720, X.721 and X.722 for CMIP. Re-using a common approach to modeling resources and re-using a generic information model for a variety of network technologies and network management applications will speed the introduction of new network services while keeping network management system development costs down.

The telecommunications industry has invested a great deal of time and energy in the development of information models for the CMIP network management protocol. A primary goal of the TMN CORBA framework is the re-use of these information models by enabling their translation to CORBA Interface Definition Language (IDL) with little change in semantics. As a result, initial IDL information models are expected to be derived from CMIP models.

1.2 Application

Recommendation M.3020 defines three phases in the development of a TMN specification. The three phases are Requirements, Analysis and Design. Figure 1 shows this process and the scope of this Recommendation for developing CORBA based interface specification relative to this process.

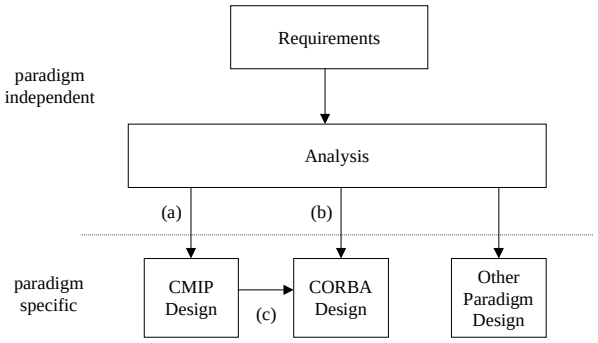

Figure 1. CORBA Based Specification with Requirements Analysis and Design

The requirements and analysis are specified using an approach that is not specific to a network management technology paradigm. The output from the analysis phases is used for development in the design phase. In this phase, network management technology specific features are used to define information models. The arrows marked as (a) and (b) show that the analysis output is mapped to GDMO/ASN.1 based model to use with CMIP or IDL models to use with CORBA/IIOP, respectively. There are no prescriptive rules available at this time to generate these models. It may be possible to develop such rules in the future in M.3020.

This Recommendation addresses the reuse of existing models developed in the CMIP paradigm if CORBA/IIOP is to be used instead of CMIP. The arrow shown as (c) is addressed by this Recommendation.

In developing the transformation from GDMO/ASN.1 definitions to CORB/IDL two approaches are possible. In the first approach, every element of the syntax is translated to CORBA/IDL using a well-specified algorithm or a prescriptive definition. This method, is the one taken with Joint Inter-Domain Management (JIDM) where a gateway can be used to support interoperability. The guidelines in this Recommendation address the design phase for applications where the translation from the existing GDMO definitions preserves the semantics and also uses the features of CORBA. The transformation is not completely prescriptive. This approach is used not for inter-working using gateways but to preserve the requirements and semantics of the models developed to meet the telecommunication context. This is applied when the managing and managed systems are designed to communicate using CORBA/IIOP.

In addition to the recommendations for translating from GDMO information models defined here, Recommendation Q.816 defines recommendations for CORBA services to be used for managing telecommunications networks. Q.816 aspects of the framework are applicable irrespective of how CORBA based specifications are developed (*i.e.*, using the path designated as (b) or (c) in Figure 1).

In addition to taking advantage of CMIP information models, another purpose of the guidelines is to take advantage of CORBA. The framework leverages the functions defined in the CORBA specifications, including a set of Common Object Services. Also, these guidelines re-use CORBA approaches and design patterns wherever they are appropriate. Finally, while re-using existing models is important, it is equally important that the framework support the development of new models. These guidelines do not require a GDMO model to be developed prior to the development of an IDL model. In fact, developing a new IDL information model for use within this framework is straightforward and guidelines for doing so are provided.

ITU-T Recommendation M.3120 [19] provides a CORBA IDL version of the generic network information model originally defined in Recommendation M.3100. The IDL version follows the object modeling guidelines defined here and is designed to use CORBA-based TMN services defined in ITU-T Recommendation Q.816.

1.3 Document Roadmap

This document has the following structure:

Section 1.	Introduction, document roadmap, and updates.
Section 2.	References.
Section 3.	Definitions of abbreviations used throughout the rest of the document.
Section 4.	Requirements for the object modeling guidelines. These are the design goals the guidelines must meet.
Section 5.	Description of the CORBA IDL module that defines interfaces to be used and sub-classed in network management interface specifications. The actual IDL is in Annexes A and B.
Section 6.	Guidelines for defining CORBA-based TMN information models. These guidelines are specifically designed for IDL objects using the TMN CORBA-based services in Recommendation Q.816.
Section 7.	Guidelines for translating GDMO information models to IDL models suitable for use with the TMN CORBA-based services in Recommendation Q.816.
Section 8.	Style idioms for CORBA IDL network management interface specifications.
Section 9.	Compliance and conformance guidelines.
Annex A.	The IDL module for the modeling guidelines specification. This annex is normative.
Annex B.	Additional IDL defining constants used by the modeling guidelines. This annex is normative.

1.4 Document Conventions

A few conventions are followed in this document to make the reader aware of the purpose of the text. While most of the document is normative, paragraphs succinctly *Editor's note – This statement on normative aspect may need to be modified based on ITU rules.*

stating mandatory requirements to be met by a management system (managing and/or managed) are preceded by a boldface "R" enclosed in parentheses, followed by a short name indicating the subject of the requirement, and a number. For example:

(R) EXAMPLE-1 An example mandatory requirement.

Requirements that may be optionally implemented by a management system are likewise preceded by an "O" instead of an "R." For example:

(O) OPTION-1 An example optional requirement.

The requirement statements are used to create compliance and conformance profiles.

Many examples of CORBA IDL are included in this document, and IDL specifying the data types and base classes are included in normative annexes. The IDL is presented in a 9-point courier typeface:

```
// Example IDL
interface foo {
        void operation1 ();
};
```

Instructions for extracting the IDL from an electronic version of this document and compiling it are presented in the next section.

1.5 Compiling the IDL

An advantage of using IDL to specify network management interfaces is that IDL can be "compiled" into programming code by tools that accompany an ORB. This actually automates the development of some of the code necessary to enable network management applications to interoperate. This document has two annexes that contain code that implementers will want to extract and compile. Both Annex A and Annex B are normative and should be used by developers implementing systems that conform with this standard. The IDL in this document has been checked with two compilers to ensure its correctness. A compiler supporting the CORBA 2.3 specification must be used.

The annexes have been formatted to make it simple to cut and paste them into plain text files that may then be compiled. Below are tips on how to do this.

1. Cutting and pasting seems to work better from the Microsoft® Word® version of this document. Cutting and pasting from the Adobe® Acrobat® file format seems to include page headers and footers, which cannot be compiled.
2. All of Annex A, beginning with the line "/* This IDL code…" through the end should be stored in a file named "itut_x780.idl" in a directory where it will be found by the IDL compiler.
3. All of Annex B, beginning with the line "/* This IDL code…" through the end should be stored in a file named "itut_x780Const.idl" in the same directory as the file containing Annex A.

4. The headings embedded in these annexes need not be removed. They have been encapsulated in IDL comments and will be ignored by the compiler.
5. Comments that begin with the special sequence "/**" are recognized by compilers that convert IDL to HTML. These comments often have special formatting instructions for these compilers. Those that will be working with the IDL may want to generate HTML as the resulting HTML files have links that make for quick navigation through the files.
6. The annexes have been formatted with tab spaces at 8-space intervals and hard line feeds that should enable almost any text editor to work with the text.

2 References

2.1 Normative References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

[1] ITU-T Recommendation Q.816, *CORBA-Based TMN Services.*

[2] The Object Management Group (OMG), "The Common Object Request Broker: Architecture and Specification", OMG Document formal/99-10-07, Revision 2.3.1, October, 1999.

[3] The Object Management Group (OMG), "JIDM Interaction Translation," Edition 4.31, OMG TC Document telecom/98-10-10, October 1998.

2.2 Additional References

The following standards contain information that was used in the development of these guidelines. As stated in the introduction, a primary design goal of these guidelines is to enable the re-use of existing network management information models, at least without significant semantic changes. These documents provide many of the details on the ITU-T's CMIP framework, and therefore define some of the functionality the CORBA object modeling guidelines must support.

[4] ITU-T Recommendation X.703 (1997), *Information Technology – Open Distributed Management Architecture*, October, 1997.

[5] CCITT Recommendation X.720 (1992) | ISO/IEC 10165-1 : 1992, *Information Technology – Open Systems Interconnections – Structure of Management Information: Management Information Model*.

[6] CCITT Recommendation X.721 (1992) | ISO/IEC 10165-2 : 1992, *Information Technology – Open Systems Interconnections – Structure of Management Information: Definition of Management Information*.

[7] CCITT Recommendation X.722 (1992) | ISO/IEC 10165-4 : 1992, *Information Technology – Open Systems Interconnections – Structure of Management Information: Guidelines for the Definitions of Managed Objects*.

[8] ITU-T Recommendation X.720 Cor. 1, *Corrigendum 1 to CCITT Recommendation X.720*, February, 1994.

[9] ITU-T Recommendation X.721 Cor. 1, *Corrigendum 1 to CCITT Recommendation X.721*, February, 1994.

[10] ITU-T Recommendation X.721 Cor. 2, *Corrigendum 2 to CCITT Recommendation X.721*, October, 1996.

[11] ITU-T Recommendation X.721 Am. 1, *Amendment 1 to CCITT Recommendation X.721*, November, 1995.

[12] ITU-T Recommendation X.722 Cor. 1, *Corrigendum 1 to CCITT Recommendation X.722*, October, 1996.

[13] ITU-T Recommendation X.722 Cor. 2, *Corrigendum 2 to CCITT Recommendation X.722*, January, 2000.

[14] ITU-T Recommendation X.722 Am. 1, *Amendment 1 to CCITT Recommendation X.722*, November, 1995.

[15] ITU-T Recommendation X.722 Am. 2, *Amendment 2 to CCITT Recommendation X.722*, August, 1997.

[16] ITU-T Recommendation X.722 Am. 3, *Amendment 3 to CCITT Recommendation X.722*, August, 1997.

[17] CCITT Recommendation X.733 (1992) | ISO/IEC 10164-4 : 1992, *Information Technology – Open Systems Interconnection – Systems Management: Alarm Reporting Function.*

[18] ITU-T Recommendation M.3010 (2000), *Principles for a Telecommunications management network*, February, 2000.

[19] ITU-T Recommendation M.3120, *CORBA-Based Generic Network Information Model.*

[20] ITU-T Recommendation Q.821 (2000), *Stage 2 and Stage 3 description for the Q3 interface - Alarm Surveillance*,(to be published).

3 Definitions

This section provides definitions for acronyms used throughout the rest of the document.

ASN.1	Abstract Syntax Notation #1.
ATM	Asynchronous Transfer Mode.
CMIP	Common Management Information Protocol.
CORBA	Common Object Request Broker Architecture.
COS	Common Object Services.

DN	Distinguished Name.
EMS	Element Management System.
GDMO	Guidelines for the Definition of Managed Objects.
GIOP	General Interoperability Protocol.
HTML	Hypertext Markup Language.
ID	Identifier.
IDL	Interface Definition Language.
IIOP	Internet Interoperability Protocol.
IOR	Interoperable Object Reference.
ITU-T	International Telecommunication Union – Telecom.
JIDM	Joint Inter-Domain Management.
MO	Managed Object.
NE	Network Element.
NMS	Network Management System.
OAM&P	Operations, Administration, Maintenance, and Provisioning.
ORB	Object Request Broker.
OID	Object Identifier.
OMG	Object Management Group.
OSI	Open Systems Interconnection.
PDU	Protocol Data Unit.
QoS	Quality of Service.
RDN	Relative Distinguished Name.
TMN	Telecommunications Management Network.
TTP	Trail Termination Point.
UID	Universal Identifier.
UML	Unified Modeling Language.
UTC	Universal Time Code.

4 CORBA Modeling Goals and Requirements

This section describes the key goals for modeling TMN resources using CORBA, and the requirements that the modeling guidelines must meet to support these goals. Section 4.1 introduces the goals of the modeling guidelines. Subsequent sub-sections then provide terminology and requirements. The requirements in Section 4 are requirements that the framework must satisfy. They are based on the telecommunications management needs. Sections 5, 6, 7, and 8 then describe modeling guidelines that meet these needs and define how to achieve the requirements of section 4 by using CORBA in a certain way. The rules in Section 5, 6, 7, and 8 on how to use CORBA also are referred to as requirements.

4.1 Goals

This document specifies guidelines for defining CORBA managed objects for use on interfaces supported by telecommunications network management systems and network elements. Some key goals of the modeling guidelines are:

- Application Interoperability
- Common Usage of CORBA Common Object Services

- Information Model Transparency

This section elaborates on these three goals.

4.1.1 Application Interoperability

A key goal of the TMN architecture, and in particular the information architecture, is to promote a standard framework for providing interoperability and information exchange between systems from a diverse set of network management system suppliers. Interoperability between systems involves many aspects of development. At its lowest layer, a common communication mechanism must be in place to support a common syntax, the establishment of connectivity and the exchange of operation requests/replies between systems. This aspect of interoperability is inherently supported by the CORBA specification.

For TMN, there is the need to provide application interoperability. That is, management systems from diverse suppliers will be utilized within a single administration's TMN to support different functions necessary to support management of its networks. To simplify integration of these various suppliers' systems, they must agree on the semantics of the information being exchanged. This is accomplished with the specification of an information model. This document specifies the rules for defining these information models.

4.1.2 Common Usage of CORBA Common Object Services

A second aspect of these guidelines is the reliance upon a common usage and profiling of the distributed processing environment of choice. Rather than re-defining the interface capabilities needed to support common network management functions such as object naming and notification filtering with each information model, these guidelines rely upon a set of support services. These support services enable the information models to be simpler, and also enhance interoperability. The support services required for CORBA based interfaces are specified in Recommendation Q.816.

4.1.3 Information Model Transparency

If CORBA is used in places within the TMN architecture where existing information models (e.g. GDMO) are well established, then the framework must support the reuse of those models without any major changes.

A single standard way to map these GDMO information models to OMG IDL is needed so that the same models are always presented by the application protocol to the application with the same set of services (capabilities).

4.2 Entities

An **entity type** describes a type of "thing" in the real world with an independent existence. An **entity type** may be an object with a physical existence – a circuit pack, managed element, or slot – or it may be an object with a conceptual existence – a subnetwork, termination point, or link. Each **entity type** has particular properties, called attributes that describe it.

An **entity instance** (or **entity**) describes a particular instance of an **entity type** (e.g., Circuit Pack #1). Each **entity's** attributes are described by particular values that represent the state of that instance. In addition, each **entity** must be uniquely identifiable.

In CORBA, an **entity** may have many manifestations. An **entity** may be represented by an IDL data structure, a value type, an interface type or a component. This document describes how CORBA is utilized to model **entities**.

4.2.1 Access Granularity

In the context of TMN operations, granularity defines the level of abstraction that is exposed between systems. Access Granularity identifies the level at which **entities** may be accessed (i.e., how information is exposed via an interface). For CORBA, each CORBA object is provided a unique address known as an Interoperable Object Reference (IOR). The IOR provides an address to the client system identifying which server system to connect to for communication with the server side CORBA object.

In CORBA, it is possible to define different access abstractions (i.e., access granularity) to the Entities defined for TMN (e.g., ITU-T Rec. M.3100). Two different access abstractions are defined here:

1) **Instance granularity**: Each entity has its own IOR. For the creation of new Entities, this implies the instantiation of a new CORBA object.
   - 1 IOR / entity instance

   For example, an **entity type** in the ATM domain is an *atmLink*. In the Instance Grained approach, a CORBA object is defined that supports the same attributes as the entity type which it represents. For each instance of the *atmLink*, an independent CORBA object is created. Thus each *atmLink* can be uniquely addressed by its IOR.

2) **Application-specific granularity**: Instances of a well-defined set of entity types are accessed via a single IOR (a single interface).
   -1 IOR / Family (set of) entity types
   Bulk operations are defined in application-specific CORBA IDL interfaces, which pass identities and states of managed entities using operation parameters employing lists of IDL structured types.

The CORBA object modeling guidelines defined in this specification are applicable to the specification of managed object interfaces that support instance-grained access granularity. TMN standards may also be defined using application-specific access granularity. Such interface specifications, however, are outside the scope of this Recommendation.

4.3 Principles of Containment and Naming

Containment is a logical representation of how entities of one type contain entities of another type. A Containment Tree defines the relationship between the entity instances. An entity instance is contained by one and only one containing entity instance. Containing entity instances may themselves be contained in another entity instance

forming a directed graph. The directed graph forms what is called the Naming (or Containment) Tree.

The containment relationship can be used to model real-world hierarchies of parts (e.g., assembly, sub-assemblies and components) or real-world organizational hierarchies (e.g., company name, org. name).

An example of a possible containment tree is shown in Figure 2 below.

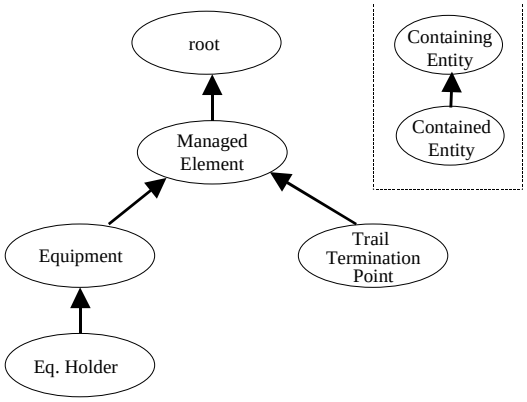

Figure 2. Example of Containment

4.3.1 Naming

One purpose of containment relationships is for naming entities. Names are designed to be unique in a specified context; for TMN, this context is determined by the containing entity instance.

An entity that is named in the context of another entity is termed a "Subordinate Entity". The entity that establishes the naming context (this term is used in general and should not have the direct connotation of a COS Naming Service Naming Context) for other entities is called the "Superior Entity".

A "Subordinate Entity" is named by the combination of:

- The name of its "Superior Entity".
- Information uniquely identifying this “Subordinate Entity” within the scope of its superior entity.

The name of an entity that is unique in a local naming context may not be so in some larger naming context. However, if the local naming context is unique in the larger context, a local name can be made unique by qualifying it by its naming context; the name of the naming context is used as the qualifier. This arrangement can be visualized as a directed graph with each edge (or arrow) pointing from a named object to a naming context.

The naming context can itself be recursively qualified by another naming context, so the complete naming structure can be visualized as a single-rooted hierarchy. This hierarchy is called the naming tree. Thus "Superior Entities" become the naming contexts and their names become the names of the contexts. An object name need only be unique within the context of its superior Entities; within a wider context its name is always qualified by names of it superior Entities.

4.3.2 Entity Identification

Because a “Superior Entity” may contain multiple “Subordinate Entities” of the same type, each of these contained entities of the same type must be distinguishable relative to their containing entity. The relative name of an entity within its containing entity is called an entity’s **Relative Distinguished Name** (RDN). For example, there may be several equipment holders within a managed element. To uniquely identify each equipment holder within the managed element, the equipment holders must be provided an RDN. The RDN should identify the name of the entity type (e.g., equipment holder, which is an entity type) and a unique value within the scope of the containing entity.

An RDN is a basic element of a **Distinguished Name** (DN), as specified in ITU-T Rec. X.720. A DN is defined by a sequence of RDNs starting from a specific context. The DN yields a unique name relative to this context.

4.4 Managed Object Classes

These modeling guidelines specify that each entity type maps one-to-one with a CORBA operational interface. When an entity type is mapped in this manner, the CORBA object representing the entity type is called a *Managed Object Class*. A Managed Object Class must also exhibit the ability to emit notifications (see ITU-T Rec. X.703).

The term “Managed Object Class” is defined in ITU-T Rec. X.720. As explained in ITU-T Rec. X.703, managed object classes and sub-classes map to interfaces and derived interfaces.

4.5 Packages

It is necessary to capture the notion of packages in CORBA IDL. Packages are groups of capabilities (attributes, actions, or notifications) that may be conditionally supported by a Managed Object Instance. A managing system must have the capability to determine which packages are supported by a Managed Object Instance. If any operations are performed on a Managed Object, and those operations are contained by a Conditional Package that is not instantiated for that Managed Object, then the Managed Object must indicate an error.

4.6 Attributes

The guidelines must support the definition of attributes (i.e., visible properties) on Managed Object Classes.

4.6.1 GET and SET

The value of an attribute may be observable or modifiable across a standard interface. If observable, the information modeler must define a "get" method for that attribute. If modifiable, the information modeler must define a "set" method for that attribute.

4.6.2 Generic Attribute Get

CORBA-based TMN information models should allow a managing system the ability to read arbitrary groups of attributes from a single managed object with a single operation. This service allows many management tasks to be performed with a single operation. Support of the Generic Attribute Get is required.

4.6.3 Set-valued Attributes

For attributes containing lists of values, a modeler should have to capability to allow managing systems to add or remove individual values to/from lists without resending all the information in the original list.

4.7 Creation and Deletion of Managed Objects

The existence of Managed Objects (MOs) is closely related to the containment relationship between the MOs. A MO's existence is tied to the existence of that MO's superior MO Instance. If the specified “Superior MO” does not exist for a “Subordinate MO”, then that “Subordinate MO” can not be created. Similarly, if a MO's “Superior MO” is deleted, then that “Subordinate MO” (and the “Subordinate MO's” subordinates) can no longer exist. Given this, there are creation and deletion semantics that must be enforced by the TMN CORBA framework.

The following sections define the high-level requirements that must be supported for object creation and deletion. Recommendation Q.816 describes the generic services used to carry out creation (i.e., the factory) and deletion (i.e., the factory in coordination with the terminator service). Section 6 defines modeling guidelines for how the requirements defined in this section are supported.

4.7.1 Creation

When creating a Managed Object, three aspects of the MO’s existence must be identified:

- The MO’s name
- The MO’s attribute values
- The conditional packages of the MO that are to be instantiated with the creation of the new MO.

Note that definition of these aspects in the create request may be either explicit or implicit. Options for identifying these aspects of a MO’s existence are defined in the following three sections.

4.7.1.1 Identification of the MO Name

The name of the MO to be created can be determined in one of two ways:

1. The manager may specify, as a parameter of the create operation, a reference to an existing MO which is to be the superior of the new MO and may specify the RDN of

the new MO in the create operation's attribute list. This results in the complete specification of the MO name being supplied by the manager.

2. The manager may specify, as a parameter of the creation operation, a reference to an existing MO which is to be the superior of the new MO and may omit specifying the RDN of the new MO. In this case, the RDN of the new MO is assigned by the managed system.

If the associated information is not correct or for some other reason the create operation can not be performed then the factory attempting to perform the operation shall indicate an error.

4.7.1.2 Identification of the MO Attributes

When a MO is created, its attributes are assigned values that are valid for the type of attribute. These values are derived from information in the Create operation and the MO class definition in one of the two manners listed below:

1. The create request is permitted to specify an explicit value for each individual attribute. When the MO is created, explicit values are assigned to attributes as required by the MO class definition.
2. The MO class definition is permitted to specify how default values are assigned to attributes that are not set by the create operation.

If default values are not specified for an attribute, then the managing system must supply a value for that attribute in the create request. If no value is specified for that attribute, then an error should occur.

If an explicit value is defined for a particular attribute in the create request, then the MO will take that value for the specified attribute over any potential default value that may be specified for that attribute.

4.7.1.3 Identification of MO Packages for Instantiation

To ensure that underlying resources can be instantiated with required capabilities, the manager must be able to specify the capabilities (i.e., the conditional packages) that the managed object should have instantiated.

Instantiation of a conditional package will occur if an associated condition is satisfied for the managed object being instantiated. The manager may also request the instantiation of a conditional package as part of the create request, by including it in the packages attribute of the create request.

4.7.2 Deletion

For deletion, deletion semantics may support the deletion of all contained entities while in other cases, the delete method immediately fails if there are contained subordinate entities. These semantics must be maintained for each entity type.

4.8 Inheritance

One "Managed Object Class" may be defined as a specialization of another "Managed Object Class" by utilizing inheritance. Specialization of a "Managed Object Class" implies that all methods and attributes defined on the superclass will also be supported by the subclass.

In CORBA IDL, an attribute or operation cannot be inherited from more than one interface, nor can an inherited operation or attribute be redefined by a subclass. (Note that, in general, it is not expected that a CORBA information model would define a method or attribute in a class, where that same method or attribute may also be defined in the superclass. However, there are cases in the mapping from GDMO to IDL where this may occur. For example, because GDMO attributes specify permitted and required values, a subclass in GDMO may sometimes redefine the same attribute. Care must be taken when mapping to IDL that the same attribute is not redefined.)

A subclass in CORBA can not inherit the same attribute or method (with the same name) from more than one superclass (unless they in turn inherited it from the same base class). Also, a subclass can not redefine the same attribute or method (with the same name) defined in one of its superclasses.

These guidelines place no constraints over CORBA inheritance.

5 The Object Model IDL Module

Before describing the rules for defining TMN managed objects using CORBA Interface Definition Language (IDL),[2] this section presents a network management module containing a set of object interfaces and supporting data structures specified in CORBA IDL. This IDL module is intended to play a role in CORBA-based network management similar to that played by the GDMO and ASN.1 definitions in ITU-T Recommendation X.721[6] for CMIP. It provides the basic set of IDL definitions on which information models are then built.

The IDL is included in Annexes A and B of this document. Annex A contains the base classes (interfaces), data structures, and notifications. Annex B is a separate file containing just constant definitions. Both of these are based on the GDMO and ASN.1 definitions found in X.721.

X.721 is a convenient source for capabilities that must be provided in network management information models. X.721 defines the following managed object classes using GDMO:

- 9 types of records (Log Record, Event Log Record, Alarm Record, Attribute Value Change Record, Object Creation Record, Object Deletion Record, Relationship Record, Security Alarm Report Record, State Change Record)
- Discriminator and Event Forwarding Discriminator
- Log
- System

- Top

Each of these has attributes, actions, and supporting data types and parameters. In addition, X.721 defines 15 notifications.

Looking at the managed object classes listed above, it is clear that many of these are covered by the CORBA Common Object Services already included in the framework (see ITU-T Recommendation Q.816 for details on the TMN CORBA Based TMN Services):

- The CORBA Telecom Event Log service defines a structure for holding log records, so the record classes need not be redefined. (Note that by specifying the use of the CORBA Telecom Event Log Service the TMN CORBA framework treats log records as data structures, not objects.)
- The CORBA Notification Service defines a filtering capability, so the discriminator and event forwarding discriminator need not be redefined.
- The CORBA Telecom Event Log Service defines the equivalent of X.721's Log.

That leaves just System and Top, along with the notifications. System is not really a framework class and belongs instead in a generic information model (if it is needed). The IDL in Annex A, therefore, defines a "top" managed object interface, called "Managed Object," that is intended to be subclassed by all other managed object interfaces similar to the way the managed object class named "Top" is subclassed by all CMIP managed object classes. Also included is a generic "factory" object. Managed object factories are used for object creation. (The CORBA based TMN services defined in ITU-T Recommendation Q.816 includes a *Terminator* service that handles object deletions independent of object type, but object creation is handled by class-specific factories so that object creation operations may be strongly typed.) The notifications are defined on a third IDL interface. In addition, a number of IDL data types are defined. Finally, some IDL pre-compiler macros are defined to ease managed object interface specification. Each of these is discussed below.

5.1 The Base (Top) Managed Object Interface

The first interface defined in Annex A is the *ManagedObject* interface, found after all the data type definitions. It is intended to be the base managed object interface from which all other interfaces inherit. It defines a set of capabilities that all managed object instances must support. These capabilities are:

- A method that returns the name of the object.
- A method that returns the interface (actual class) name of the object.
- A method that returns the conditional packages supported by the object instance.
- A method that returns the creation source of the object (whether it was created autonomously by the managed resource, in response to a management operation, or unknown).
- A method that returns the delete policy for the instance. This is an enumerated value and indicates if the object is not deletable, if it is deletable only if it contains no objects, or if all contained objects will be deleted when it is deleted.

- A method that returns a CORBA value type object containing all of the readable attributes for the object.
- A destroy operation.

The IDL describing the *ManagedObject* interface (without comments) is:

```
interface ManagedObject {

        NameType nameGet()
           raises (ApplicationError);

        ObjectClassType objectClassGet()
           raises (ApplicationError);

        StringSetType packagesGet()
           raises (ApplicationError);

        SourceIndicatorType creationSourceGet()
           raises (ApplicationError);

        DeletePolicyType deletePolicyGet()
           raises (ApplicationError);

        ManagedObjectValueType attributesGet (
           inout StringSetType attributeNames)
           raises (ApplicationError);

        void destroy()
           raises (ApplicationError, DeleteError);
```

}; // end of ManagedObject interface

5.1.1 The nameGet() *Operation*

The first operation, *nameGet*(), returns the CORBA name of the object. *NameType* is a type definition for the CORBA Naming Service *Name* type. *NameType* is used to conform to the IDL conventions defined later in this document. This method returns the compound name of the object, beginning with the name assigned to the local root naming context under which the object is contained. That is, the method returns the "globally unique" name for the object. See ITU-T Recommendation Q.816 for details on assigning a unique name to the root naming context of a managed system. The *ApplicationError* exception is defined to be raised by any managed object operation if the operation cannot be completed due to some resource problem. See Section 5.5 below for details on this and all the other exceptions.

5.1.2 The objectClassGet*() Operation*

The *objectClassGet*() operation returns the scoped interface name (actual class name) of the object. Scoped interface names include the name(s) of the module(s) in which the interface is defined. The return value type, *ObjectClassType*, is a type definition for string. If the object's class is a minor extension of another class (e.g., an "R1" class), the string returned is the name of the actual class (with the "R1"). For example, "EquipmentR1".

5.1.3 The packagesGet*() Operation*

The *packagesGet*() operation returns the list of conditional packages supported by an object instance. The notion of conditional packages, each with a string name, is supported by these guidelines. See Section 6.6 for details. *StringSetType* is a type definition for a list of strings.

Note that this differs slightly from the *packages* attribute on CMIP objects because this framework does not support the definition of mandatory packages, only conditional. In CMIP it is possible for the packages attribute to list mandatory packages. Obviously, since the definition of mandatory packages is not supported by this framework, they can't be listed in the packages attribute of a managed object.

5.1.4 The creationSourceGet*() Operation*

The *creationSourceGet*() operation returns a value indicating the system that caused the object to be created. *SourceIndicatorType* is an enumerated type with three values: *resourceOperation*, *managementOperation*, and *unknown*. It indicates if the object was created autonomously by the resource, in response to a management operation, or if it is unknown why the object was created.

5.1.5 The deletePolicyGet*() Operation*

The *deletePolicyGet*() operation returns the delete policy for this object instance. This is an enumerated value that indicates if the object is not deletable, if it is deletable only if it contains no objects, or if all contained objects will be deleted when it is deleted. (Deleting an object but not its contained objects is not allowed.) This policy is set when the object is created by its factory based on the name binding information identified in the create operation.

5.1.6 The attributesGet*() Operation*

The *attributesGet*() method is used to return all, or any subset, of an object's attribute values in one operation. For each managed object interface in an information model, a CORBA *valuetype* containing data members for each of the readable attributes on that interface will be defined. (Readable attributes are those with an <attribute name>Get() operation.) This method may be used to retrieve this value type for any managed object. The value types will be defined following the inheritance hierarchy of the managed object interfaces (except that value types cannot support multiple inheritance), and each will ultimately be derived from the *ManagedObjectValueType* defined for the *ManagedObject* interface. The managed object must return a value type defined for its interface in response to this method. Thus, when a client invokes the *attributesGet*() operation on any managed object, it will receive back a reference to a *ManagedObjectValueType* which it may then narrow (cast) to the value type defined for the interface on which the operation was invoked.

Complicating this somewhat are the concerns that a client may not want to retrieve all of the attribute values from an instance, and an instance may not support all of the attributes that are in conditional packages. (The value types include attributes in conditional packages.) This is accommodated through the use of the in/out *attributeNames*

parameter. On invocation, the client may submit a list of the names of the attributes in which it is interested, with a null list having the special meaning that all supported attributes should be returned. Any names on the list that are not valid attribute names should be ignored by the managed object. In its response the object will return the actual list of attributes for which values are supplied. Note that this list may not match the submitted list. The object must always return an accurate list, even if the submitted list was null or had invalid names. If all the names on the submitted list are invalid, the object should return a null list and an empty value type.

Because the structure of the value type is pre-defined, the object must fill in some value for the attributes not requested or not supported. Basically, the object may return any values for these attributes, but the values should be as short as possible for efficiency. Thus, null values should be returned for strings, references, and lists of any kind. Any value may be returned for integers and enumerated types. The client must consider any value for an attribute not named in the list returned by the object to be invalid.

The base interface *ManagedObject* currently only has a method that returns a CORBA value type containing all of the readable attributes for the object. It does not contain a similar method for setting the attributes because not all attributes are settable.

5.1.7 The destroy*() Operation*

The final operation on the object, the *destroy*() operation, is used to release any resources associated with the managed object and to delete it. The *DeleteError* exception is raised by the object if it has a delete policy of *NotDeletable*. The *DeleteError* exception is also an extensible means of reporting problems destroying an object that are model-dependent. For example, trying to delete a Trail Termination Point object before the Trail is deleted might result in a *DeleteError*. ITU-T Recommendation Q.816 defines a service called the "Terminator Service," however, to implement the logic needed to enforce delete policies and to maintain the integrity of the naming tree. The destroy operation is actually intended to be used by this service, and should not be directly invoked by a managing system. See ITU-T Recommendation Q.816 for details on the Terminator Service.

(R) OBJECT-1. The interfaces used to model resources on a managed system shall inherit (directly or indirectly) from the *ManagedObject* interface described above and defined in the CORBA IDL in Annex A. The capabilities described above shall be supported.

5.2 The Managed Object Factory

Sometimes managed objects are created automatically by the managed system, sometimes they are created as a result of an action on another object (such as a cross-connection object created in response to a connect action on a fabric), and sometimes they are created in response to a request from a manager to create an object. In this last case, on CMIP systems, the create operation is typically handled by the CMIP agent framework. It can't be handled by the object itself because it hasn't been created yet. In CORBA implementations there is no agent framework, so something needs to be present

on the managed system to enable the managing system to create objects. In CORBA systems this is often handled by "factory" objects. The *ManagedObjectFactory* interface is intended to be the base interface from which other factory interfaces inherit. It will define capabilities that all managed object factories are expected to support. Currently, no such capabilities have been identified, so the interface is null (inherits from nothing and has no attributes or methods). It is a placeholder in which capabilities may be placed in the future if needed. It also serves as a common superclass for all factories.

CORBA IDL information models are expected to include a factory interface per managed object interface (unless the managed object class is not instantiable). The factories will contain operations for creating managed objects. These operations will take a number of parameters, such as the new object's superior object, the new object's name, and values for each of the writeable or set-by-create attributes, etc. Upon successful creation of the new object, the factory will return a reference to it.

In addition to creating objects, it is expected that factories will also create name bindings in the CORBA Naming Service for the new objects. Though this functionality could be implemented elsewhere, it is believed that implementing it in the factories will simplify implementations by relieving the managed object implementation from this task, leaving them to focus on representing resources. See ITU-T Recommendation Q.816 for details on how the TMN CORBA framework makes use of the CORBA Naming Service.

To help clients find factories, ITU-T Recommendation Q.816 defines a Factory Finder Service. This service acts as a broker between clients and factories. Basically, factories register themselves with the service, then clients query the well-known service to find a factory of a particular type. See ITU-T Recommendation Q.816 for details on the Factory Finder Service.

(R) FACTORY-1. The factory objects used to create managed objects on a managed system shall inherit (either directly or indirectly) from the *ManagedObjectFactory* interface described above and defined in the CORBA IDL in Annex A.

(R) FACTORY-2. All factories shall be registered in the Factory Finder object(s) instantiated on that system.

5.3 The Notifications Interface

The third interface defined in Annex A is the notifications interface. Each of the notifications in X.721 has a corresponding operation on this interface. The notifications are defined as typed method calls as required by ITU-T Recommendation Q.816. The OMG Notification Service is used to filter and broadcast notifications. The typed notification methods can be used directly with a notification service that supports typed notifications. Mappings between these typed event methods and structured events are provided in Q.816.

All of the notification operations defined in this interface pass a number of parameters, some of which are common to all of the notifications. Several of the notifications have

identical parameters, but are used for slightly different reasons. The notifications interface IDL looks like this:

```
interface Notifications {

 void equipmentAlarm (
        in ExternalTimeType                 eventTime,
        in NameType                         source,
        in ObjectClassType                  sourceClass,
        in NotifIDType                      notificationIdentifier,
        in CorrelatedNotificationSetType    correlatedNotifications,
        in AdditionalTextType               additionalText,
        in AdditionalInformationSetType     additionalInfo,
        in ProbableCauseType                probableCause,
        in SpecificProblemSetType           specificProblems,
        in PerceivedSeverityType            perceivedSeverity,
        in BooleanTypeOpt                   backedUpStatus,
        in NameType                         backUpObject,
        in TrendIndicationTypeOpt           trendIndication,
        in ThresholdInfoType                thresholdInfo,
        in AttributeChangeSetType           stateChangeDefinition,
        in AttributeSetType                 monitoredAttributes,
        in ProposedRepairActionSetType      proposedRepairActions,
        in BooleanTypeOpt                   alarmEffectOnService,
        in BooleanTypeOpt                   alarmingResumed,
        in SuspectObjectSetType             suspectObjectList
        );
...

}; // end of Notifications interface
```

The other fourteen notification operations are similar to the one above. The names of the 15 notifications defined are:

- Attribute Value Change
- Communications Alarm
- Environmental Alarm
- Equipment Alarm
- Integrity Violation
- Object Creation
- Object Deletion
- Operational Violation
- Physical Violation
- Processing Error Alarm
- Quality of Service Alarm
- Relationship Change
- Security Violation
- State Change
- Time Domain Violation

This CORBA Framework requires the use of notification identifiers where they may not be required in other interfaces (they are not required in ITU-T X.733). To illustrate, below are four possible cases where the mapping of alarm notification identifiers from the network element / EMS interface to the EMS / NMS interface must be done:

1. The network element always uses notification identifiers and the managed object is represented in both interfaces. In this case, the EMS passes the alarm (with its notification identifier) on to the NMS.

2. The network element never uses notification identifiers and the managed object is represented in both interfaces. In this case, the EMS uses an internal counter, includes this value as the notification identifier and passes the alarm onto the NMS.

3. The network element sometimes uses notification identifiers and the managed object is represented in both interfaces. Because the notification identifier is required, the EMS must define a value when one is not provided. It may be difficult to define a value at the EMS because notification identifier values must be unique across all notifications of a particular managed object instance throughout the time that correlation is significant [17]. Thus, the EMS must choose a value that is not being used in current alarms and will not be used in subsequent alarms. Extra care must be taken when doing this, since the algorithm for choosing notification identifier values is owned by the producing system (in this case, the network element).

   In one possible solution, the EMS could supply its own value for notification identifier for all alarms. This would also require the updating of each alarm's correlated notification lists, resulting in the EMS maintaining a complete mapping of network element Notification Identifier values to EMS Notification Identifier values.

   In another possible solution, the EMS and network element could agree on supporting different subsets of notification identifier numbers.

   Alternatively the EMS could supply its own number and ignore potential collisions, thus allowing their rare occurrence.

4. An alarm is mapped from one network element / EMS interface object to a different EMS / NMS interface object. Similar to the above item, the EMS must supply a notification identifier value that is unique for the EMS / NMS managed object. The correlated notification lists also must be updated.

5.4 The Data Type Definitions

Preceding the interface definitions in Annex A are a number of data structure and type definitions. Most of these are used in the notifications. These were derived from the ASN.1 module in X.721 with minor changes to simplify syntax. Where possible, modern object-oriented concepts such as in/out parameters and exceptions have been employed and are reflected in these types.

One data type to note is the time type. These guidelines adopt the universal time code defined for CORBA's Time Service. This data type consists of a large integer that counts the hundreds of nanoseconds that have passed since midnight 15 October, 1582. To account for worldwide time, the time is expressed relative to the time in the Greenwich time zone using a signed short integer for the difference. This means systems based on these guidelines must know their local time zone. This approach makes it easy to compare times, though, because time is represented as an integer. Standard libraries for converting between the integer representation and more familiar formats will likely be widely available.

5.5 Exceptions

The IDL Module in Annex A defines some exceptions for use by managed object operations. These may be raised on some operations, as defined below. In addition, any of the standard CORBA exceptions may be raised on any operation. For example, the "CORBA:NO_PERMISSION" exception might be raised to signal a security violation. The exceptions defined are:

```
valuetype ApplicationErrorInfoType {
          public UIDType          error;
          public Istring          details;
};

valuetype CreateErrorInfoType : ApplicationErrorInfoType {
          public MOSetType        relatedObjects;
          public AttributeSetType attributeList;
};

valuetype DeleteErrorInfoType : ApplicationErrorInfoType {
          public MOSetType        relatedObjects;
          public AttributeSetType attributeList;
};

valuetype PackageErrorInfoType : CreateErrorInfoType {
          public StringSetType    packages;
};

exception ApplicationError { ApplicationErrorInfoType info; };

exception CreateError { CreateErrorInfoType info; };

exception DeleteError { DeleteErrorInfoType info; };
```

1.1.1 The ApplicationError Exception

An *ApplicationError* exception is raised when an operation cannot be completed due to some application-level condition at the managed system. Information returned with the exception includes an identifier for a specific condition, and a string with additional details or an explanation.

A few identifiers for specific error conditions are defined by the framework. These should be used whenever possible. Information models, though, may define additional error condition codes, or create their own exceptions.

The data returned with the application error exception is a value type, which means that it may be extended. That is, for a certain error condition codes, the actual data type returned might be an extension of the base application error info type. Because the error code is in the base type, the client code can examine it, and if its value is one that is passed back in a sub-class, the client can narrow (cast) the value type and access the additional information.

The *ApplicationError* exception shall be included in the *raises* clause of every managed object and managed object factory operation. A few error code values for the application

error exception have been defined for the framework. Each is discussed in sections below.

5.5.1.1 invalidParameter

An application error exception with an error code of *invalidParameter* is raised when the value of some operation parameter is not valid for the operation requested. The name of the bad parameter is returned in the details field.

5.5.1.2 resourceLimit

An application error exception with an error code of *resourceLimit* is raised when an operation cannot be completed due to some transient error on the managed system, such as lack of memory. A string containing an explanation is returned in the details field.

5.5.1.3 downstreamError

An application error exception with an error code of *downstreamError* is raised when an operation cannot be completed due to an error downstream from the managed system. An example of this is when an operation can't be completed because an EMS cannot communicate with an NE.

1.1.2 The CreateError *Exception*

The *CreateError* exception is raised when an error occurs on a factory create operation. It should be included in the *raises* clause of every managed object factory create operation.

The data returned with this exception extends that of a general *ApplicationError,* and adds a list of related object, and the attribute values the object would have had if it had been created. The specific error codes defined for this exception by this framework are presented below. Implementations should use these whenever possible. Information models may add new values, or define new exceptions for special cases.

5.5.1.4 invalidNameBinding

A create error exception with an error code equal to *invalidNameBinding* is raised when the name binding included in the create operation does not support the creation of the object in this situation.

5.5.1.5 duplicateName

A create error exception with an error code equal to *duplicateName* is raised when the name included in the create operation is a duplicate.

5.5.1.6 unsupportedPackages

A create error exception with an error code equal to *unsupportedPackages* is raised when one or more of the requested packages is not supported by the implementation. Note that when this error code is used, the returned data structure is actually a *PackagesErrorInfoType* structure, which extends the *CreateErrorInfoType* structure. The

PackagesErrorInfoType structure includes a list of packages, which in this case will be the unsupported packages.

5.5.1.7 incompatiblePackages

A create error exception with an error code equal to *incompatiblePackages* is raised when some of the requested packages are not compatible with each other or the resource for which the object is being created. Note that when this error code is used, the returned data structure is actually a *PackagesErrorInfoType* structure, which extends the *CreateErrorInfoType* structure. The *PackagesErrorInfoType* structure includes a list of packages, which in this case will be the incompatible packages.

1.1.3 The DeleteError *Exception*

The *DeleteError* exception is raised when an error occurs on a delete operation. It is included in the *raises* clause of the destroy operation on the base *ManagedObject* interface, which is then inherited by every managed object.

The data returned with this exception extends that of a general *ApplicationError*, and adds a list of related object, and the attribute values the object had when the delete attempt was made. The specific error codes defined for this exception by this framework are presented below. Implementation should use these whenever possible. Information models may add new values, or define new exceptions for special cases.

5.5.1.8 notDeletable

A delete error exception with the constant value equal to *notDeletable* is raised when an attempt is made to invoke the destroy() operation on a managed object that should not be destroyed according to its delete policy. (Note that the destroy() managed object operation is defined for use by other parts of the framework. Managing systems that invoke it directly run the risk of corrupting data on the managed system.)

Also, the Terminator Service will raise this exception when a client tries to delete an object with a delete policy of *notDeletable*.

5.5.1.9 containsObjects

A delete error exception with the constant value equal to *containsObjects* is raised when an attempt is made to delete a managed object that has subordinates and a delete policy of *deleteOnlyIfNoContainedObjects*.

Managed objects are not responsible for detecting this condition, but the Terminator Service is.

5.6 Macro Definitions

Following the interfaces in Annex A are the definitions of some macros. These macros simply provide shorthand notations for identifying which notifications are supported by which objects. Due to the limited capability of CORBA IDL to accept information like this, it was felt these macros would be useful.

The *MandatoryNotification* macro identifies notifications that must be supported by an object, and the *ConditionalNotification* macro identifies notifications that must be emitted by a managed object if it supports a particular package. Both macros take arguments identifying the name of an operation (recall that operations are used to convey notifications) and the scoped name of the interface on which the operation is defined. The *ConditionalNotification* macro also accepts a third parameter, the name of the package to which the notification belongs.

The notification macros expand into nothing. Unfortunately, IDL is simply too limited to provide a way to capture this information. Comments could be generated, but they are just immediately discarded by the compiler. Formatted comments, like those used to generate HTML, unfortunately can't be used because they require some IDL construct to which they are associated. It was hoped that the upcoming CORBA Component Model would provide a solution, but implementations won't be available in time for these guidelines. In the future it may be possible to modify the macros to generate IDL consistent with the CORBA Component Model. For now, though, the information about which notifications are emitted by which object classes is captured by these macros.

5.7 The Constant Definitions

Interface specifications always contain a number of constants whose values are agreed upon by everyone to mean the same thing. For example, everyone agrees a "1" in a certain field means a loss of signal, a "2" means a loss of frame, etc. X.721 is no exception and defines a number of constants. These are reproduced in IDL form in Annex B. For details on the mechanism used to convey pre-defined constants, see Section 6.11.

6 Information Modeling Guidelines

This section presents guidelines for developing CORBA-based TMN information models. Guidelines for the translation of existing models specified in GDMO are provided in the next section.

6.1 Modules

IDL Modules are used to group together interfaces, type definitions, exceptions, and other IDL constructs. Modules also provide name-space delineation; identifiers within a module must be unique but may be re-used in other modules. In almost all cases, a module shall be used to group the constructs used to specify an information model. Modules may be nested within other modules, and modules may span multiple files. The IDL specified in these guidelines is contained within a single module, named "itut_x780". For example:

```
module itut_x780 {
…
};  // end of module itut_x780
```

This module has sub-modules for constant definitions.

6.2 Interfaces

Each *entity* accessible via the CORBA network management interface shall have an IDL interface defined for it. Interfaces group together a set of attributes and methods that can be thought of as being provided by a single software object. Interfaces may inherit capabilities from other interfaces and interfaces defined to model an *entity* must inherit (directly or indirectly) from the interface named *ManagedObject* defined in this document. For example:

```
interface Equipment : ManagedObject {
…
};  // end of interface Equipment
```

Such interfaces are referred to as "managed object interfaces." The objects that support these interfaces are "managed objects." Because the *ManagedObject* interface defined in this document has a set of capabilities that are inherited by all managed object interfaces, each managed object must implement a base set of functions to exist in the TMN CORBA framework.

One issue information modelers may face is CORBA's limited support for multiple inheritance. An interface may inherit an operation or attribute from multiple super classes only if they in turn inherited them from the same super class. This is known as "diamond" inheritance, and is depicted in the figure below.

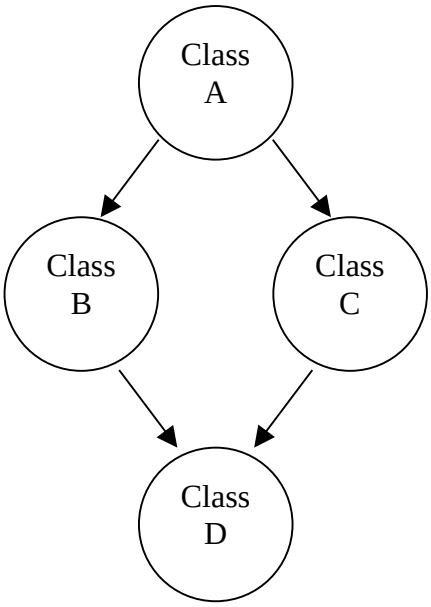

Figure 3. Diamond Inheritance

If an information modeler is faced with having to inherit the same capability from two different classes that do not share a common super class, the modeler may have to modify the classes and create a virtual super class from which the capability can be inherited. For example, creating "D" from "B" and "C" above but where "A" does not exist, the

modeler may have to modify the super classes by creating a new virtual class ("A") with the common capability that is then inherited by "B" and "C."

6.3 Attributes

Attributes are modeled within interfaces as operations used to access the attribute's value. The names of the operation, as well as the input and output types, indicate the name of the attribute as well as the type of operation. (CORBA IDL does support attributes in addition to operations, but at this time only operations are allowed to raise user-defined exceptions. As will be seen, user-defined exceptions are needed on attribute accesses. For this reason, operations are defined to access attributes rather than merely defining attributes. Future versions of CORBA plan to allow user-defined exceptions on attribute access, and these guidelines may change to take advantage of this.)

6.3.1 Readable Attributes

Managed objects should have an operation named "<attribute name>Get" on their interface for each readable attribute. The type returned by this operation reflects the type of the attribute. For example:

```
AdministrativeStateType administrativeStateGet()
        raises (ApplicationError);
```

Attributes that are settable but not readable, which is rare, should not have a read operation defined on the interface.

Attribute get operations that may return large amounts of data should define an iterator to enable the client system to control the return flow of information. For an example of the use of iterators, see ITU-T Q.816.

6.3.2 Settable Attributes

Managed object interfaces should have an operation named "<attribute name>Set" for each settable attribute. The operation return type should be *void* and the input parameter should reflect the type of the attribute. For example:

```
void administrativeStateSet (in AdministrativeStateType adminState)
        raises (ApplicationError);
```

Attributes that are not settable should not have such an operation on the interface.

6.3.3 Set-valued Attributes

Many managed object attributes may contain sets of values. In these cases, the operations defined above should still be supported (if the attribute is readable and/or writeable). Because CORBA does not explicitly define a complex type for sets, the input or return types for these operations will be CORBA sequences. Values returned for these attributes should not contain duplicate values, and the order of the values is unimportant. Also, it may be necessary to support the addition or removal of values to these attributes. These operations should be named "<attribute name>Add" and "<attribute name>Remove". The return types for these operations should be void and the input parameter to each should be a sequence reflecting the type of the attribute. For example:

```
void supportedByObjectsAdd (in ManagedObjectSetType objects)
        raises (ApplicationError);

void supportedByObjectsRemove (in ManagedObjectSetType objects)
        raises (ApplicationError);
```

6.3.4 Exceptions

Attribute access operations may also raise exceptions. The following exceptions are defined to be raised on attribute access operations:

1. *ApplicationError*. This exception shall be included in the *raises* clause of every managed object operation, including attribute access operations. It may be used to signal a number of conditions, such as a value that is out-of-range, a resource limitation on the managed system, etc.

2. Conditional Package Exceptions. If the attribute is part of a conditional package, the exception defined for that conditional package shall be included in the raises clause of the attribute access operations. It is raised when an attempt to access the attribute is made but the package to which it belongs is not supported by the instance. See more on Conditional Packages in Section 6.6 below.

In addition to these, an implementation may also raise any of the standard CORBA exceptions. Operations that raise exceptions shall not modify the value of the attribute. An example of an attribute access operation that raises an exception is:

```
void supportedByObjectsRemove (in ManagedObjectSetType objects)
        raises (ApplicationError);
```

6.3.5 Standard Attributes

Managed objects model resources, and often there is commonality among managed objects. This is sometimes represented using an inheritance relationship among object classes, but there may also be commonality between objects when no inheritance relationship exists. A good example of this is similar attributes. Many managed objects have similar attributes. To make the implementation of management interfaces easier, these guidelines define some standard data types that should be used for attributes whenever possible. That is, modelers should attempt to use these type definitions instead of defining new types. Also, the attribute name, and the names of the operations to access the operation should be used. In fact, when defining a new model, it is good practice to re-use attribute types and names from existing models whenever possible. The standard attributes defined are:

Data Type	Attribute Name	Access Method
AdministrativeStateType	administrativeState	administrativeStateGet()
AvailabilityStatusSetType	availabilityStatus	availabilityStatusGet()
BackedUpStatusType	backedUpStatus	backedUpStatusGet()
ControlStatusSetType	controlStatus	controlStatusGet()
SourceIndicatorType	creationSource*	creationSourceGet()

DeletePolicyType	deletePolicy*	deletePolicyGet()
ExternalTimeType	externalTime	externalTimeGet()
NameType	name*	nameGet()
ObjectClassType	objectClass*	objectClassGet()
OperationalStateType	operationalState	operationalStateGet()
StringSetType	packages*	packagesGet()
ProceduralStatusSetType	proceduralStatus	proceduralStatusGet()
StandbyStatusType	standbyStatus	standbyStatusGet()
UnknownStatusType	unknownStatus	unknownStatusGet()
UsageStateType	usageState	usageStateGet()

* These attributes are inherited by all managed objects.

Table 1. Standard Attributes

6.4 Actions

In addition to attributes, many managed objects will have *actions* – methods for purposes other than accessing an attribute. The parameters and return types for these operations are simply defined to meet the needs of the action. The name of the operation should reflect the purpose of the operation. The following exceptions have been defined to be raised on action operations:

1. *ApplicationError*. This exception shall be included in the *raises* clause of every managed object operation, including action operations. It may be used to signal a number of conditions, such as a parameter value that is out-of-range, a resource limitation on the managed system, etc.

2. Conditional Package Exceptions. If the action is part of a conditional package, the exception defined for that conditional package shall be included in the raises clause of the action operations. It is raised when an attempt to invoke the action is made but the package to which it belongs is not supported by the instance. See more on Conditional Packages in Section 6.6 below.

In addition to these, an implementation may also raise any of the standard CORBA exceptions. Other exceptions specific to the action may and should be defined for other error conditions. Alternatively, an information model may extend the error code points defined for the *ApplicationError* exception.

Actions that may return large amounts of data should define an iterator to enable the client system to control the return flow of information. For an example of the use of iterators, see ITU-T Q.816..

6.5 Notifications

Most managed objects are expected to emit notifications under certain conditions. In the TMN CORBA framework, notifications are conveyed by method invocations from a managed object back to a managing system, with the help of the Notification Service.

Thus, the notification operation is actually defined for the managing system's CORBA interface, not the managed object's interface. These guidelines define a number of standard notifications, but if a new notification must be defined it should be defined as an operation on an interface named "Notifications" within the information model's module. The name of the operation should be the name of the notification. The parameters to the operation should reflect the data to be reported in the notification. The notification operation's return type must be void, and it must have only "in" parameters. Note that the "oneway" keyword preceding the notification operation definition should not be used. Notifications following these guidelines are confirmed. That is, when a managed object sends a notification to a channel, the receipt of that notification will be confirmed back to the managed object by the channel. Likewise, as the channel sends the notification to each recipient, a confirmation is received by the channel. Quality of Service guarantees, specified in ITU-T Recommendation Q.816 define the reliability of the channel itself. Thus, the delivery of notifications to recipients can be guaranteed.

A means of documenting which managed objects emit which notifications is also needed. Rather than simply noting this through comments in an IDL file, a macro statement is used. Actually, these guidelines define two macros, one for use when the notification is mandatory and the other when the notification is part of a conditional package. The macros are intended to be used within a managed object interface and are defined as follows:

```
MANDATORY_NOTIFICATION(<interface name>,
        <notification operation name>);

CONDITIONAL_NOTIFICATION(<interface name>,
        <notification operation name>, <package name>);
```

For example:

```
interface Equipment : ManagedObject {
…
MANDATORY_NOTIFICATION(itut_x780::Notifications, objectCreation);
CONDITIONAL_NOTIFICATION(itut_x780::Notifications,
        equipmentAlarm, equipmentAlarmPackage);
…
};  // end of Equipment interface
```

The package name used in the conditional notification macro is the same as used elsewhere. See Section 6.6 on packages for details. The macros actually expand into nothing because there really isn't a good alternative in CORBA IDL. Thus, the macros are for documentation purposes and don't actually result in code generation. An item for further study is modifying the macros to generate IDL that would identify the notifications supported by an object. The release of the CORBA Component Model specification provides an opportunity to do this in a manner consistent with that model. Only one notification may be listed in each macro. This is to make the possible future modification of the macros simpler.

6.6 Conditional Packages

These information modeling guidelines support the notion that not all capabilities defined for a class of managed objects need to be supported by all instances. In fact, groups of

capabilities can be defined so that either all or none of the capabilities are supported. These groups of capabilities are referred to as *packages*. The choices for representing packages in IDL are limited. Defining a separate interface for each package would result in too many interfaces, so instead the approach described here is used.

Each operation that is part of a conditional package may raise an exception defined for the package. The name of the exception shall be NO<package_name>. For example:

```
exception NOadministrativeStatePackage {};
…
AdmininstrativeStateType administrativeStateGet()
        raises (NOadministrativeStatePackage);
```

Notifications that are emitted as part of a conditional package are denoted with the *CONDITIONAL_NOTIFICATION* statement as described above.

Rules concerning when the capabilities included in a package should be supported and when they shouldn't are placed in comments related to the managed object interface. An operation may be included in more than one conditional package by listing multiple NO<package name> exceptions in its *raises* clause. An exception will be raised only if none of the packages are present, and then any of the package exceptions may be raised. If an operation is mandatory, it must list no package exceptions in its *raises* clause. A notification may list multiple packages in the *CONDITIONAL_NOTIFICATION* macro.

6.7 Behavior

CORBA IDL lacks a formal means of capturing object behavior. In the future it is possible that information models will be documented with UML and will include use cases and object interaction diagrams. IDL, however, is limited to comments. Therefore, when necessary or helpful, comments must be used to describe object behavior.

The IDL in this document contains a number of comments. They are formatted to be parsed by compilers used to convert IDL to HTML for easier reading. A formatted comment begins with /** and ends with */ and is associated with the next IDL construct. HTML formatting tags are allowed with these comments, as are certain keywords (preceded by a '@' symbol) that are converted by the IDL-to-HTML compilers into additional formatting. While viewing IDL with an HTML browser is convenient, note that the use of the macros described above is impacted by this. Because macro expansion is performed as a part of the conversion to HTML, the pre-expanded macro information will be lost. Thus, the macros used to identify the notifications supported by each managed object will have been expanded.

6.8 Name Binding Information

Containment is a very important relationship in network management. In the TMN CORBA-based framework, containment is represented through names. This, unfortunately, places no restrictions on the containment relationships that could possibly exist. There is nothing to prevent, for example, a network object from being contained by a connection object. Clearly, some means of restricting the possible containment

relationships to only those that are sensible is desirable. These restrictions, however, must be extensible under control of the information modeler.

To meet these needs, these guidelines require that IDL modules specifying CORBA-based TMN information models also contain information defining the possible containment relationships among the managed object classes. This containment relationship information is referred to as *managed object name binding information*. (Unfortunately, this may be easy to confuse with the name binding information stored in the CORBA Naming Service. The two are not the same.)

Managed object name binding information is represented in CORBA IDL using the following conventions:

1. Each information model IDL module shall contain a sub-module named "NameBindings" for managed object name binding information.
2. Within this name binding module, sub-modules shall be defined for each allowed containment relationship.
3. Each name binding sub-module shall assign values to these 7 constants;

```
const string            superiorClass
const boolean           superiorSubclassesAllowed
const string            subordinateClass
const boolean           subordinateSubclassesAllowed
const boolean           managersMayCreate
const DeletePolicyType  deletePolicy
const string            kind
```

The *superiorClass* constant contains the scoped class name of the superior (containing) object. If an object may be the "top-most" object on a managed system, that is, if it may be contained directly under a local root naming context, the *superiorClass* name binding value shall be an empty string. The *superiorSubclassesAllowed* constant is a Boolean field that will have a value of *true* if subclasses of the superior class type are acceptable using this name binding. The *subordinateClass* constant contains the scoped class name of the subordinate object (the object to be created). The *subordinateSubclassesAllowed* constant indicates if subclasses of the subordinate object may be created using this name binding. The *managersMayCreate* flag indicates if object creation is supported across the management interface using this name binding. The value of setting this flag to *false* is that it enables all containment relationship information to be documented in IDL, even if the subordinate object is only created by the managed system. The *deletePolicy* constant contains the value that will be assigned to the managed object's *deletePolicy* attribute when it is created. The *kind* constant contains the value that will be assigned to the *kind* field in the CORBA Name Binding for the object when it is created.

The value chosen for the *kind* field in a name binding will typically be the unscoped subordinate class name. (Unscoped class names will typically be used to reduce the length of names.) The main purpose of the *kind* field is to segment the naming space to keep naming collisions from occurring. Name binding modules for new versions

of existing interfaces might reuse the *kind* values used for the older interfaces. For example, name binding modules for Equipment and EquipmentR1 interfaces might both use the value “Equipment”. Otherwise, though, it will probably be safest to use a unique value for each class of interface.

4. The name of a name binding sub-module shall be <subordinateClass>_<superiorClass>, where <subordinateClass> is the value assigned to the *subordinateClass* constant and <superiorClass> is the value assigned to the *superiorClass* constant in the module. If two name binding modules in the same parent module share the same *superiorClass* and *subordinateClass* values but differ in other values, the name of one of the modules shall be appended with a word denoting a difference between the two. For example: “Equipment_Equipment” and “Equipment_Equipment_NotDeleteabe”.

Some example managed object name bindings:

```
module itut_m3120 {
…
    /** The following module contains name binding information */

    module NameBindings {

        /** This name binding module allows Equipment objects to be
        created under Managed Element objects.
        */

        module Equipment_ManagedElement {
                const string    superiorClass = “itut_m3120::ManagedElement”;
                const boolean   superiorSubclassesAllowed = TRUE;
                const string    subordinateClass = “itut_m3120::Equipment”;
                const boolean   subordinateSubclassesAllowed = TRUE;
                const boolean   managersMayCreate = TRUE;
                const DeletePolicyType deletePolicy =
                                itut_x780::DeleteOnlyIfNoContainedObjects;
                const string    kind = “Equipment”;
        }; // end of Equipment_ManagedElement name binding module

        /** This name binding module allows Equipment objects to be
        created under other Equipment objects.
        */

        module Equipment_Equipment {
                const string    superiorClass = “itut_m3120::Equipment”;
                const boolean   superiorSubclassesAllowed = TRUE;
                const string    subordinateClass = “itut_m3120::Equipment”;
                const boolean   subordinateSubclassesAllowed = TRUE;
                const boolean   managersMayCreate = TRUE;
                const DeletePolicyType deletePolicy =
                                itut_x780::DeleteOnlyIfNoContainedObjects;
                const string    kind = “Equipment”;
        }; // end of Equipment_Equipment name binding module

    }; end of name binding module
}; end of itut_m3120 module
```

Note that the *deletePolicy* constant is of an enumerated type and according to CORBA IDL constant definition rules, if this type is defined in another module, the value assigned to the constant must be scoped to that module. The *DeletePolicyType* is defined in module itut_x780, and the example IDL module is itut_m3120. Therefore, the *DeleteOnlyIfNoContained* value must be scoped by preceding it with the string "itut_x780::". The type itself, *DeletePolicyType*, must also be scoped. This can be done with a *typedef* statement at the beginning of the module.

6.9 Factories

The TMN CORBA-based framework defines a service for deleting objects, but objects are created with class-specific *factories*. Factories are objects with interfaces distinct from the objects they are used to create, but usually related. Each class of managed objects will also have a factory class. This is done so that the factory create operations may be strongly typed and specific to the class of objects they create. The result of this is that the IDL modules defining managed object interfaces will also contain interfaces for the factories used to create the objects. The name of the factory IDL interface shall be "<Managed Object Class Name>Factory".

This document defines a base managed object factory interface from which each factory interface must inherit. Factories do not follow the same inheritance hierarchy as the objects they create. Factories simply inherit from the *ManagedObjectFactory* interface. An example of a factory interface definition is:

```
interface EquipmentFactory : ManagedObjectFactory {
…
};  // end of EquipmentFactory interface
```

Because factories cannot create subclasses of objects, new factories must be defined for each subclass.

Every instantiable class shall have a factory defined for it, even if at the time no name binding modules allowing managers to create instances are defined. This is to allow for the future definition of name binding modules that do enable managers to create instances.

6.9.1 Create Operations

Each factory interface shall define a single operation for clients to use to create objects. The name of this operation shall be "create" and it shall return a reference to the type of object created by the factory. The first four parameters to every create operation are always the same. After these come parameters for each writeable or set-by-create attribute defined for the managed object. (A set-by-create attribute is one for which the object has no "set" operation, but for which a value is specified on the create operation.) The names of these parameters are the same as the name of the attribute. (This is the name of an attribute accessor operation minus the ending "Get" or "Set".) Each create operation also has to accept parameters to set the values of any writeable or set-by-create attributes of all super-classes of the object created by the factory. Here is an example of a create operation for an equipment factory:

```
Equipment create(
        in NameBindingType nameBinding, // module name containing NB info.
        in ManagedObject superiorObject,// Reference to containing object.
        inout string name,              // In/out, may be null if auto-create.
        in StringSetType packages,      // List of packages requested.
        …                               // Writeable and set-by-create values
                                        // for Equipment superclass attributes.
        …                               // Writeable and set-by-create values
                                        // for Equipment attributes.
);
```

6.9.1.1 Name Binding

The name binding parameter conveys the name of a module containing managed object name binding information, as described in the Section 6.8. An example value might be "itut_m3120::NameBindings::Equipment_Equipment". Given this, the factory can check to see if the value is a valid name binding identifier. (A factory might either be "hard-coded" with name binding information available when the system is compiled, or it might access the information in the CORBA Interface Repository at run-time.) If the name binding information can not be found, the factory shall raise an *invalidParameter ApplicationError* exception, returning "nameBinding" as an argument. (This is an *ApplicationError* exception with the *error* code set to *invalidParameter* and the *details* string set to "nameBinding".) If the name binding information can be found, but is incomplete, the factory shall raise an *invalidNameBinding CreateError* exception.

The factory must also check to see if the subordinate class type specified in the name binding module matches the type of objects it creates. If it is doesn't, the factory can then check to see if the type of objects it creates is a subclass of the subordinate class constant value. If it is, and if the *subordinateSubclassesAllowed* constant is *true*, it can proceed to create the object. If not, it would reject the request by raising an *invalidNameBinding CreateError* exception.

Finally, if the *managersMayCreate* constant in the name binding module is *false*, the factory would also reject the request by raising an *invalidNameBinding CreateError* exception. (Factories may have a second create operation for internal use by the managed system that does not check this value and that is not exposed across the management interface.) The inclusion of name binding modules with *managersMayCreate* values set to *false* enables capturing all of the containment information in IDL, as is possible with GDMO, even if the objects are created only by the managed system itself.

The other information in the name binding module will be used by the factory when it creates the object and its CORBA naming service name binding. The *deletePolicy* constant will be assigned to the new managed object's attribute of the same name. The *kind* constant value will be used when the factory creates the managed object's name binding in the CORBA naming service.

6.9.1.2 Superior Object

The second parameter in the create operation is a reference to the superior object, under which the new object is to be created. Using standard CORBA capabilities, the factory

shall examine the class of the superior object to determine if it matches the type specified in the *superiorClass* constant defined in the name binding module. If it doesn't, the factory must next check to see if the supplied reference is of a subclass of the type specified in the *superiorClass* constant. If it is, and if the *superiorSubclassesAllowed* constant in the name binding is *true*, the factory may proceed to create the object. If not, the factory must reject the request by raising an *invalidNameBinding CreateError* exception, returning "superiorObject" in the details.

If the *superiorClass* constant in the name binding module is an empty string, then objects of the subordinate class may be created with no superior object (parent), and their name is bound directly to a local root naming context. Usually, these objects will be created by the managed system, but in these cases the superior object reference would be null.

6.9.1.3 Name

The third parameter is the name to be assigned to the new object. This string will become the *ID* field of the CORBA Name Binding created in the CORBA naming service for the new object. This will be relative to the superior object's name. If the parameter is *inout*, it indicates that the factory must support auto-naming. In this case, a client may submit a null string for the name, and the factory will choose a suitable string and return the chosen value. If instead the client submits a string, the factory shall use this value instead (and return it as the *out* value). If the parameter is *in* only, auto-naming is not supported and the client must supply a name. If it doesn't, the factory shall raise a *badName CreateError* exception. The factory raises a *duplicateName CreateError* exception if the supplied name is a duplicate. (This means both the *ID* and *kind* fields match an existing object contained by the superior object.)

6.9.1.4 Packages

The packages attribute is important. It tells the factory not only which packages an instance must support, but which parameter values on the create operation it must ignore. Because they are strongly-typed, create methods include a parameter for each writeable or set-by-create attribute of an object, even if an attribute is part of a conditional package. The factory must ignore the values for any attribute in packages that are not requested by the client, even if the factory instantiates the object with the package anyway. (If the factory instantiates an object with a package not requested by the client, the factory must choose the initial values.) This frees the client from having to supply values for attributes in packages it does not want. Instead, the client can submit any value. For efficiency, the values submitted for attributes in packages not requested by the client should be short.

If the client supplies an invalid package name in the packages parameter, the factory shall raise an *unsupportedPackage CreateError* exception and return the name of the package as the argument. An *incompatiblePackages CreateError* exception may also be raised if the client requests the creation of an instance but specifies packages that may not coexist in the same instance.

6.9.1.5 Superclass Parameters

Following these first four parameters will be parameters for each of the writeable and set-by-create attributes for any superclasses of the type of objects created by the factory.

6.9.1.6 Object Class Parameters

Finally, following the superclass parameters will parameters for each of the writeable and set-by-create attributes for the managed object class created by the factory.

6.9.2 Factory Finder

To ease the task of finding a factory, ITU-T Recommendation Q.816 defines a factory finder interface. (The factory finder is a common design pattern in CORBA applications.) This enables a client to easily find a factory by interacting with a well-know broker with knowledge of all the factories present on a managed system.

6.10 Managed Object Class Value Types

Each managed object class compliant with these guidelines inherits an operation from the base Managed Object class that returns all or some subset of the object's attributes in a single *valuetype*. (CORBA 2.3 introduces the concept of value types, objects that are passed by value instead of by reference.) Not only must the managed object implementation support this feature, the IDL describing the managed object must include a value type with public attributes for each of the attributes supported by the managed object. These guidelines define a base *ManagedObjectValueType*, and the value types defined for managed objects must ultimately derive from this base value type. The value types defined for managed objects should usually follow the inheritance pattern of the managed objects interface, but since CORBA's value types support only single inheritance, this won't always be possible. This is not a serious limitation, though. It simply means that the value types defined for interfaces using multiple inheritance will have to singly inherit from one of the superior value types, and the other attributes will have to be added and maintained by hand.

As an example, assume the Equipment managed object interface inherits directly from the base *ManagedObject* class, and has, among others, an attribute access function called *userLabelGet* that returns a type *UserLabelType*. The IDL describing the value type for the Equipment managed object would look like this:

```
valuetype EquipmentValueType : ManagedObjectValueType {
   public UserLabelType    userLabel;
   …                              // other attributes
};
```

The name of the value type is the name of the interface with "ValueType" appended. Notice, too, that the name of the public attribute in the value type is the name of the method on the managed object interface used to access the attribute without the appended "Get." This convention should be followed for all attributes in value types. The type of the attribute is the same as the type returned by the attribute access function.

Code on the client side wishing to retrieve the attribute values for an equipment object might look something like this:

```
ManagedObjectValueType   moValue;
EquipmentValueType       eqValue;
Equipment                eq;

eq = …   // code that sets eq to a CORBA proxy representing an
         // equipment object.

moValue = eq.getAttributes();
eqValue = (EquipmentValueType) moValue;  // cast return to proper type
System.out.println("User Label = " + eqValue.userLabel); // print label
```

When the IDL is compiled into an object-oriented programming language, both the interfaces (in this case, Equipment) and the value types (*ManagedObjectValueType* and *EquipmentValueType*) will be translated into classes. For the interfaces, the classes are actually proxies. When methods are invoked upon them they make use of the ORB to send the request back to the server. The classes translated from value types, however, are not proxies. They are simply local objects.

When the client invokes the call on the equipment proxy to get attributes, the response from the server will be an *EquipmentValueType*. When the ORB receives this, it will create a local instance of an *EquipmentValueType* object with the attribute values received from the server. Because the return type to the *attributesGet*() method, defined on the base Managed Object interface, is *ManagedObjectValueType*, the reference to the *EquipmentValueType* instance is passed back as a reference of type *ManagedObjectValueType*. This works because *EquipmentValueType* is derived from *ManagedObjectValueType*. In order to access attributes that are specific to *EquipmentValueType*, though, the client must narrow the reference by casting it to type *EquipmentValueType*.

While the behind-the-scenes processing being done by the ORB is a bit complicated, the alternative would be to use lists of CORBA *any* types to hold the attribute values. This approach, though, would require even more processing. The *any* types would be much more complicated for the programmer, too. As shown in the example above, using the value types is actually quite simple.

6.11 Constants

Network management systems require the ability to exchange information with previously agreed-upon meanings. For example, a state change notification with a probable cause of "1" might mean it was likely caused by a loss of signal, while a "2" means a loss of frame, etc. It's simple enough to define an enumeration or set of integer values to be passed across an interface in some field, but it is a little trickier to make this mechanism extensible by multiple groups, likely acting in parallel. The mechanism used by these guidelines for this is referred to as the "Universal Identifier (UID)."

A UID is a data structure with two fields. The first is a string meant to contain the scoped name of an IDL module containing the constants defined for some field. The second is a

short (16 bit) signed integer containing the value. For example, to send a value of "loss of signal" in a probable cause field, a system would construct a UID structure with a *moduleName* string equal to "itut_x780::ProbableCauseConst" and an integer value equal to 29. (Annex B contains the constants defined for these guidelines. In it is a module named "ProbableCauseConst" which contains a constant named *lossOfSignal* with a value of 29.)

Note that this is the only format for constant values used within this framework. There are no "local" values used.

These conventions shall be followed when defining constants for an information model:

1. Constant values shall be defined in separate modules, one for each set of constants defined for a particular field. These sub-modules shall be contained within the top-level module that contains the other constructs defined for the information model.
2. The name of the module shall be the name of the field appended with "Const". For example, values for the *probableCause* field (defined as type UIDType) are contained within a module named "ProbableCauseConst".
3. The constants defined within the sub-module must be of type *const short*. For example:

```
const short lossOfSignal = 29;
```

4. Constants may be kept in a separate file, to reduce the length and complexity of the main IDL file. Even if the constants are in a separate file, the sub-modules shall be within an IDL module statement with the same name as the module in the main file. The main file shall have a pre-compiler *include* statement at the top of the file to include the constants in any compilation run.
5. The sub-module shall also contain a string constant named "moduleName" that contains the scoped name for that module. For example:

```
module itut_x780 {
 …
 module ProbableCauseConst {
         const string moduleName = "itut_x780::ProbableCauseConst";
         …
 };  // end of module ProbableCauseConst
 …
};  // end of module itut_x780
```

   This is really just a courtesy to allow programmers to refer to the module's name by a constant rather than hard-coding module string names.

Note that other information models may extend the values for probable cause. There could, for example, be a module "itut_m3120::ProbableCauseConst" with additional values for the probable cause field. These modules can even re-use the value 29. The UID will still be unique because the module names will differ.

6.12 Registration

CORBA IDL requires that all the identifiers within a module must be unique. This means that as long as a module name is unique, all of its contents will be uniquely named. CORBA IDL also defines an IDL compiler *pragma* statement that may be used to define a unique prefix to the module identifiers when they are registered in the CORBA interface repository, a central directory of interface information used by CORBA ORBs. This framework requires that IDL documents contain a pragma prefix statement using the organization's Internet domain name as a prefix for the contained modules.

This eliminates the need to register each individual construct.

6.13 Versioning of CORBA/IDL Specifications

When using CORBA, a management interface is specified as one or more object interfaces defined using IDL. Inevitably, management interfaces change. Adding a new CORBA object interface to a management interface is straightforward. The new CORBA interface simply needs to be defined in IDL, and added to the specification identifying the object interfaces to be supported on that particular management interface.

Updating an existing CORBA object interface, however, is a little trickier. These guidelines place a priority on backward compatibility. Therefore, the following rules apply to extending an existing managed object interface. Note that these rules apply only to extensions being made to a base class that do not result in changing the business purpose of the object. That is, the new class models the same resource as the old class, it simply has some additional capabilities.

1. The name of the new object interface shall be the same as the existing interface with the letter "R" and a numeral appended, starting with "1." Subsequent extensions will increment the numeral. So, extending an interface for "Equipment" managed objects would result in an interface named "EquipmentR1."
2. The new interface shall be defined within the same module name as the existing interface. (CORBA modules are really just name spaces, and may be spread across multiple files.)
3. The new interface shall inherit from the existing interface.
4. Capabilities inherited from the existing interface cannot be removed or modified in the new interface. If an operation definition must be modified, a new operation must be defined. The name of the new operation shall be the same as the existing operation with the letter "R" and a numeral appended, starting with "1." Subsequent extensions will increment the numeral.
5. The value for the *kind* field used in name bindings will continue to be determined by a constant in the name binding modules referenced when the object is created. Any name bindings valid for the existing interface shall be valid for the new interface. That is, a name binding module for an Equipment object shall also be valid for an EquipmentR1 object, even if the module's value for *subordinateSubclassesAllowed* is false.

6. References to the new interfaces should be of the most specific type. (If they aren't, the new capabilities can't be accessed.) Also, the value of the *objectClass* attribute reported by an object of the new class should be the most specific type. CORBA provides some means for determining the actual class of a reference based on information contained in the IOR.

For example, consider the following object interface:

```
interface Foo {
        void action(in int A, in int B);
}
```

The action might be extended like this:

```
interface FooR1: Foo {
        void actionR1(in int A, in int B, in int C);
}
```

The old action would still be a valid operation.

A similar approach, appending the name with "R" and an incremented number, shall be used when other existing IDL definitions are revised, including constant definitions, type definitions, and valuetype definitions.

7 GDMO Translation

This section provides guidelines for creating IDL information models from existing information models described using GDMO. The sections below describe how each of the GDMO templates is to be translated to CORBA IDL.

7.1 Managed Object Classes

Each Managed Object Class in a GDMO specification shall be translated into a managed object interface. Translations of Managed Object Classes derived from the GDMO Top class shall inherit from the *ManagedObject* CORBA IDL interface. Translations of classes not derived directly from Top shall inherit from the translation of whatever class they are derived from. All managed object interfaces must inherit directly or indirectly from the *ManagedObject* interface. Multiple inheritance is allowed subject to the rules of CORBA IDL. Note, however, that these rules do differ from CMIP. In particular, CORBA does not allow an attribute or operation to be inherited from multiple sources unless they in turn inherited it from the same common source. If a multiple-inheritance translation from CMIP does not meet the CORBA rules, the translator will have to choose to inherit from one superclass and manually add the other capabilities from the other class. Another option is to modify the conflicting superclasses so that they inherit the conflicting capability from a common source. This, of course, would require re-definition of these superclasses.

The inability to inherit from a potential superclass also means manual work may be required if the potential superclass or any of its super classes is modified. A more serious issue is that CORBA polymorphism is based on inheritance. If the subclass does not

inherit from a class, it can not be polymorphic to it. Unfortunately this is a limitation of CORBA, not these guidelines.

Attributes, actions, and notifications in mandatory and conditional packages are translated into operations on the interface according to the guidelines below. A comment preceding the interface should describe the conditions under which the capabilities of a conditional package are to be supported by an instance, based on the PRESENT IF clause for that package. Note that CORBA does not allow the re-definition of a capability present in a superclass. Therefore, if a capability is defined as conditional in a superclass, it cannot be redefined as mandatory in a subclass. (As described above, capabilities are denoted conditional when they raise a *NO<package_name>* exception. This exception cannot be removed in a subclass. The best alternative will be a comment indicating that the subclass should not raise the exception. Another alternative would be to forsake inheritance and manually add the capability, making it mandatory while doing so. This could lead to problems with polymorphism, however, and manual updating.)

Registration of individual interfaces is not required.

7.2 Packages

Unfortunately, IDL does not provide a means of defining packages in one place other than by translating a package into an interface. This, though, would result in a large number of extra interfaces and increase the complexity of the CORBA interface. Instead, these guidelines include the concept of conditional support for groups of capabilities.

As described above, whenever a GDMO package is included in a Managed Object Class, the translation of that class to an IDL interface includes a translation of each of the templates in the package.

GDMO attributes that are part of a conditional package shall be translated into access operations each with a *raises* clause that includes the exception defined for that package. GDMO actions that are part of a conditional package shall be translated into an operation that also has a *raises* clause that includes the exception defined for that package. GDMO notifications that are part of a conditional package shall be translated into a *CONDITIONAL_NOTIFICATION* macro statement.

The *present if* clause in the GDMO object's conditional package statement shall be translated to a comment preceding the IDL translation of the object.

Translations from CMIP can also encounter problems when the same capability is included in different conditional packages. These rules shall be followed:

1. If the capability is mandatory in one source and conditional in another, it must be mandatory in the translated class.
2. If the capability is part of multiple conditional packages, the translated operation will include an exception for each package. An exception will be raised only if none of the packages is present, and then any one of the exceptions may be raised.

3. If the same conditional package is included from multiple super classes, the condition under which the packages is included in the new class is a logical “OR” of the conditions in the super classes.
4. Notifications that are part of multiple packages are translated into just a single macro statement. If any of the packages are mandatory, the *MANDATORY_NOTIFICATION* macro statement is used. Otherwise, the *CONDITIONAL_NOTIFICATION* macro statement is used, and all of the package exceptions are listed.

If a GDMO template occurs in multiple conditional packages included in a single object, the modeler may want to consider making the capability mandatory or defining a new conditional packages for the capability.

Note that using exceptions to represent packages only supports conditional packages. If multiple mandatory packages are present in a GDMO class, they won’t be distinguishable on the translated interface

Behavior statements accompanying a package definition shall be translated to comments in the interface definitions of the IDL objects translated from the GDMO objects that include the package.

Registration of packages is not required.

7.3 Attributes

As described above, GDMO managed object classes list the packages that are to be included in the class definition. The package then lists the attributes, actions, and notifications that make up that package. When translating a managed object class, each template in the included packages will be translated to an operation on the managed object interface, and most of these will include attribute definitions.

Attributes that support *GET* capabilities shall have an <Attribute Name>Get operation defined for them. The return type for the operation shall be a translation of the attribute’s ASN.1 syntax.

Attributes that support *REPLACE* capabilities shall have an <Attribute Name>Set operation defined for them. The input parameter type for the operation shall be a translation of the attribute’s ASN.1 syntax.

Attributes that support *ADD* capabilities shall have an <Attribute Name>Add operation defined for them. Attributes that support *REMOVE* capabilities shall have an <Attribute Name>Remove operation defined for them. The input parameter type for these operations shall be IDL sequences translated from the attribute’s ASN.1 syntax.

Attributes that support the *set-by-create* capability shall accept an initial value for the attribute on factory create methods but shall not have a *SET* operation. (The factory

create method will also accept values for attributes that are settable, but not attributes that are merely readable.)

Default values are defined as constants within an interface. The identifier of the constant shall be <AttributeName>Default. The interface may also have an operation for setting the attribute to its default, or the client can just use the *SET* operation with the default constant. The set-to-default operation shall be named <AttributeName>SetDefault and it shall accept no parameters and return *void*. CORBA IDL allows constants to be defined for only base types and enumerated types, so if the attribute's type is complex, no default can be defined for it. In these cases, a set-to-default operation must be defined and a comment associated with the set-to-default operation shall describe the default value.

A few other attribute-related GDMO capabilities cannot be re-created with IDL. GDMO attributes with a *DERIVED-FROM* clause will have to have the capabilities of the other attribute manually added to the interface specification. (The syntax of the derived-from attribute will be used.) Matching rules are defined by the Multiple-Object Operation Service constraint language, which is part of the TMN CORBA services defined in ITU-T Rec. Q.816. These matching rules simply depend on the basic type of the attribute. There are no matching rules per attribute. Initial values, permitted values, and required values are not supported.

It will often make sense to define an IDL type for each attribute. Even if the attribute is a simple type, an IDL *typedef* statement may be used to define a type for it. A comment preceding the type definition for an attribute is the best place to put a translation of an attribute's behavior statement. Otherwise, the behavior statement may be translated to a comment preceding the attribute's access operation on the object interface.

The standard attributes defined by these guidelines shall be used whenever possible. See Section 6.3.5.

Registration of attributes is not required.

7.4 Attribute Groups

These guidelines do not support the concept of attribute groups. GDMO attribute groups have no equivalent translation.

7.5 Actions

Actions shall be translated to IDL operations. The input parameters, output parameters, and return type for the operation shall be translated from the action's input and output ASN.1 syntax. That is, the input syntax should be translated to IDL *in* parameters, while the output syntax is translated to a mix of *out* parameters and the return value. IDL *inout* (in/out) parameters may be used where appropriate. Also, exceptions should be defined to return values for error conditions rather than returning unions of normal and error values.

GDMO actions with a mode of *unconfirmed* (those that lack the *MODE CONFIRMED* clause) may be translated to methods with the IDL keyword *oneway* preceding the return type. Such operations must have a return type of *void* and no *out* or *inout* parameters, though. IDL operations without the *oneway* keyword are confirmed.

7.6 Notifications

These guidelines define the IDL equivalent of the 15 notifications found in ITU-T Rec. X.721, which are the notifications used in most GDMO information models. Typically, notifications in GDMO packages will simply be translated to a notification macro statement on each interface that includes the package. A *MANDATORY_NOTIFICATION* statement is used if the notification is part of a mandatory package and a *CONDITIONAL_NOTIFICATION* statement is used if it is part of a conditional package.

The mapping of object attributes to notification fields within a notification statement is not supported. If some special mapping is required it should be documented with a comment. Replies to notifications are not supported.

If a new notification must be defined it should be defined as an operation on an interface named “Notifications” within the information model’s module. The name of the operation shall be the name of the notification. The parameters to the operation shall be translated from the notification’s information syntax. The notification operation’s return type must be void, and it must have only *in* parameters. ITU-T Recommendation Q.816 provides information on how the data is placed into a structured notification. Note that attribute IDs are not needed. Instead, parameters are identified with a name and data type. The scoped interface name and notification operation may then be used within notification macro statements.

If a notification needs to be extended, it must be done by defining a new operation. The new operation should contain the same parameters as the old. For example, the IDL below extends the equipment alarm by adding a parameter named “newData” of type “newType.”

```
module newModule {
…
        interface Notifications {

                void equipmentAlarm (
                        in ExternalTimeType             eventTime,
                        …
                        in SuspectObjectSetType         suspectObjectList,
                        in newType                      newData);
        }
}
```

7.7 Behaviors

GDMO behavior templates shall be translated to formatted IDL comments immediately preceding the IDL construct with which each behavior is associated. Attribute behaviors shall be translated to IDL comments preceding the type definition for the attribute type.

Package behaviors shall be translated to IDL comments preceding the exception defined for the comment.

7.8 Name Bindings

Each GDMO name binding shall be translated into an IDL name binding module as defined in Section 6.8. The various constructs in the GDMO name binding shall be translated as follows.

The superior class name in the name binding shall be assigned to the value of the *superiorClass* constant in the name binding module. If the GDMO superior class clause has an *AND SUBCLASSES* modifier, the value of the IDL name binding constant *superiorSubclassesAllowed* shall be *true*. Otherwise, it shall be *false*.

The subordinate class name in the name binding shall be assigned to the value of the *subordinateClass* constant in the name binding module. If the GDMO subordinate class clause has an *AND SUBCLASSES* modifier, the value of the IDL name binding constant *subordinateSubclassesAllowed* shall be *true*. Otherwise, it shall be *false*.

If the GDMO name binding has a *CREATE* clause, the value of the IDL name binding constant *managersMayCreate* shall be *true*. Otherwise, it shall be *false*.

If the GDMO name binding has no *DELETE* clause, the value of the IDL name binding constant *deletePolicy* shall be *notDeletable*. If it has a *DELETE* clause with either no modifier or an *ONLY-IF-NO-CONTAINED-OBJECTS* modifier, the value of *deletePolicy* shall be *deleteOnlyIfNoContainedObjects*. If it has a *DELETE* clause with a *CONTAINED-OBJECTS* modifier, the value of *deletePolicy* shall be *deleteContainedObjects*.

If the name binding create clause has a *WITH-AUTOMATIC-INSTANCE-NAMING* modifier, the managed object factory create operation should define the name parameter as *inout*, and include a comment indicating that the client may submit a null name, and if so the factory will choose a name and return it.

Creating an object by copying a partial set of attribute values from a reference object is not possible with a strongly-typed factory method because there is no way for the factory to tell which values it should copy and which it should use from the operation's parameters. A strongly-typed operation that copies all values from a reference could be defined, but the utility of this is limited. A weakly-typed operation that accepted a reference object as well as a partial list of attributes could also be defined on a factory, but the difficulty of implementing this does not seem to be worth the benefit. Therefore, the translation of the *WITH-REFERENCE-OBJECT* modifier in a name binding create clause is not supported.

Parameters on create clauses shall be translated to *CreateError* exceptions. This may require defining a new value for the error ID. A comment should be placed in the name binding IDL module noting which *CreateError* exception error IDs apply to objects

created with that name binding. If it is not possible to translate a create clause parameter to a *CreateError* exception, another, less desirable, alternative is to define a new factory, and translate the parameter to an exception on a create operation on that factory. Because of the general-purpose nature of the *CreateError* exception, though, the need for this should be rare. (See more on parameters, below.)

7.9 Parameters

GDMO parameters provide extensibility for GDMO information models. Parameter templates are used to augment an existing specification in the areas of notifications, actions (requests, responses, and failures), and specific errors when defining subclasses. The GDMO definitions of all notifications and many actions contain an extensibility field that is further defined by the subclasses (if required). In the case of specific errors, class-specific errors are used to augment the general "processing failure" error in CMIP. The format of this information is often a list of name-value pairs, where the name defines the data type of the value.

Translating GDMO parameters to IDL provides a good opportunity to make the currently defined extensions that have been found useful with many object classes a "normal," strongly-typed part of the model. For example, three GDMO parameters that have been defined for alarms have been included in the notifications defined in the IDL. (The three parameters are "Alarm Effect On Service," "Suspect Object List," and "Alarming Resumed.")

There are several key words used in GDMO parameter templates to specify the semantics of the extensions. The translation of the various extension capabilities available with parameter templates based on these keywords is discussed below.

7.9.1 ACTION-INFO and ACTION-REPLY

In keeping with the strong typing recommended in the framework, GDMO parameters with the keywords "ACTION-INFO" in the template are not translated as an extension field. Instead, a new interface is subclassed from an existing interface that specifies the action but adds the extensions as regular "in" parameters of that method. The name of the IDL parameter should be taken from the name of the parameter, and the data type of the parameter should be translated from the GDMO parameter's syntax. "ACTION-REPLY" parameters would likewise be translated to "out" parameters on the operation.

The above method implies that subsequently adding a parameter to an already-existing IDL operation is not supported. Instead, the information modeler may use the more conventional approaches provided by CORBA for extending an interface, such as subclassing an object interface and defining a new method there, with additional *in* and/or *out* parameters, or additional exceptions. See Section 6.13 for guidelines on this.

7.9.2 EVENT-INFO and EVENT-REPLY

In cases where the "EVENT-INFO" parameters have already been defined, they are translated to regular "in" parameters on the IDL operations used to convey a notification.

These guidelines do not support responses to notifications, so there is no translation for "EVENT-REPLY" parameters.

Since this framework already defines a set of notifications, translating EVENT-INFO parameters could mean redefining one of the notification operations. See Section 7.6.

In most cases, however, re-using an existing notification definition will be preferred In cases where the GDMO extensions are predefined, as for alarm information, they should be included in the translated notification IDL specifications. The framework notification IDL, however, also supports an "additional information" field, which is a weakly typed name-value pair list. This can be used to add information to these previously-defined notifications. The notification event type will not change. The new managed object interface that needs to use the extension for a specific parameter must note the use of this parameter in comments, though. Unfortunately, there is no other mechanism except using the macros shown above to specify which notifications are supported by which objects, and this does not support also specifying parameters. The advantage of using the same notification type is to allow the managers to receive the notifications and not be concerned with having to register for a new notification type. If the extensions are not understood because of different versions of manager and agent, then the additional information is discarded.

The specification of the extensions for the additional information is described below.

The notifications defined by this framework include a field named "additionalInformation" that closely resembles the "additionalInformation" field in CMIP notifications. The IDL syntax of the "additionalInformation" field in the notifications is type "AdditionalInformationSetType":

```
struct ManagementExtensionType {
        UIDType id;             // identifies the type of info
        any     info;           // type will depend on id
};

typedef sequence <ManagementExtensionType> AdditionalInformationSetType;
```

Parameters with the EVENT-INFO keywords are translated by defining a Unique Identifier (UID) for each parameter. See Section 6.11 for details on this. In short, though, the modeler defines a sub-module named "AdditionalInformationConst" in which a constants of value type "short" is defined. The names of these constants are the names of the GDMO parameters. The value of each constant could also be derived from the GDMO, based perhaps on the last number of the parameter's registration. Otherwise, an integer unique to the constants in that module should be chosen. This definition must also include a comment indicating the data type of the value that accompanies the UID in the "additionalInformation" field. As an example, if the Alarm Effect On Service parameter had not been made a normal member of the Alarm Info data structure used by alarms in this framework, it might have been translated like this:

```
module itut_m3100 {
…
```

```
        module AdditionalInformationConst {

        /** Alarm effect on service parameters are accompanied by a boolean
        value in the “any” field indicating if service has been affected. */

        const short alarmEffectOnService = 1;
        …
        }; // end of module AdditionalInformationConst

      }; // end of module itut_m3100
```

A managed object’s IDL interface can then identify the notifications it supports as usual, but a comment should indicate the parameters that will be included in the notifications.

7.9.3 Context-Keyword

Context-keyword parameters identify information that is to be passed in a named field in a CMIP PDU. This named field is usually a sequence of data structures consisting of an identifier and an “any” data type which holds a value whose type depends on the identifier. In CMIP, these context-keyword parameters may be passed in action parameters or in notifications. The translation of context-keyword parameters for actions is not supported by this framework due to the preference for strong typing. Instead, additional information for actions should be translated to regular operation parameters. (See ACTION-INFO parameters above.)

For notifications, except for extensions (explained above), if fields are defined to be of a weak type, then the same approach as for the extension field can be used. However, this approach has not been used in most of the GDMO standards. The distinction in the case with EVENT-INFO keyword versus context-keyword is the former is designed for extensibility where one or more parameters can be added. The recommended approach in the case of multiple extension is the use of EVENT-INFO and therefore all standards have defined parameters using this keyword.

7.9.4 SPECIFIC-ERROR

“SPECIFIC-ERROR” parameters are returned in CMIP processing failure messages. They indicate an abnormal outcome of an operation. There are two options for translating these parameters. First, they may be translated to IDL exceptions raised by the operation for which the specific error parameter is defined. The name of the exception should be taken from the GDMO parameter name, and the data type returned with the exception should be derived from the GDMO parameter’s syntax. Since specific-error parameters may be defined for different kinds of GDMO templates, specific error parameters on actions should be translated to exceptions raised by the action and specific error parameters on attributes should be translated to exceptions raised by the attribute access operation. Also, specific error parameters on the “Create” clause of a name binding should be translated to exceptions on the create operation on the factory interface. There is no translation of a specific-error parameter on a notification supported by this framework since responses to notifications are not allowed.

The second option for translating specific-error parameters is to translate the parameter into a new code point for one of the standard exceptions defined by the framework. The

framework defines three standard exceptions: the *CreateError* exception, raised on factory create operations, the *DeleteError* exception, raised on managed object delete operations, and *ApplicationError* exceptions, raised on all other managed object operations. The *ApplicationError* exception returns a unique identifier that identifies the specific application error, and a text explanation. The create and delete error exceptions extend this information by adding a list of related objects that may be involved, and the attributes of the object on which the object was attempted. The list of related objects might show, for example, some objects that must be deleted before the target object can be deleted. The attributes might contain object state information pertinent to the error.

Translating a specific-error to a code point used by one of these standard exceptions should be used whenever possible. Since the data types returned in the exceptions are value types, they may be extended for specific code points. Because the delete operation is inherited from the base managed object interface specific-error parameters appearing in GDMO name binding delete clauses must be translated to *DeleteError* exception code points. This is done similarly to the EVENT-INFO parameters described above. Basically, the modeler defines a delete error sub-module for UID constants. The constant definitions must include a comment indicating what data will be placed in the "relatedObjects" and "attributeList" fields accompanying an error with that identifier. Also, if the modeler has extended the standard value type returned for the code point, a comment must note the actual data type returned so that the managing system may narrow the type and access the additional information. The framework, in fact, includes some delete error code points that extend the standard delete error value type.

Finally, a comment on the managed object's IDL interface indicates the delete error values that might be raised in an exception when an incorrect attempt to delete the object is made. An example translation is:

```
module itut_m3100 {
…
 module DeleteErrorConst {

 /** Network TTP Terminates Trail delete errors are raised when an
 attempt is made to delete a TTP before the trail has been deleted.
 It includes a reference to the Trail in the "relatedObjects" field.  */

 const short networkTTPTerminatesTrail = 54;
 …
 }; // end of module DeleteErrorConst

}; // end of module itut_m3100
```

7.10 ASN.1 Data Types

GDMO uses the ASN.1 language to define the syntax of attributes as well as operation and notification parameters, so when converting GDMO templates to IDL, these syntax definitions will also have to be translated. This section gives guidelines on translating ASN.1 syntax to CORBA IDL.

7.10.1 Basic Types

CORBA IDL defines the following basic types to which ASN.1 basic types may be translated: any, boolean, char, double (for double-precision floating-point numbers), enum (for enumerated types), fixed, float (for single-precision floating-point numbers), long (for large integers), object (for object references), octet, short (for small integers), string, wchar (for "wide" characters), and wstring (for strings of "wide" characters).

This framework uses the *string* type for all strings, and defines a *typedef* called "Istring" for cases where the string may contain escaped international characters. Istring is a typedef of wstring, or "wide" strings. These are string composed of "wide" (16-bit) characters.

> Temporary Note - Contributions are solicited on using the alternative typedef of Istring to string instead of wstring since strings can carry international character sets when codeset negotiation, supported by GIOP version 1.1 and greater, is used. Wstring types are mapped by CORBA language bindings to the programming language wstring type, which is often tied to just Unicode.

In addition, the CORBA Time service defines a time type referred to a "UtcT" that is used by this framework.

7.10.2 Sequence

CORBA IDL supports the definition of data structures using the *struct* keyword, similar to ASN.1 *sequence* types.

7.10.3 Sequence of

CORBA IDL supports the definition of sequences of types, both basic and complex, in much the same way as the ASN.1 *sequence of* type.

7.10.4 Set of

CORBA IDL does not support the definition of complex set types as does ASN.1. Instead, sets are translated to IDL sequences. The convention of ending the type name with "SetType" shall be followed. When handling set values, duplicates should be eliminated and order ignored.

7.10.5 Choice

CORBA IDL supports the definition of discriminated unions, which serve the same purpose as ASN.1 choice types.

In the interest of simplifying the implementation of CORBA-based TMN standards, this framework recommends the conservative use of discriminated unions. Often when translating from ASN.1 to CORBA IDL, the translated type can be simplified with no loss of semantics. For example, usually a choice between a string and null can simply be translated to a string. A comment that the string may possibly be null can be added to identify this possibility. A choice between a sequence of (or set of) and null can likewise be translated to just the sequence.

7.10.6 Object Identifier (OID)

This framework defines a type called “Universal Identifier” (UID) that is designed to be a replacement for ASN.1 OIDs.

7.10.7 Object Instance

The framework supports two possible translations for ASN.1 object instance types. Since each managed object has a name, the name type defined by the CORBA Naming Service can be used. (This framework defines a *typedef* for the CORBA Naming Service names, called *NameType*.) Also, CORBA object references may be used. Since all managed object interfaces must inherit from the *ManagedObject* interface, the type *ManagedObject* should be used whenever a general reference to an object is required. The modeler may also use a type specific to a class of managed objects, such as *Equipment*. This has the advantage of making a model more strongly typed.

8 Style Idioms for CORBA IDL Specifications

This section defines a set of style idioms for the Interface Definition Language (IDL) of the Common Object Request Broker Architecture (CORBA) to be used in interface specifications. Having a set of style idioms will result in CORBA/IDL specifications with a consistent style. This may require some additional work by editors, but this extra effort is worth the increased readability of the CORBA/IDL specifications. It is important to keep in perspective that style conventions are for the benefit of the reader, not necessarily to the benefit of the author.

8.1 Use Consistent Indentation

This section demonstrates the indentation style that may be used in the IDL modules. As an example, an excerpt from the CORBA Security Service non-repudiation module is shown below:

```
enum EvidenceType {
    SecProofofCreation,
    SecProofofReceipt,
    SecProofofApproval,
    SecProofofRetrieval,
    SecProofofOrigin,
    SecProofofDelivery,
    SecNoEvidence // used when request-only token desired
};

interface NRPolicy {
    void get_NR_policy_info (
        out Security::ExtensibleFamily  NR_policy_id,
        out unsigned long               policy_version,
        out Security::TimeT             policy_effective_time,
        out Security::TimeT             policy_expiry_time,
        out EvidenceDescriptorListType  supported_evidence_types,
        out MechanismDescriptorListType supported_mechanisms
    );
};
```

8.2 Use Consistent Case for Identifiers

Several languages enforce case rules (such as ASN.1) while others have de-facto rules. These rules allow readers to easily distinguish identifiers of different type leading to increased readability. IDL does not enforce case, so the following rules are proposed.

- Operations, parameters, attributes, members and constants shall have every embedded word capitalized except for the first word capitalized.
- All other identifiers shall have the first letter of every embedded word capitalized.

```
module CarModule {

    struct EngineType {
        PistonType piston;
        RodType    pistonRod;
    };

    typedef string KeyType;

    enum WontStartReasonType {
        BatteryIsDead,
        NoGas
    };

    exception WontStart {
        WontStartReasonType reasonEngineWontStart;
    };

    interface FordRanger {

        void startEngine(
            in KeyType key
        )
        raises (
            WontStart;
        );

        attribute EngineType engine;

    };
};
```

8.3 Follow JIDM Approach for IMPORT

At the beginning of a module that imports a type from another module, create a local typedef. This explicitly lists the type that the importing module is dependent upon from the exporting module. (Note: the name of the local identifier need not be the same name

```
module ImportingModule {

    // Imports
    typedef ExportingModule::SomeType            SomeType;
    typedef ExportingModule::SomeOtherType       SomeOtherType;
    typedef ExportingModule::SomethingElse       SomethingElseType;

    ...
};
```

as the identifier in the exporting module).

8.4 Use JIDM Approach for OPTIONAL and CHOICE

For enumerated and numeric (integer and floating) types, use the ASN OPTIONAL and CHOICE mappings to IDL as prescribed in the Open Group and Open-Network Management Forum Joint Inter-domain Management (JIDM) group's Inter-Domain Management: Specification Translation.[3] An example is given below:

```
// Choice
enum CarChoiceType {
    Ford,
    Cheverolet,
    Chrysler
};

union CarType switch (CarChoiceType) {
    case Ford:        FordType          fordValue;
    case Cheverolet:  ChevroletType     chevroletValue;
    case Chrysler:    ChryslertType     chryslerValue;
}

// Optional
union SunRoofTypeOpt switch(boolean) {case TRUE: SunRoofType the_value};
```

For strings, sequences, and object references, a null value can usually be used to represent optional cases where no value is present. In cases where there is a semantic difference between a null and a not present, the above method may be used.

For structures and unions, the above method may be used or a decision may be made to use null values within the structure to represent optional values that are not present. For example, for a structure composed of two strings, two nulls could represent an optional value that is not present. If a value is optional it should be marked as optional with a comment.

As always, guidelines need to be used with common sense. The resulting translation should be evaluated for clarity and usability. If the translation is too complex, the modeler may want to try to simplify it.

8.5 Use a Consistent Type Suffix

Append the suffix "Type" to all IDL types. This allows type identifiers and members to use the same name without collisions since IDL is case insensitive. In addition, this idiom increases readability by clearly separating type identifiers from other identifiers.

8.6 Use a Consistent Suffix for Sequence Types.

For sequences (ordered, duplicates allowed) use a suffix of "SeqType" to distinguish sequences from singulars.

8.7 Use a Consistent Suffix for Set Types.

For sets (unordered, duplicates disallowed) use a suffix of “SetType” to distinguish sets from singulars.

8.8 Use a Consistent Suffix for Optional Types

For optional types use a suffix of “TypeOpt” to distinguish them from the non-optional type.

8.9 Arrange Operation Parameters in a Consistent Manner

A consistent ordering of parameters increases readability. Arrange parameters to operations by in, out, then inout.

8.10 Assume No Global Identifier Spaces

To reduce name collisions and promote reuse, all identifiers shall be scoped to a particular context (e.g., module, and interface).

8.11 Module Level Definitions

All type definitions shall be at the module level. Nesting type definitions within a lower context leads to difficulties in reuse and duplication.

8.12 Use of Exceptions and Return Codes

Exceptions shall be used for exceptional conditions such as error conditions. Normal returns shall be handled though return codes and output parameters.

8.13 Explicit vs. Implicit Operations

An operation should perform an explicit function. Using parameters as a flag to implicitly change the behavior of the operation can be confusing. Factor each behavior into a separate explicit operation.

8.14 Don’t Create a Large Number of Exceptions

Having a large number of exceptions increases the difficulty of understanding an interface definition. Group exceptions by category, or make use of the standard exceptions (*ApplicationError*, *CreateError*, and *DeleteError*) by defining new error code points for them, if necessary.

9 Compliance and Conformance

This section defines the criteria that must be met by other standards documents claiming compliance to these guidelines and the functions that must be implemented by systems claiming conformance to this specification.

9.1 Standards Document Compliance

Any specification claiming compliance with these guidelines shall:

1. Derive (directly or indirectly) all interfaces that model resources from the *ManagedObject* interface described in Section 5.1 and defined in the CORBA IDL in Annex A.
2. Define, for each managed object class that can be instantiated, a factory interface derived (directly or indirectly) from the *ManagedObjectFactory* interface described in Section 5.2 and defined in the CORBA IDL in Annex A.
3. Use the constants defined in the CORBA IDL in Annex B whenever appropriate.
4. Use the notifications described in Section 5.3 and defined in the CORBA IDL in Annex A whenever appropriate.
5. Adhere to the conventions for defining CORBA TMN managed objects specified in Section 6.
6. Adhere to the IDL conventions specified in Section 8
7. Specify notifications as methods on a "Notifications" interface if none of the notifications defined in this document are applicable.
8. Define and use a NO<package name> exception for identifying the attributes and actions that are parts of a conditional package.
9. Use the macros defined in this document for identifying the notifications that are to be supported by a managed object.
10. Use the definitions for generic attribute types found in Section 6.3.5 wherever applicable.
11. Define IDL name binding modules to identify allowable containment relationships.
12. State in its compliance clause a reference to the module(s) from which other generic attributes are used.
13. Follow the GDMO to IDL mapping rules defined in Section 7 if the IDL model is a translation from GDMO.

9.2 System Conformance

An implementation claiming conformance to this document shall:

1. Support all of the capabilities of the *ManagedObject* interface described in Section 5.1
2. Support the create operation behavior described in Section 6.9.

9.3 Conformance Statement Guidelines

The users of these guidelines must be careful when writing conformance statements. Because IDL modules are being used as name spaces, they may, as allowed by OMG IDL rules, be split across files. Thus, when a module is extended its name won't change. Instead, a new IDL file will simply be added. Simply stating the name of a module in a conformance statement, therefore, will not suffice to identify a set of IDL interfaces. The conformance statement must identify a document and year of publication to make sure the right version of IDL is identified.

Annex A The Object Model CORBA IDL Module

(Normative)

```
/* This IDL code is meant to be stored in a file named “itut_x780.idl”
located in the search path used by IDL compilers on your system. */

#ifndef ITUT_X780_IDL
#define ITUT_X780_IDL

#include <CosNaming.idl>
#include <CosTime.idl>
#include <itut_x780Const.idl>

#pragma prefix “itu.int”

/* Most comments in this file are formatted to be parsed by an IDL-to-HTML
converter such as idldoc or orbacus hidl. */
```

// MODULE itut_x780

```
/** This module provides the fundamental capabilities for implementing network
management interfaces and defines the "managed object" interface. The
interfaces below are modeled after the managed object specifications
found in the ITU-T CMIP specification document X.721. */

module itut_x780 {
```

// IMPORTED TYPES

```
        // Types imported from CosNaming
        typedef CosNaming::Name NameType;

        // Types imported from CosTime
        typedef TimeBase::UtcT UtcT;
```

// FORWARD DECLARATIONS AND TYPEDEFS

```
        /** International strings are strings of wide (16 bit unicode)
        characters. */

        typedef wstring Istring;

        /** Istring Sets are just sets of Istrings */

        typedef sequence <Istring> IstringSetType;

        /**  Additional Text Type is often used in notifications to convey a
        text explanation for the notification.
        */

        typedef Istring AdditionalTextType;

        /**  Avalibility Type is used in a sequence to indicate the
        availability of a resource.  Zero or more of these conditions may be
        indicated.
        */

        typedef short AvailabilityStatusType;
```

```
const AvailabilityStatusType inTest = 0;
const AvailabilityStatusType failed = 1;
const AvailabilityStatusType powerOff = 2;
const AvailabilityStatusType offLine = 3;
const AvailabilityStatusType offDuty = 4;
const AvailabilityStatusType dependency = 5;
const AvailabilityStatusType degraded = 6;
const AvailabilityStatusType notInstalled = 7;
const AvailabilityStatusType logFull = 8;

/**  Availability status is used to indicate the availability of a
resource.  It is represented as a sequence of integers because several
of the conditions may exist at once.
*/

typedef sequence<AvailabilityStatusType> AvailabilityStatusSetType;

/** Backed Up Status Type is used to indicate if an object has a back
up. */

typedef boolean BackedUpStatusType;

/** Control Status Type is used in a sequence to indicate the
control status of a resource.  Zero or more of these may be indicated.
*/

typedef short ControlStatusType;

const ControlStatusType subjectToTest = 0;
const ControlStatusType partOfServicesLocked = 1;
const ControlStatusType reservedForTest = 2;
const ControlStatusType suspended = 3;

/**  Control status set is used to indicate the control status of a
resource.  It is represented as a sequence of integers because several
of the conditions may exist at once.
*/

typedef sequence<ControlStatusType> ControlStatusSetType;

/** Generalized time is a basic ASN.1 type.  It is usually represented
as a string in computing languages but it has certain, parseable
formats. The 3 possible forms are:  <ol><li>

Local time only. "YYYYMMDDHHMMSS.fff", where the optional fff is
accurate to three decimal places, <li>
Universal time (UTC time) only. "YYYYMMDDHHMMSS.fffZ", and <li>
Difference between local and UTC times. "YYYYMMDDHHMMSS.fff+-HHMM".
</ol>
The options for representing this in IDL seem to be either a string or
the UtcT structure from the CORBA Time Service.  UtcT makes it a little
easier to compare times from different zones, but requires managed
systems to know their time zones.  UtcT was picked.
*/

typedef UtcT GeneralizedTimeType;

/** External Time is generalized time. */

typedef GeneralizedTimeType ExternalTimeType;

/** Forward declaration. */
```

```
interface ManagedObject;

/** MO is shorthand for Managed Object.  CORBA uses object references
of type "object" to identify objects.  These are used instead of ASN.1
object instances.  For network management interfaces, all objects will
inherit from the "ManagedObject" interface.  */

typedef ManagedObject MO;

/** MO Set is a set of MO references. */

typedef sequence <MO> MOSetType;

/** MO Seq is a sequence of MO references. */

typedef sequence <MO> MOSeqType;

/** A set of names is definded as a sequence of names.  */

typedef sequence <NameType> NameSetType;

/** Notification IDs are long integers.  */

typedef long NotifIDType;

/** This defines a set of notification IDs. */

typedef sequence <long> NotifIDSetType;

/** Procedural Status Type is used in a sequence to indicate the
procedural status of a resource.  Zero or more of these may be
indicated.
*/

typedef short ProceduralStatusType;

const ProceduralStatusType initializationRequired = 0;
const ProceduralStatusType notInitialized = 1;
const ProceduralStatusType initializing = 2;
const ProceduralStatusType reporting = 3;
const ProceduralStatusType terminating = 4;

/**  Procedural Status Set is used to indicate the procedural status of
a resource.  It is represented as a sequence of integers because
several of the conditions may exist at once.
*/

typedef sequence<ProceduralStatusType> ProceduralStatusSetType;

/** ScopedName is just a string. */

typedef string ScopedNameType;

/** Scoped Name Sets are simply sets of Scoped Names. */

typedef sequence <ScopedNameType> ScopedNameSetType;

/** In CORBA, strings containing scoped names are used to identify
object classes (actually, "interfaces"). */

typedef ScopedNameType ObjectClassType;

/** Object Class Set is a set of object classes */
```

```
typedef sequence <ObjectClassType> ObjectClassSetType;

/** Name Binding Modules are identified with scoped names. */

typedef ScopedNameType NameBindingType;

/** StartTimeType is used to specify a time when something starts.
It is often paired with a StopTimeType to control the activation of
some function.
*/

typedef GeneralizedTimeType StartTimeType;

/** String sets are sets of strings. */

typedef sequence <string> StringSetType;

/** Unknown status is used to indicate if the status of a resource is
not known.  A value of true indicates the status is unknown.  */

typedef boolean UnknownStatusType;
```

// ENUMERATED TYPES

```
/* The following state objects are used in many interfaces and parallel
the state objects in CMIP standards. */

/** Administrative State is read/write.  A "locked" object is usually
one that may not be changed or one which is not providing service.
Setting the Admininstrative State of an object to "shuttingDown" begins
the shutdown process for that object. */

enum AdministrativeStateType {locked, unlocked, shuttingDown};

/** Operational State is read only.  It simply reports the current
capability of the object to provide service. */

enum OperationalStateType {disabled, enabled};

/** Usage state is read only.  If "idle," the resource is completely
unused.  If "busy," the total capacity of the resource is in use.
"Active" is in between.  */

enum UsageStateType {idle, active, busy};

/** Delete Policy indicates if an object can be deleted and if so if
any contained objects should automatically be deleted.  Since objects
must not be orphaned, if an object has a delete policy of
“deleteOnlyIfNoContainedObjects” the object must not be deleted if it
has contained objects.  A value of “deleteContainedObjects” means if
the object is deleted its contained objects should also be deleted.  */

enum DeletePolicyType {notDeletable, deleteOnlyIfNoContainedObjects,
                deleteContainedObjects};

/** PerceivedSeverity reports the severity of an alarm. "Indeterminate"
is used when it is not possible to assign one of the other values */

enum PerceivedSeverityType {indeterminate, critical, major, minor,
                                warning, cleared};
```

```
/** Source Indicator is used in many notifications. It identifies
whether the notification is a result of a management operation or
something that occurred on the managed system.  */

enum SourceIndicatorType {resourceOperation, managementOperation,
                                unknown};

/** The standby status attribute is single-valued and read-only.
The value is only meaningful when the back-up relationship role exists.
If "hot standby" the resource is not providing service, but is
operating in synchronism with another resource that is to be backed-up.
If "cold standby" the resource is to back-up another resource, but is
not synchronized with that resource. If "providing service" the back-up
resource is providing service and is backing up another resource.
*/

enum StandbyStatusType {hotStandby, coldStandby, providingService};

/** Stop times are used to specify when some function should cease.
There are normally two choices, the function runs continually (in
which case no actual time is specified) or the function ends at
a specified time.
*/

enum StopTimeChoice {specific, continual};

/** Threshold indication describes if the threshold crossed was in the
up or down direction. */

enum ThresholdIndicationType {up, down};

/** TrendIndication values indicate if some observed condition is
getting better, worse, or not changing.  */

enum TrendIndicationType {lessSevere, noChange, moreSevere};
```

// STRUCTURES AND UNIONS

```
/* The structures defined below are used to pass values that may be
optionally included.  For some types of values, like strings, lists,
and pointers, it is easy to tell if the value is included.  For others,
like enumerations, numbers, and structures, it is not.  */

/** AdministrativeStateTypeOpt is an optional type.  If the
discriminator is true the value is present, otherwise the value is
null.  */

union AdministrativeStateTypeOpt switch (boolean) {
        case TRUE:      AdministrativeStateType value;
};

/** BooleanTypeOpt is an optional type.  If the discriminator is
true the value is present, otherwise the value is null.  */

union BooleanTypeOpt switch (boolean) {
        case TRUE:      boolean value;
};

/** FloatTypeOpt is an optional type.  If the discriminator is
true the value is present, otherwise the value is null.  */

union FloatTypeOpt switch (boolean) {
```

```
        case TRUE:      float   value;
};

/** LongTypeOpt is an optional type.  If the discriminator is
true the value is present, otherwise the value is null.  */

union LongTypeOpt switch (boolean) {
        case TRUE:      long    value;
};

/** OperationalStateTypeOpt is an optional type.  If the discriminator
is true the value is present, otherwise the value is null.  */

union OperationalStateTypeOpt switch (boolean) {
        case TRUE:      OperationalStateType    value;
};

/** ShortTypeOpt is an optional type.  If the discriminator is
true the value is present, otherwise the value is null.  */

union ShortTypeOpt switch (boolean) {
        case TRUE:      short   value;
};

/** TrendIndicationTypeOpt is an optional type.  If the discriminator
is true the value is present, otherwise the value is null.  */

union TrendIndicationTypeOpt switch (boolean) {
        case TRUE:      TrendIndicationType     value;
};

/** UnsignedShortTypeOpt is an optional type. If the discriminator is
the value is present, otherwise the value is null. */

union UnsignedShortTypeOpt switch (boolean) {
        case TRUE:      unsigned short value;
};

/** UsageStateTypeOpt is an optional type.  If the discriminator is
true the value is present, otherwise the value is null.  */

union UsageStateTypeOpt switch (boolean) {
        case TRUE:      UsageStateType  value;
};

/** Many times interface specifications need to define standard values
to be passed across the interface.  Also, often the scheme used to
define these values needs to be extensible as new interfaces are
subclassed, so enumerations don't work well.  CMIP uses OIDs, strings
of numbers that are often appended, in standards.  To serve this
purpose, the Unique ID is used.  It consists of two parts, a string
containing a scoped module name, and an integer value defined as a
constant within that module.  These UIDs, and the ObjectClass type
defined above, replace ASN.1 OIDs.  It is expected that each module
will contain a constant string named "moduleName" that contains the
name of the module for error-free use by the programmer.  A null module
name will indicate a null value for the UID. <p>

Code to interpret a UID might look like the following code snippet:<br>
<code><pre>
UIDType pc;     // probable cause
...
if (pc.moduleName ==
```

```
        itut_x780::ProbableCauseConst::moduleName) //string compare
        switch (pc.value) {
        case itut_x780::ProbableCauseConst::adapterError:
        ...
        case
        itut_x780::ProbableCauseConst::applicationSubsystemFailure:
        ...
        case itut_x780::ProbableCauseConst::bandwidthReduced:
        ...
        }
else if (pc.moduleName == MyLocal::ProbableCauseConst::moduleName)
        switch (pc.value) {
        ...
        }
</pre></code>
@member moduleName      The scoped module name where values are
                        defined.
@member value           The value defined as a constant within the
                        module.
*/

struct UIDType {
        string moduleName;      // module where value is defined
        short value;            // constant within the module
};
typedef sequence <UIDType> UIDSetType;

/** Management Extension is a structure for flexibly reporting
information.  It is typically used in the Additional Information field
of notifications.
@see <a href="#AdditionalInformationSetType">
AdditionalInformationSetType </a>
@member id      identifies the type of information
@member any     contains the actual information, type will depend on
                the value of the id member.
*/

struct ManagementExtensionType {
        UIDType id;             // identifies the type of info
        any     info;           // type will depend on id
};

/** Additional Information is a flexible way to report information that
does not fit into the structure of a notification.  It contains a
sequence of a structure called "Management Extension". */

typedef sequence <ManagementExtensionType>
                        AdditionalInformationSetType;

/** An Attribute Value structure is used in a notification to report
the value of any attribute.  The string used for the attribute's name
is the same as the name of the data member in the value object defined
for the object. In other words, it is the name of an attribute accessor
method minus the "get" or "set".
@member attributeName   the name of the attribute
@member value           contains the value of the attribute, type will
                        depend on the attributeName.
*/

struct AttributeValueType {
        string  attributeName;
        any     value;          // type will depend on the attribute
};
```

```
/** Attribute Value Sets are used to report attributes generically,
in a batch mode. */

typedef sequence <AttributeValueType> AttributeSetType;

/** An Attribute Value Change structure is used in a notification to
report an attribute that has been changed.
@see <a href="#AttributeValueType">AttributeValueType</a>
@member attributeName   the name of the attribute
@member oldValue        the old value, type will depend on the
                        attributeName
@member newValue        the new value, type will depend on the
                        attributeName.
*/

struct AttributeValueChangeType {
        string          attributeName;
        any             oldValue;       // type depends on attribute
        any             newValue;       // type depends on attribute
};

/** An Attribute Change Set is used to report the attributes that have
been changed in an attribute value change notification. */

typedef sequence <AttributeValueChangeType> AttributeChangeSetType;

/** A Correlated Notification is identified by the object that emitted
the notification and the notification ID.  Both are included in case
the Notification IDs are not unique across objects.
@member source  Reference to object that emitted the correlated
                notification.  If null, the correlated notifications
                are from the same source as the notification containing
                this data structure.
@member notifIDs IDs of the correlated notifications.  Notification
                identifiers must be chosen to be unique across all
                notifications from a particular managed object
                throughout the time that correlation is significant.
*/

struct CorrelatedNotificationType {
        NameType        source;
        NotifIDSetType  notifIDs;
};

/** Correlated Notification sets are sets of Correlated Notification
structures. */

typedef sequence <CorrelatedNotificationType>
                CorrelatedNotificationSetType;

/** ProbableCause, in CMIP standards, may be either an integer or GDMO
OID, a dot-notation string.  The UID type is used instead. */

typedef UIDType ProbableCauseType;

/** Proposed Repair Actions are sets of unique identifiers. */

typedef UIDSetType ProposedRepairActionSetType;

/** Security Alarm Causes are unique identifiers. */

typedef UIDType SecurityAlarmCauseType;
```

```
/** Security Alarm Detector can indicate either a mechanism or a
specific object.  According to X.721 a choice is made between one or
the other, though it is not clear why.  (Actually, X.721 adds a third
choice for an AE-title which has no equivalent here.)  Unless otherwise
indicated, then, at most one of the members will be non-null.  Two
nulls may be sent if the managed system does not support this property.
@member mechanism       the scheme or function detecting the alarm, may
                        be null
@member obj             the object detecting the alarm, may be null
*/

struct SecurityAlarmDetectorType {
        UIDType         mechanism;      // may be null
        NameType        obj;            // may be null
};

/** Service User
@member id      the id of the service user
@member details details about the service user, type will depend on id
*/

struct ServiceUserType {
        UIDType id;
        any     details;        // value will depend on id
};

/** Service Providers share the same representation as Service Users.
*/

typedef ServiceUserType ServiceProviderType;

/** Specific Problems are sets of unique identifiers. */

typedef UIDSetType SpecificProblemSetType;

/** A Stop Time Type is used to indicate when some function should
cease.  In the specific case, an actual time is given.  In the
continual case, the function runs continually and no value is
carried in this union.
*/

union StopTimeType switch (StopTimeChoice) {
        case specific:  GeneralizedTimeType time;
        /* case continual carries NULL value */
};

/** A SuspectObject identifies an object that may be the cause of a
failure.  It is usually a component of a SuspectObjectList.
@member objectClass             Object class of the suspect object
@member suspectObjectInstance   Object instance of the suspect object
@member failureProbability      Optional failure responsibility
                                probability from 1 to 100
*/

struct SuspectObjectType {
        ObjectClassType         objectClass;
        MO                      suspectObjectInstance;
        UnsignedShortTypeOpt    failureProbability;
};

/** Suspect Object Lists are used to identify objects that may be the
cause of a failure.
```

```
*/

typedef sequence<SuspectObjectType> SuspectObjectSetType;

/** Threshold Level Indication describes multi-level threshold
crossings.  Up is the only permitted choice for a counter.  In ASN.1,
if indication is “up”, low value is optional.
@member indication      indicates up or down direction of crossing.
@member low             the low observed value.
@member high            the high observed value.
*/

struct ThresholdLevelIndType {
        ThresholdIndicationType         indication;
        FloatTypeOpt                    low;    // observed value
        float                           high;   // observed value
};

/** Threshold Level Ind Type Opt is an optional type.  If the
discriminator is true the value is present, otherwise the value is
null.  */

union ThresholdLevelIndTypeOpt switch (boolean) {
        case TRUE:      ThresholdLevelIndType   value;
};

/** Threshold Information indicates some guage or counter attribute
passed a set threshold. The structure differs from X.721 some to
simplify the syntax.
@member attributeID     Identifies the attribute that crossed the
                        threshold.  Actually, it is an operation name
                        on an interface minus the "get" or "set".  The
                        interface on which the operation is defined is
                        included elsewhere in the notification as
                        ObjectClass.  A Null value indicates the entire
                        structure is null.
@member observedValue   Attributes that are of type integer will be
                        converted to floats.
@member thresholdlevel  This parameter is for multi-level threhsolds.
                        Optional.
@member armTime         May be null(0). */

struct ThresholdInfoType {
        string                          attributeID;
        float                           observedValue;
        ThresholdLevelIndTypeOpt        thresholdLevel;
        ExternalTimeType                armTime;
};
```

// EXCEPTIONS

```
/** Application error info types are passed back in managed object
exceptions.
@member error   A unique identifier identifying the problem.
@member details A text message with additional information about the
problem.
*/

valuetype ApplicationErrorInfoType {
        public UIDType          error;
        public Istring          details;
};
```

```
/** Create error info types are passed back in managed object create
exceptions.  They extend application error info types.
@member relatedObjects  objects that have some relationship to the
                        object to be created that somehow prevented the
                        creation.
@member attributeList   the values that would have been assigned to the
                        created object.  These may hold some key to why
                        the object could not be created.
*/

valuetype CreateErrorInfoType : ApplicationErrorInfoType {
        public MOSetType        relatedObjects;
        public AttributeSetType attributeList;
};

/** Delete error info types are passed back in managed object delete
exceptions.  They extend application error info types.
@member relatedObjects  objects that have some relationship to the
                        object to be deleted that somehow prevented the
                        deletion.
@member attributeList   the attribute values assigned to the object to
                        be deleted.  These may hold some key to why the
                        object could not be deleted.
*/

valuetype DeleteErrorInfoType : ApplicationErrorInfoType {
        public MOSetType        relatedObjects;
        public AttributeSetType attributeList;
};

/** A package error info type is a special create error.  It will be
passed back in a managed object create exception as a create error.  If
the UID error code matches the package error info type, the client
application may narrow the value type from create error info type to
package error info type to access the additional information.
@member packages        the list of requested packages that conflicted
                        or could not be supported.
*/

valuetype PackageErrorInfoType : CreateErrorInfoType {
        public StringSetType    packages;
};

/** Application error exceptions may be raised on any managed object
operation to identify a problem preventing the operation from being
completed.  */

exception ApplicationError { ApplicationErrorInfoType info; };

/** Create error exceptions may be raised on any managed object create
operation to identify a problem preventing the object from being
completed.  */

exception CreateError { CreateErrorInfoType info; };

/** Delete error exceptions may be raised by a managed object in
response to an attempt to delete the object.  They may also be raised
by the terminator service.  */

exception DeleteError { DeleteErrorInfoType info; };
```

// MANAGED OBJECT INTERFACE

```
/** This valuetype object contains members for each of the attributes
accessible on this interface. */

valuetype ManagedObjectValueType {
        public NameType                 name;
        public ObjectClassType          objectClass;
        public StringSetType            packages;
        public SourceIndicatorType      creationSource;
        public DeletePolicyType         deletePolicy;
};

/** The Managed Object interface is intended to be the base interface
from which all other managed object interfaces inherit.  It is a
central place to specify basic functions which all managed objects are
expected to support. */

interface ManagedObject {

        /** This method returns the fully-qualified name for the
        object.  This method is used rather than having a "get*ID"
        method defined for each interface, as is done in CMIP
        specifications.  This will ensure that objects have only a
        single operation to retrieve names when they are sub-classed.
        <p>

        The response is a sequence of name component structures,
        starting with the name assigned to the “local root” naming
        context under which this object is contained.  The client may
        find the superiors of this object by removing components from
        the tail end of this sequence and performing a resolve
        operation on the first part of the name.  */

        NameType nameGet()
                raises (ApplicationError);

        /** This method returns the scoped name of the most-specific
        class of the interface (e.g. “EquipmentR1”). */

        ObjectClassType objectClassGet()
                raises (ApplicationError);

        /** This method returns a list of all the conditional packages
        supported by this instance. */

        StringSetType packagesGet ()
                raises (ApplicationError);

        /** This method returns an indication of how the object was
        created. */

        SourceIndicatorType creationSourceGet()
                raises (ApplicationError);

        /** This method returns a value indicating if the object may be
        deleted and if it may, if all contained objects are
        automatically deleted. */

        DeletePolicyType deletePolicyGet ()
                raises (ApplicationError);

        /** This method may be used to generically get all of the
```

```
            attributes supported by an instance.  Each interface is
            expected to sub-class the Managed Object value type and add the
            other attributes supported by that interface.  The managed
            object must return a value object of that type.  The client
            must then narrow the reference to access all the attributes.
            <p>

            The client may also submit a list of names indicating the
            attributes it wishes to receive.  These names must match the
            member names in the value object.  For members not on the list,
            and for members that are part of packages that are not
            supported, the server may return any value but it should be as
            short as possible.  The server also returns the list of
            attributes, which may be shorter due to exclusion of attributes
            in unsupported packages.  The client must regard the value of
            any member not in the list as garbage. <p>

            A null attribute names list indicates that all supported
            attributes are to be returned.  The server must return the
            actual list.  */

            ManagedObjectValueType attributesGet (
                    inout StringSetType attributeNames)
                    raises (ApplicationError);

            /** This method destroys the object.  It is used to simply
            release any resources associated with the managed object.  It
            does not check for contained objects or remove name bindings
            from the naming tree. <p>

            The intent of this operation is to allow support services to
            destroy the managed object.  <p><b>

            NOTE:  Direct invocation of this operation from a managing
            system could corrupt the naming tree and is recommended only
            under extraordinary circumstances.  Clients wishing to delete
            an object should instead use the terminator service. </b>  */

            void destroy()
                    raises (ApplicationError, DeleteError);

    }; // end of ManagedObject interface
```

// MANAGED OBJECT FACTORY INTERFACE

```
    /** This interface defines the generic managed object factory
    interface.  All Managed Object factories should inherit from this
    interface.  <p>

    In addition to providing the means for creating objects by management
    operation, the factories are assumed to take responsibility for
    maintaining the integrity of the naming tree by creating name bindings
    for the objects they create.  <p>

    Currently, this interface is null.  It is included, however, as a
    placeholder for capabilities that must be supported by all managed
    object factories.
    */

    interface ManagedObjectFactory {
```

```
}; // end of ManagedObjectFactory interface
```

// NOTIFICATIONS INTERFACE

```
/** This interface contains the definitions of notifications emitted by
many managed objects. <p>

The use of "typed" notifications is done here so that the notifications
can be documented in IDL and to support typed notifications for those
manager and managing systems that wish to use them.  Note that the
OMG's Notification Service supports both structured and typed
notifications.  It is not clear if implementations of the Notification
Service will support translation between them.  It is expected that the
implementation agreement between the managing and managed system will
specify the use of structured or typed notifications.  <p>

Notification users wishing to use typed notifications need only support
the interfaces below.  Notification publishers and subscribers wishing
to use structured notifications based on the operations defined below
should follow these rules for constructing and reading the notification
structure: <ul><li>

The domain_type string in the fixed header of the structure should be
set to "telecommunications".  <li>

The event_type string in the fixed header of the structure should be
set to the scoped name of the operation.  For example, for the
Attribute Value Change notification defined below this field would be
"itut_x780::Notifications::attributeValueChange".  <li>

The event_name string in the fixed header of the structure should be
null. <li>

Optional header fields may be included to support features like Quality
of Service as appropriate.  <li>

Each parameter in the operation should be placed in a name-value pair
in the filterable body portion of the notification.  The fd_name string
of this pair shall be set to the name of the parameter and the type
placed in the associated fd_value will be the type specified for the
parameter.  For example, each of the notifications defined below has a
parameter named “eventTime” that is an “ExternalTimeType.”  This
parameter would be placed in the filterable data portion of the event.
The fd_name string of this pair would be set to "eventTime" and
fd_value would contain an ExternalTimeType value.  <li>

The remainder of the body of the notification (the unfilterable part)
should be null.  </ul>

Unfortunately, typed notifications are mapped to notification
structures differently, so if one system wants to use typed
notifications and the other structured, the structured notification
user must be aware of how the CORBA Notification Service translates
typed notifications to structured notifications.  See the specification
for details.  In short, however, each of the parameters in the
operations below will be converted into a name-value pair in the
filterable data protion of the structured notification.  Also, the
event_type field in the fixed header of the structured notification
will be set to the special value "%TYPED" and the domain_type field
will be an empty string.  Finally, a name-value pair will be added as
the first element in the filterable data portion of the notification
with the name "operation".  The value associated with this name will be
```

```
    a string with the value set to the scoped name of the operation used to
    emit the notification
    (e.g. itut_x780::Notifications::attributeValueChange).  <p>

    Also, structured notification publishers may exclude notification
    parameters that are marked “optional” or are of an optional type (a
    type name ending in “TypeOpt.”  This should be done for efficiency.
    This will, however, preclude the automatic conversion of structured
    notifications to typed, so managers must be capable of accepting
    structured notifications. (They do not strictly have to support typed
    notifications, but if managed systems emit typed notifications managers
    should accept them rather than translations because it will be more
    efficient.)  If an “optional” parameter is included in a notification,
    the “optional” type (discriminated union) must be used.  <p>

    Parameters named "operation" should be avoided in notification
    operations to support the use of typed notifications.  While the
    notification channel should be able to differentiate the real parameter
    from the one added based on their positions in the filterable data
    list, it could have an impact on filtering as the default filtering
    language does not have a way to differentiate parameters based on
    position.  <p>

    Because the scoped operation name is placed in either the type_name
    string (when structured notifications are used) or a filterable body
    name-value pair with the name "operation" (when typed notifications are
    used), there is no "event type" parameter explicitly included in any of
    the notification data structures.  */

interface Notifications {

    /** An Attribute Value Change notification is used to report changes to
    the attributes of an object such as addition or deletion of members to
    one or more set-valued attributes and replacement of the value of one
    or more attributes.
    @param eventTime               Managed system's current time.
    @param source                  Object emitting notification.
    @param sourceClass             Actual class of source object.
    @param notificationIdentifier  A unique identifier for this
                                   notification.  Must be unique for
                                   an object instance.  (Optional in X.721
                                   but not here.  See text for
                                   discussion of possible implications)
    @param correlatedNotifications List of correlated notifications.
                                   Optional.  Zero length sequence
                                   indicates absence of this parameter.
    @param additionalText          Text message. Optional.  Zero length
                                   string indicates absence of this
                                   parameter.
    @param additionalInfo          Optional.  Zero length sequence
                                   indicates absence of this parameter.
    param sourceIndicator          Cause of event.  Optional.  Use
                                   "unknown" if not supported.
    @param attributeChanges        Changed attributes
    */

    void attributeValueChange (
            in ExternalTimeType                  eventTime,
            in NameType                          source,
            in ObjectClassType                   sourceClass,
            in NotifIDType                       notificationIdentifier,
            in CorrelatedNotificationSetType     correlatedNotifications,
            in AdditionalTextType                additionalText,
```

```
        in AdditionalInformationSetType         additionalInfo,
        in SourceIndicatorType                  sourceIndicator,
        in AttributeChangeSetType               attributeChanges
);

/** A Communications Alarm notification is used to report when an
object detects a communications error.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional.  Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional.  See type for details.
@param thresholdInfo            Optional.  See type for details.
@param stateChangeDefinition    Optional.  Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional.  Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional.  Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/

void communicationsAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType       correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
```

```
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);

/** An Environmental Alarm notification is used to report a problem in
the environment.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional.  Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional.  See type for details.
@param thresholdInfo            Optional.  See type for details.
@param stateChangeDefinition    Optional.  Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional.  Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional.  Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/

void environmentalAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType       correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
```

```
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);

/** An Equipment Alarm notification is used to report a failure in the
equipment.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param probableCause
@param specificProblems         Optional.  Zero length sequence
                                indicates absence of this parameter.
@param perceivedSeverity
@param backedUpStatus           "True" if backed up
@param backUpObject             Will be null if backedUpStatus is
                                "false"
@param trendIndication          Optional.  See type for details.
@param thresholdInfo            Optional.  See type for details.
@param stateChangeDefinition    Optional.  Zero length sequence
                                indicates absence of this parameter.
@param monitoredAttributes      Optional.  Zero length sequence
                                indicates absence of this parameter.
@param proposedRepairActions    Optional.  Zero length sequence
                                indicates absence of this parameter.
@param alarmEffectOnService     True if alarm is service effecting.
@param alarmingResumed          True if alarming was just resumed,
                                possibly resulting in delayed reporting
                                of an alarm
@param suspectObjectList        Objects possibly involved in failure.
*/

void equipmentAlarm (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType       correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in ProbableCauseType                    probableCause,
        in SpecificProblemSetType               specificProblems,
        in PerceivedSeverityType                perceivedSeverity,
        in BooleanTypeOpt                       backedUpStatus,
        in NameType                             backUpObject,
        in TrendIndicationTypeOpt               trendIndication,
        in ThresholdInfoType                    thresholdInfo,
        in AttributeChangeSetType               stateChangeDefinition,
        in AttributeSetType                     monitoredAttributes,
        in ProposedRepairActionSetType          proposedRepairActions,
```

```
        in BooleanTypeOpt                       alarmEffectOnService,
        in BooleanTypeOpt                       alarmingResumed,
        in SuspectObjectSetType                 suspectObjectList
);

/** An Integrity Violation notification is used to report that a
potential interruption in information flow has occurred such that
information may have been illegally modified, inserted or deleted.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed?  X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/

void integrityViolation (
        in ExternalTimeType                     eventTime,
        in NameType                             source,
        in ObjectClassType                      sourceClass,
        in NotifIDType                          notificationIdentifier,
        in CorrelatedNotificationSetType      correlatedNotifications,
        in AdditionalTextType                   additionalText,
        in AdditionalInformationSetType         additionalInfo,
        in SecurityAlarmCauseType               securityAlarmCause,
        in PerceivedSeverityType                securityAlarmSeverity,
        in SecurityAlarmDetectorType            securityAlarmDetector,
        in ServiceUserType                      serviceUser,
        in ServiceProviderType                  serviceProvider
);

/** An Object Creation notification is used to report the creation of a
managed object to another open system. Note that the source field
should be set to the created object, not the factory.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
```

```
                                    string indicates absence of this
                                    parameter.
        @param additionalInfo       Optional.  Zero length sequence
                                    indicates absence of this parameter.
        @param sourceIndicator      Cause of event.  Optional. Use
                                    "unknown" if not supported.
        @param attributeSet         Attribute values. Optional. Zero length
                                    sequence indicates absence of this
                                    parameter.
        */

        void objectCreation (
                in ExternalTimeType                 eventTime,
                in NameType                         source,
                in ObjectClassType                  sourceClass,
                in NotifIDType                      notificationIdentifier,
                in CorrelatedNotificationSetType    correlatedNotifications,
                in AdditionalTextType               additionalText,
                in AdditionalInformationSetType     additionalInfo,
                in SourceIndicatorType              sourceIndicator,
                in AttributeSetType                 attributeList
        );

        /** An Object Deletion notification is used to report the deletion of a
        managed object.  Note that the source field should be set to
        the object being deleted.
        @param eventTime            Managed system's current time.
        @param source               Object emitting notification.
        @param sourceClass          Actual class of source object.
        @param notificationIdentifier   A unique identifier for this
                                    notification.  Must be unique for
                                    an object instance.  (Optional in X.721
                                    but not here.  See text for
                                    discussion of possible implications)
        @param correlatedNotifications  List of correlated notifications.
                                    Optional.  Zero length sequence
                                    indicates absence of this parameter.
        @param additionalText       Text message. Optional.  Zero length
                                    string indicates absence of this
                                    parameter.
        @param additionalInfo       Optional.  Zero length sequence
                                    indicates absence of this parameter.
        @param sourceIndicator      Cause of event.  Optional. Use
                                    "unknown" if not supported.
        @param attributeSet         Attribute values. Optional. Zero length
                                    sequence indicates absence of this
                                    parameter.
        */

        void objectDeletion (
                in ExternalTimeType                 eventTime,
                in NameType                         source,
                in ObjectClassType                  sourceClass,
                in NotifIDType                      notificationIdentifier,
                in CorrelatedNotificationSetType    correlatedNotifications,
                in AdditionalTextType               additionalText,
                in AdditionalInformationSetType     additionalInfo,
                in SourceIndicatorType              sourceIndicator,
                in AttributeSetType                 attributeList
        );

        /** An Operational Violation notification is used to report that the
        provision of the requested service was not possible due to the
```

```
unavailability, malfunction or incorrect invocation of the service.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed?  X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/

void operationalViolation (
        in ExternalTimeType                  eventTime,
        in NameType                          source,
        in ObjectClassType                   sourceClass,
        in NotifIDType                       notificationIdentifier,
        in CorrelatedNotificationSetType    correlatedNotifications,
        in AdditionalTextType                additionalText,
        in AdditionalInformationSetType      additionalInfo,
        in SecurityAlarmCauseType            securityAlarmCause,
        in PerceivedSeverityType             securityAlarmSeverity,
        in SecurityAlarmDetectorType         securityAlarmDetector,
        in ServiceUserType                   serviceUser,
        in ServiceProviderType               serviceProvider
);

/** A Physical Violation notification is used to report that a physical
resource has been violated in a way that indicates a potential security
attack.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed?  X.721 appears to
                                restrict the "cleared" value on this
```

```
                                    alarm but clears should be allowed.
        @param securityAlarmDetector
        @param serviceUser
        @param serviceProvider
        */

        void physicalViolation (
                in ExternalTimeType                     eventTime,
                in NameType                             source,
                in ObjectClassType                      sourceClass,
                in NotifIDType                          notificationIdentifier,
                in CorrelatedNotificationSetType       correlatedNotifications,
                in AdditionalTextType                   additionalText,
                in AdditionalInformationSetType         additionalInfo,
                in SecurityAlarmCauseType               securityAlarmCause,
                in PerceivedSeverityType                securityAlarmSeverity,
                in SecurityAlarmDetectorType            securityAlarmDetector,
                in ServiceUserType                      serviceUser,
                in ServiceProviderType                  serviceProvider
        );

        /** A Processing Error Alarm notification is used to report a
        processing failure in a managed object.
        @param eventTime                Managed system's current time.
        @param source                   Object emitting notification.
        @param sourceClass              Actual class of source object.
        @param notificationIdentifier   A unique identifier for this
                                        notification.  Must be unique for
                                        an object instance.  (Optional in X.721
                                        but not here.  See text for
                                        discussion of possible implications)
        @param correlatedNotifications  List of correlated notifications.
                                        Optional.  Zero length sequence
                                        indicates absence of this parameter.
        @param additionalText           Text message. Optional.  Zero length
                                        string indicates absence of this
                                        parameter.
        @param additionalInfo           Optional.  Zero length sequence
                                        indicates absence of this parameter.

        @param probableCause
        @param specificProblems         Optional.  Zero length sequence
                                        indicates absence of this parameter.

        @param perceivedSeverity
        @param backedUpStatus           "True" if backed up
        @param backUpObject             Will be null if backedUpStatus is
                                        "false"
        @param trendIndication          Optional.  See type for details.
        @param thresholdInfo            Optional.  See type for details.
        @param stateChangeDefinition    Optional.  Zero length sequence
                                        indicates absence of this parameter.
        @param monitoredAttributes      Optional.  Zero length sequence
                                        indicates absence of this parameter.
        @param proposedRepairActions    Optional.  Zero length sequence
                                        indicates absence of this parameter.
        @param alarmEffectOnService     True if alarm is service effecting.
        @param alarmingResumed          True if alarming was just resumed,
                                        possibly resulting in delayed reporting
                                        of an alarm
        @param suspectObjectList        Objects possibly involved in failure.
        */

        void processingErrorAlarm (
                in ExternalTimeType                     eventTime,
```

```
        in NameType                          source,
        in ObjectClassType                   sourceClass,
        in NotifIDType                       notificationIdentifier,
        in CorrelatedNotificationSetType    correlatedNotifications,
        in AdditionalTextType                additionalText,
        in AdditionalInformationSetType      additionalInfo,
        in ProbableCauseType                 probableCause,
        in SpecificProblemSetType            specificProblems,
        in PerceivedSeverityType             perceivedSeverity,
        in BooleanTypeOpt                    backedUpStatus,
        in NameType                          backUpObject,
        in TrendIndicationTypeOpt            trendIndication,
        in ThresholdInfoType                 thresholdInfo,
        in AttributeChangeSetType            stateChangeDefinition,
        in AttributeSetType                  monitoredAttributes,
        in ProposedRepairActionSetType       proposedRepairActions,
        in BooleanTypeOpt                    alarmEffectOnService,
        in BooleanTypeOpt                    alarmingResumed,
        in SuspectObjectSetType              suspectObjectList
);

/** A Quality of Service Alarm notification is used to report a failure
in the quality of service of the managed object.
@param eventTime               Managed system's current time.
@param source                  Object emitting notification.
@param sourceClass             Actual class of source object.
@param notificationIdentifier  A unique identifier for this
                               notification.  Must be unique for
                               an object instance.  (Optional in X.721
                               but not here.  See text for
                               discussion of possible implications)
@param correlatedNotifications List of correlated notifications.
                               Optional.  Zero length sequence
                               indicates absence of this parameter.
@param additionalText          Text message. Optional.  Zero length
                               string indicates absence of this
                               parameter.
@param additionalInfo          Optional.  Zero length sequence
                               indicates absence of this parameter.

@param probableCause
@param specificProblems        Optional.  Zero length sequence
                               indicates absence of this parameter.

@param perceivedSeverity
@param backedUpStatus          "True" if backed up
@param backUpObject            Will be null if backedUpStatus is
                               "false"
@param trendIndication         Optional.  See type for details.
@param thresholdInfo           Optional.  See type for details.
@param stateChangeDefinition   Optional.  Zero length sequence
                               indicates absence of this parameter.
@param monitoredAttributes     Optional.  Zero length sequence
                               indicates absence of this parameter.
@param proposedRepairActions   Optional.  Zero length sequence
                               indicates absence of this parameter.
@param alarmEffectOnService    True if alarm is service effecting.
@param alarmingResumed         True if alarming was just resumed,
                               possibly resulting in delayed reporting
                               of an alarm
@param suspectObjectList       Objects possibly involved in failure.
*/

void qualityOfServiceAlarm (
        in ExternalTimeType                  eventTime,
```

```
        in NameType                          source,
        in ObjectClassType                   sourceClass,
        in NotifIDType                       notificationIdentifier,
        in CorrelatedNotificationSetType    correlatedNotifications,
        in AdditionalTextType                additionalText,
        in AdditionalInformationSetType      additionalInfo,
        in ProbableCauseType                 probableCause,
        in SpecificProblemSetType            specificProblems,
        in PerceivedSeverityType             perceivedSeverity,
        in BooleanTypeOpt                    backedUpStatus,
        in NameType                          backUpObject,
        in TrendIndicationTypeOpt            trendIndication,
        in ThresholdInfoType                 thresholdInfo,
        in AttributeChangeSetType            stateChangeDefinition,
        in AttributeSetType                  monitoredAttributes,
        in ProposedRepairActionSetType       proposedRepairActions,
        in BooleanTypeOpt                    alarmEffectOnService,
        in BooleanTypeOpt                    alarmingResumed,
        in SuspectObjectSetType              suspectObjectList
);

/** A Relationship Change notification is used to report the change in
the value of one or more relationship attributes of a managed object,
that result through either internal operation of the managed object or
via management operation.
@param eventTime               Managed system's current time.
@param source                  Object emitting notification.
@param sourceClass             Actual class of source object.
@param notificationIdentifier  A unique identifier for this
                               notification.  Must be unique for
                               an object instance.  (Optional in X.721
                               but not here.  See text for
                               discussion of possible implications)
@param correlatedNotifications List of correlated notifications.
                               Optional.  Zero length sequence
                               indicates absence of this parameter.
@param additionalText          Text message. Optional.  Zero length
                               string indicates absence of this
                               parameter.
@param additionalInfo          Optional.  Zero length sequence
                               indicates absence of this parameter.
@param sourceIndicator         Cause of event.  Optional. Use
                               "unknown" if not supported.
@param relationshipChanges     Changed relationship attributes
*/

void relationshipChange (
        in ExternalTimeType                  eventTime,
        in NameType                          source,
        in ObjectClassType                   sourceClass,
        in NotifIDType                       notificationIdentifier,
        in CorrelatedNotificationSetType    correlatedNotifications,
        in AdditionalTextType                additionalText,
        in AdditionalInformationSetType      additionalInfo,
        in SourceIndicatorType               sourceIndicator,
        in AttributeChangeSetType            relationshipChanges
);

/** A Security Violation notification is used to report that a security
attack has been detected by a security service or mechanism.
@param eventTime               Managed system's current time.
@param source                  Object emitting notification.
@param sourceClass             Actual class of source object.
```

```
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param securityAlarmCause
@param securityAlarmSeverity    Clears allowed?  X.721 appears to
                                restrict the "cleared" value on this
                                alarm but clears should be allowed.
@param securityAlarmDetector
@param serviceUser
@param serviceProvider
*/

void securityViolation (
        in ExternalTimeType                  eventTime,
        in NameType                          source,
        in ObjectClassType                   sourceClass,
        in NotifIDType                       notificationIdentifier,
        in CorrelatedNotificationSetType    correlatedNotifications,
        in AdditionalTextType                additionalText,
        in AdditionalInformationSetType      additionalInfo,
        in SecurityAlarmCauseType            securityAlarmCause,
        in PerceivedSeverityType             securityAlarmSeverity,
        in SecurityAlarmDetectorType         securityAlarmDetector,
        in ServiceUserType                   serviceUser,
        in ServiceProviderType               serviceProvider
);

/** A State Change notification is used to report the change in the the
value of one or more state attributes of a managed object, that result
through either internal operation of the managed object or via
management operation.
@param eventTime                Managed system's current time.
@param source                   Object emitting notification.
@param sourceClass              Actual class of source object.
@param notificationIdentifier   A unique identifier for this
                                notification.  Must be unique for
                                an object instance.  (Optional in X.721
                                but not here.  See text for
                                discussion of possible implications)
@param correlatedNotifications  List of correlated notifications.
                                Optional.  Zero length sequence
                                indicates absence of this parameter.
@param additionalText           Text message. Optional.  Zero length
                                string indicates absence of this
                                parameter.
@param additionalInfo           Optional.  Zero length sequence
                                indicates absence of this parameter.
@param sourceIndicator          Cause of event.  Optional. Use
                                "unknown" if not supported.
@param stateChanges             Changed state attributes.
*/

void stateChange (
```

```
            in ExternalTimeType                    eventTime,
            in NameType                            source,
            in ObjectClassType                     sourceClass,
            in NotifIDType                         notificationIdentifier,
            in CorrelatedNotificationSetType       correlatedNotifications,
            in AdditionalTextType                  additionalText,
            in AdditionalInformationSetType        additionalInfo,
            in SourceIndicatorType                 sourceIndicator,
            in AttributeChangeSetType              stateChanges
    );

    /** A Time Domain Violation notification is used to report that an
    event has occurred at an unexpected or prohibited time.
    @param eventTime                Managed system's current time.
    @param source                   Object emitting notification.
    @param sourceClass              Actual class of source object.
    @param notificationIdentifier   A unique identifier for this
                                    notification.  Must be unique for
                                    an object instance.  (Optional in X.721
                                    but not here.  See text for
                                    discussion of possible implications)
    @param correlatedNotifications  List of correlated notifications.
                                    Optional.  Zero length sequence
                                    indicates absence of this parameter.
    @param additionalText           Text message. Optional.  Zero length
                                    string indicates absence of this
                                    parameter.
    @param additionalInfo           Optional.  Zero length sequence
                                    indicates absence of this parameter.
    @param securityAlarmCause
    @param securityAlarmSeverity    Clears allowed?  X.721 appears to
                                    restrict the "cleared" value on this
                                    alarm but clears should be allowed.
    @param securityAlarmDetector
    @param serviceUser
    @param serviceProvider
    */

    void timeDomainViolation (
            in ExternalTimeType                    eventTime,
            in NameType                            source,
            in ObjectClassType                     sourceClass,
            in NotifIDType                         notificationIdentifier,
            in CorrelatedNotificationSetType       correlatedNotifications,
            in AdditionalTextType                  additionalText,
            in AdditionalInformationSetType        additionalInfo,
            in SecurityAlarmCauseType              securityAlarmCause,
            in PerceivedSeverityType               securityAlarmSeverity,
            in SecurityAlarmDetectorType           securityAlarmDetector,
            in ServiceUserType                     serviceUser,
            in ServiceProviderType                 serviceProvider
    );

    /** These constants define the names of the notifications declared
    above and are provided to help reduce errors.  */

    const string attributeValueChangeTypeName =
            "itut_x780::Notifications::attributeValueChange";
    const string communicationsAlarmTypeName =
            "itut_x780::Notifications::communicationsAlarm";
    const string environmentalAlarmTypeName =
            "itut_x780::Notifications::environmentalAlarm";
    const string equipmentAlarmTypeName =
```

```
            "itut_x780::Notifications::equipmentAlarm";
        const string integrityViolationTypeName =
            "itut_x780::Notifications::integrityViolation";
        const string objectCreationTypeName =
            "itut_x780::Notifications::objectCreation";
        const string objectDeletionTypeName =
            "itut_x780::Notifications::objectDeletion";
        const string operationalViolationTypeName =
            "itut_x780::Notifications::operationalViolation";
        const string physicalViolationTypeName =
            "itut_x780::Notifications::physicalViolation";
        const string processingErrorAlarmTypeName =
            "itut_x780::Notifications::processingErrorAlarm";
        const string qualityOfServiceAlarmTypeName =
            "itut_x780::Notifications::qualityOfServiceAlarm";
        const string relationshipChangeTypeName =
            "itut_x780::Notifications::relationshipChange";
        const string securityViolationTypeName =
            "itut_x780::Notifications::securityViolation";
        const string stateChangeTypeName =
            "itut_x780::Notifications::stateChange";
        const string timeDomainViolationTypeName =
            "itut_x780::Notifications::timeDomainViolation";

        /** These constants define the names of the parameters used in the
        notifications declared above and are provided to help reduce errors.
        */

        const string additionalInfoName = "additionalInfo";
        const string additionalTextName = "additionalText";
        const string alarmEffectOnServiceName = "alarmEffectOnService";
        const string alarmingResumedName = "alarmingResumed";
        const string attributeChangesName = "attributeChanges";
        const string attributeListName = "attributeList";
        const string backedUpStatusName = "backedUpStatus";
        const string backUpObjectName = "backUpObject";
        const string correlatedNotificationsName = "correlatedNotifications";
        const string eventTimeName = "eventTime";
        const string monitoredAttributesName = "monitoredAttributes";
        const string notificationIdentifierName = "notificationIdentifier";
        const string perceivedSeverityName = "perceivedSeverity";
        const string probableCauseName = "probableCause";
        const string proposedRepairActionsName = "proposedRepairActions";
        const string relationshipChangesName = "relationshipChanges";
        const string securityAlarmCauseName = "securityAlarmCause";
        const string securityAlarmDetectorName = "securityAlarmDetector";
        const string securityAlarmSeverityName = "securityAlarmSeverity";
        const string serviceProviderName = "serviceProvider";
        const string serviceUserName = "serviceUser";
        const string sourceName = "source";
        const string sourceClassName = "sourceClass";
        const string sourceIndicatorName = "sourceIndicator";
        const string specificProblemsName = "specificProblems";
        const string stateChangeDefinitionName = "stateChangeDefinition";
        const string stateChangesName = "stateChanges";
        const string suspectObjectListName = "suspectObjectList";
        const string thresholdInfoName = "thresholdInfo";
        const string trendIndicationName = "trendIndication";

    }; // end of Notifications interface

}; // end of itut_x780 module
```

// MACROS

```
/* The following macros are provided for quickly and concisely defining
the notifications to be supported by an object.  Example usage (within an
interface):

MANDATORY_NOTIFICATION(itut_x780::Notifications, objectCreation);
CONDITIONAL_NOTIFICATION(itut_x780::Notifications, stateChange, statePackage);

The macros simply expand into nothing, as CORBA IDL doesn't really have
anything for them to expand into that makes sense. Eventually, these
may be changed to expand into IDL supporting the CORBA Component Model.
*/

#undef  MANDATORY_NOTIFICATION
#define MANDATORY_NOTIFICATION(InterfaceName, NotificationName)

#undef  CONDITIONAL_NOTIFICATION
#define CONDITIONAL_NOTIFICATION(InterfaceName, NotificationName, PackageName)

#endif // end of ifndef itut_x780_IDL
```

Annex B Network Management Constant Definitions

(Normative)

```
/* This IDL code is intended to be stored in a file named “itut_x780Const.idl”
and located in the same directory as the file containing Annex A */

#ifndef ITUT_X780Const_IDL
#define ITUT_X780Const_IDL

#pragma prefix “itu.int”

module itut_x780 {
```

// ApplicationErrorConst Module

```
/** This module contains the constants defined for the error code contained in
Application Error Info structures returned with Application Error exceptions.
*/

module ApplicationErrorConst {

        const string moduleName = "itut_x780::ApplicationErrorConst";

        /** This application error exception code indicates the operation
            failed due to a problem downstream from the managed system,
            possibly a communication problem between the managed system
            and the resource */

        const short downstreamError = 1;

        /** An application error exception returining this code will return
            the name of the offending paramter in the details field. */

        const short invalidParameter = 2;

        /** This application error exception code indicates the operation
            failed due to a transient problem on the managed system. */

        const short resourceLimit = 3;

}; // end of module ApplicationErrorConst
```

// CreateErrorConst Module

```
/** This module contains the constants defined for the error code contained in
Create Error Info structures returned with Create Error exceptions.
*/

module CreateErrorConst {

        const string moduleName = "itut_x780::CreateErrorConst";


        /** This create error exception code indicates that the name included
            in the create operation is not valid. */

        const short badName = 1;

        /** This create error exception code indicates that the name included
```

```
            in the create operation is a duplicate. */

        const short duplicateName = 2;

        /** This create error exception code indicates some packages requested
            in the create operation are incompatible with each other.  It must
            be included in a PackageErrorInfoType structure (subclass of
            CreateErrorInfoType).  The packages list contains the names of the
            unsupported packages.  */

        const short incompatiblePackages = 3;

        /** This create error exception code indicates that the name binding
            referenced in the create operation is not valid. */

        const short invalidNameBinding = 4;

        /** This create error exception code indicates a package requested in
            the create operation is not supported.  It must be included in a
            PackageErrorInfoType structure (subclass of CreateErrorInfoType).
            The packages list contains the names of the unsupported packages.
            */

        const short unsupportedPackages = 5;

}; // end of module CreateErrorConst
```

// DeleteErrorConst Module

```
/** This module contains the constants defined for the error code contained in
Delete Error Info structures returned with Delete Error exceptions.
*/

module DeleteErrorConst {

        const string moduleName = "itut_x780::DeleteErrorConst";

        /** This delete error exceptin code indicates the object has both
            subordinates and a delete policy of deleteOnlyIfNoContained.  */

        const short containsObjects = 1;

        /** This delete error exception code indicates the object has a delete
            policy of notDeletable, and cannot be deleted.  */

        const short notDeletable = 2;

        /** This delete error exception code indicates the object had a
            subordinate object that could not be deleted, so the superior
            object(s) could not be deleted.  */

        const short undeletableContainedObject = 3;

        /** This delete error exception code indicates the object is in
            a state in which it cannot be deleted.  */

        const short invalidStateForDestroy = 4;

}; // end of module DeleteErrorConst
```

// ProbableCauseConst Module

```
/** This module contains the constant values defined for the
ProbableCause UID.  These values were borrowed from X.721. */

module ProbableCauseConst {
const string moduleName = "itut_x780::ProbableCauseConst";

        const short indeterminate = 0;
        const short adapterError = 1;
        const short applicationSubsystemFailure = 2;
        const short bandwidthReduced = 3;
        const short callEstablishmentError = 4;
        const short communicationsProtocolError = 5;
        const short communicationsSubsystemFailure = 6;
        const short configurationOrCustomizationError = 7;
        const short congestion = 8;
        const short corruptData = 9;
        const short cpuCyclesLimitExceeded = 10;
        const short dataSetOrModemError = 11;
        const short degradedSignal = 12;
        const short dTE_DCEInterfaceError = 13;
        const short enclosureDoorOpen = 14;
        const short equipmentMalfunction = 15;
        const short excessiveVibration = 16;
        const short fileError = 17;
        const short fireDetected = 18;
        const short floodDetected = 19;
        const short framingError = 20;
        const short heatingOrVentilationOrCoolingSystemProblem = 21;
        const short humidityUnacceptable = 22;
        const short inputOutputDeviceError = 23;
        const short inputDeviceError = 24;
        const short lANError = 25;
        const short leakDetected = 26;
        const short localNodeTransmissionError = 27;
        const short lossOfFrame = 28;
        const short lossOfSignal = 29;
        const short materialSupplyExhausted = 30;
        const short multiplexerProblem = 31;
        const short outOfMemory = 32;
        const short ouputDeviceError = 33;
        const short performanceDegraded = 34;
        const short powerProblem = 35;
        const short pressureUnacceptable = 36;
        const short processorProblem = 37;
        const short pumpFailure = 38;
        const short queueSizeExceeded = 39;
        const short receiveFailure = 40;
        const short receiverFailure = 41;
        const short remoteNodeTransmissionError = 42;
        const short resourceAtOrNearingCapacity = 43;
        const short responseTimeExcessive = 44;
        const short retransmissionRateExcessive = 45;
        const short softwareError = 46;
        const short softwareProgramAbnormallyTerminated = 47;
        const short softwareProgramError = 48;
        const short storageCapacityProblem = 49;
        const short temperatureUnacceptable = 50;
        const short thresholdCrossed = 51;
        const short timingProblem = 52;
        const short toxicLeakDetected = 53;
        const short transmitFailure = 54;
```

```
        const short transmitterFailure = 55;
        const short underlyingResourceUnavailable = 56;
        const short versionMismatch = 57;

}; // end of ProbableCauseConst module


// SecurityAlarmCauseConst Module

/** This module contains the constant values defined for the
SecurityAlarmCause UID.  These values were borrowed from
X.721. */

module SecurityAlarmCauseConst {
const string moduleName = "itut_x780::SecurityAlarmCauseConst";

        const short authenticationFailure = 1;
        const short breachOfConfidentiality = 2;
        const short cableTamper = 3;
        const short delayedInformation = 4;
        const short denialOfService = 5;
        const short duplicateInformation = 6;
        const short informationMissing = 7;
        const short informationModificationDetected = 8;
        const short informationOutOfSequence = 9;
        const short intrusionDetection = 10;
        const short keyExpired = 11;
        const short nonRepudiationFailure = 12;
        const short outOfHoursActivity = 13;
        const short outOfService = 14;
        const short proceduralError = 15;
        const short unauthorizedAccessAttempt = 16;
        const short unexpectedInformation = 17;
        const short unspecifiedReason = 18;

}; // end of SecurityAlarmCauseConst module


}; // end of itut_x780 module

#endif // end of ifndef ITUT_X780Const_IDL
```