International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.780.2

(03/2007)

SERIES X: DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

OSI management – Management functions and ODMA functions

TMN guidelines for defining service-oriented CORBA managed objects and façade objects

ITU-T Recommendation X.780.2

ITU-T X-SERIES RECOMMENDATIONS

DATA NETWORKS, OPEN SYSTEM COMMUNICATIONS AND SECURITY

PUBLIC DATA NETWORKS	
Services and facilities	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalling and switching	X.50–X.89
Network aspects	X.90–X.149
Maintenance	X.150–X.179
Administrative arrangements	X.180–X.199
OPEN SYSTEMS INTERCONNECTION	
Model and notation	X.200–X.209
Service definitions	X.210–X.219
Connection-mode protocol specifications	X.220–X.229
Connectionless-mode protocol specifications	X.230–X.239
PICS proformas	X.240–X.259
Protocol Identification	X.260–X.269
Security Protocols	X.270–X.279
Layer Managed Objects	X.280–X.289
Conformance testing	X.290–X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300–X.349
Satellite data transmission systems	X.350–X.369
IP-based networks	X.370–X.379
MESSAGE HANDLING SYSTEMS	X.400–X.499
DIRECTORY	X.500–X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600–X.629
Efficiency	X.630–X.639
Quality of service	X.640–X.649
Naming, Addressing and Registration	X.650–X.679
Abstract Syntax Notation One (ASN.1)	X.680–X.699
OSI MANAGEMENT	
Systems Management framework and architecture	X.700–X.709
Management Communication Service and Protocol	X.710–X.719
Structure of Management Information	X.720–X.729
Management functions and ODMA functions	**X.730–X.799**
SECURITY	X.800–X.849
OSI APPLICATIONS	
Commitment, Concurrency and Recovery	X.850–X.859
Transaction processing	X.860–X.879
Remote operations	X.880–X.889
Generic applications of ASN.1	X.890–X.899
OPEN DISTRIBUTED PROCESSING	X.900–X.999
TELECOMMUNICATION SECURITY	X.1000–

For further details, please refer to the list of ITU-T Recommendations.

ITU-T Recommendation X.780.2

TMN guidelines for defining service-oriented CORBA managed objects and façade objects

Summary

This Recommendation defines a set of TMN CORBA managed object and façade object modelling guidelines required to support service-oriented interfaces. It specifies how service-oriented CORBA TMN interfaces are to be defined. It covers IDL repertoire, information modelling in IDL and IDL style conventions. It also provides guidelines on redesigning fine-grained and coarse-grained interfaces to service-oriented interfaces. An IDL module is provided defining basic data types, exceptions, notification constructs, a base CORBA struct to be included by every service-oriented managed object and a base CORBA interface to be inherited by every service-oriented CORBA interface. This Recommendation and ITU-T Recommendation Q.816.2 together compose a framework for CORBA-based service-oriented TMN interfaces with a wide range of applications.

Source

ITU-T Recommendation X.780.2 was approved on 16 March 2007 by ITU-T Study Group 4 (2005-2008) under the ITU-T Recommendation A.8 procedure.

Keywords

Common Object Request Broker Architecture (CORBA), CORBA services, distributed processing, façade, Interface Definition Language (IDL), lightweight object, managed object, managed system, managing system, service orientation, service-oriented façade object, TMN interface.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. 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 Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 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.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure e.g. interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. 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, ITU 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 at http://www.itu.int/ITU-T/ipr/.

CONTENTS

		Page
1	Scope	1
1.1	Purpose	2
1.2	Application	2
2	References	2
3	Definitions	4
3.1	Terms defined elsewhere	4
4	Abbreviations and acronyms	5
5	Conventions	8
6	Service-oriented interface design considerations	8
6.1	Flexible use of façade design pattern	9
6.2	Lightweight use of ORB	10
6.3	Naming of managed objects and service-oriented façade objects	11
6.4	Creation and deletion of objects	11
6.5	Inheritance and containment	12
6.6	Exceptions	12
6.7	Retrieval and modification of objects and attributes	12
6.8	Notifications	12
6.9	Conditional capabilities and supported capabilities	12
6.10	Additional service-oriented interface design considerations	13
7	Service-oriented framework and requirements overview	13
7.1	Framework overview	13
7.2	Framework constituents overview	13
8	Service-oriented object model IDL	16
8.1	IDL repertoire and foundation IDL	17
8.2	Use of name/value pairs with string values	22
8.3	Single-valued, multi-valued and tagged parameters	22
8.4	Modelling of transmission technologies	24
8.5	Common attributes `Common_T` of service-oriented managed objects	25
8.6	Exceptions	27
8.7	Notifications	28
9	Providing service-oriented façade interfaces	49
9.1	Façade instantiation	49
9.2	Getting and setting objects and attributes	50
9.3	Service-oriented façade interface base class `Common_I`	52
9.4	Iterator interfaces	55

Page

10 Service-oriented CORBA modelling guidelines 59
10.1 Rules and conventions 61
10.2 Superclasses of service-oriented managed objects and façade objects 62
10.3 Naming conventions for managed objects and façade objects 62
10.4 Service-oriented modelling of transmission technologies 69
10.5 Modelling of IDL extensions 80
10.6 Additional CORBA modelling guidelines and principles 88
11 Style guide for lightweight CORBA IDL modelling 90
12 Guidelines for redesigning fine-grained and coarse-grained models to service-oriented 90
13 Service-oriented compliance and conformance 90
13.1 Standards document compliance 91
13.2 System conformance 91
13.3 Conformance statement guidelines 97
Annex A (Normative) – Service-oriented modelling IDL 98
A.1 Module `globaldefs` 100
A.2 Module `common` 105
A.3 Module `transmissionParameters` 110
A.4 Module `notifications` 112
Bibliography 132

ITU-T Recommendation X.780.2

TMN guidelines for defining service-oriented CORBA managed objects and façade objects

1 Scope

The TMN architecture defined in [ITU-T M.3010] introduces concepts from distributed processing and includes the use of multiple management protocols. [ITU-T Q.816] and [ITU-T X.780] subsequently define within this architecture a framework for applying the Common Object Request Broker Architecture (CORBA) as one of the TMN management protocols. [ITU-T Q.816] and [ITU-T X.780] use an approach where each manageable resource is addressable with a unique interoperable object reference (IOR) and have become known as "fine-grained" interfaces.

[ITU-T Q.816.1] and [ITU-T X.780.1] extend the CORBA-based TMN interface framework so that not every manageable resource needs to have an IOR. The extension has become known as "coarse-grained" interfaces. At a coarse-grained TMN interface, a CORBA object (with an IOR), which is used to access manageable resources having no IOR and invoke operations on them, is referred to as a façade, while an object that is accessed through a façade is referred to as a lightweight object. All managed objects that are accessible through a particular coarse-grained façade are required to be of the same class and, for each managed object class, (at least) one façade interface shall be provided. Managed objects accessed through a façade still have a CORBA name even though they may not (but nevertheless could) have an individual CORBA interface. To support the ability to determine a managed object's façade based on only its name, the CORBA names of managed objects accessible through a coarse-grained façade are extended slightly beyond the names of managed objects not accessible through a façade.

This Recommendation, along with [ITU-T Q.816.2], further extends the framework to enable it to support a different style of interaction between managing systems and managed systems. This style of interaction is termed "service-oriented". The "service-oriented" style introduces the so-called "service-oriented façade objects" in the managed systems and relieves a managing system from having to retrieve the location information for each manageable resource or type of manageable resource it wishes to access. This Recommendation provides the IDL-related service-oriented modelling guidelines, including the superclasses of the service-oriented managed objects and façade objects, rules and conventions, naming of managed objects and service-oriented façade objects, IDL repertoire and style guide for use by IDL modellers, modelling of transmission technologies according to [ITU-T Q.805] and [ITU-T G.809], and standardized modelling of IDL extensions. The companion Recommendation [ITU-T Q.816.2] specifies the lightweight use of built-in CORBA system services of ORB products and of certain OMG common object services, and introduces a new ITU-T TMN support service, the session service, which allows for the implementation of a very high degree of lightweight object.

The scope of this Recommendation is the same as the original TMN CORBA framework in the Q.816-series and X.780-series of ITU-T Recommendations. The framework and these extensions cover all interfaces in the TMN where CORBA may be used. These interfaces are OS-OS interfaces according to [ITU-T M.3010], where one OS takes a client/manager role (e.g., an NMS) and the other OS takes a server/agent role (e.g., an EMS). To be concrete, the service-oriented modelling IDL of Annex A refers to an NML-EML interface but it can be applied to any managing system and managed system. It is expected, however, that not all capabilities and services 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 at various logical layers) as well as between management systems and network elements.

1.1 Purpose

The purpose of this Recommendation and the companion Recommendation [ITU-T Q.816.2] is to extend the TMN CORBA framework to enable it to be used in a wider range of applications. The extensions enable a lightweight mode of interaction between the managing and managed systems which may be preferred in many situations. They also enable the endorsement of *de facto* CORBA services and information models usage which are recognized in the telecommunications industry. Thus, this Recommendation is intended for use by various groups specifying network management interfaces.

1.2 Application

[ITU-T Q.816.2] accompanies this Recommendation and extends the framework ORB usage, the OMG common services usage and the ITU-T TMN framework support services defined in [ITU-T Q.816] and [ITU-T Q.816.1]. Collectively, [ITU-T Q.816.2] and this Recommendation define a *framework* for CORBA-based service-oriented TMN interfaces. The fine-grained, coarse-grained and service-oriented choices of the CORBA-based TMN interface framework are used in Amendment 1 to [ITU-T M.3010] to provide conformance definitions for TMN interfaces between operations systems. Refer to [ITU-T Q.816.2] for details.

2 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; 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. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T G.805] ITU-T Recommendation G.805 (2000), *Generic functional architecture of transport networks*.

[ITU-T G.809] ITU-T Recommendation G.809 (2003), *Functional architecture of connectionless layer networks*.

[ITU-T G.852.2] ITU-T Recommendation G.852.2 (1999), *Enterprise viewpoint description of transport network resource model*.

[ITU-T G.8010] ITU-T Recommendation G.8010/Y.1306 (2004), *Architecture of Ethernet layer networks*.

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

[ITU-T M.3017] ITU-T Recommendation M.3017 (2003), *Framework for the integrated management of hybrid circuit/packet networks*.

[ITU-T M.3100] ITU-T Recommendation M.3100 (2005), *Generic network information model*.

[ITU-T Q.816] ITU-T Recommendation Q.816 (2001), *CORBA-based TMN services*.

[ITU-T Q.816.1] ITU-T Recommendation Q.816.1 (2001), *CORBA-based TMN services: Extensions to support coarse-grained interfaces*.

[ITU-T Q.816.2] ITU-T Recommendation Q.816.2 (2007), *CORBA-based TMN services: Extensions to support service-oriented interfaces*.

[ITU-T Q.821.1] ITU-T Recommendation Q.821.1 (2001), *CORBA-based TMN alarm surveillance service*.

[ITU-T Q.822] ITU-T Recommendation Q.822 (1994), *Stage 1, stage 2 and stage 3 description for the Q3 interface – Performance management*.

[ITU-T X.200] ITU-T Recommendation X.200 (1994) | ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model*.

[ITU-T X.721] ITU-T Recommendation X.721 (1992) | ISO/IEC 10165-2:1992, *Information technology – Open Systems Interconnection – Structure of management information: Definition of management information*.

[ITU-T X.733] ITU-T Recommendation X.733 (1992) | ISO/IEC 10164-4:1992, *Information technology – Open Systems Interconnection – Systems Management: Alarm reporting function*.

[ITU-T X.780] ITU-T Recommendation X.780 (2001), *TMN guidelines for defining CORBA managed objects*.

[ITU-T X.780.1] ITU-T Recommendation X.780.1 (2001), *TMN guidelines for defining coarse-grained CORBA managed object interfaces*.

[ITU-T Y.1311] ITU-T Recommendation Y.1311 (2002), *Network-based VPNs – Generic architecture and service requirements*.

[ITU-T Y.1314] ITU-T Recommendation Y.1314 (2005), *Virtual private network functional decomposition*.

[ETSI TS 132 150] ETSI TS 132 150 V6.5.0 (2006), *Digital cellular telecommunications system (Phase 2+); UMTS; Telecommunication management; Integration Reference Point (IRP) Concept and definitions*.

[OASIS] OASIS Standard soa-rm (2006), *Reference Model for Service Oriented Architecture 1.0*, OASIS Standard.

[OMG 98-07-01] OMG Document formal/98-07-01, *The Common Object Request Broker: Architecture and Specification*, Revision 2.2.

[OMG 99-10-07] OMG Document formal/99-10-07, *The Common Object Request Broker: Architecture and Specification*, Revision 2.3.1.

[OMG 00-06-22] OMG Document formal/00-06-22, *Property Service*, Version 1.0.

[OMG 01-02-33] OMG Document formal/01-02-33, *The Common Object Request Broker: Architecture and Specification*, Revision 2.4.2.

[OMG 04-10-02] OMG Document formal/04-10-02, *Event Service Specification*, Version 1.2.

[OMG 04-10-03] OMG Document formal/04-10-03, *Naming Service*, Version 1.3.

[OMG 04-10-13] OMG Document formal/04-10-13, *Notification Service Specification*, Version 1.1.

[TMF versioning] TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA*

	IDL Solution Set, Supporting Document "Programmatic Versioning", file "versioning.pdf".
[TMF objectNaming]	TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA IDL Solution Set, Supporting Document "Multi-Technology Network Management support for a Naming Convention"*, file "objectNaming.pdf".
[TMF servicesUsages]	TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA IDL Solution Set, Supporting Document "Guidelines for Using the OMG Notification and Telecom Log Service"*, file "OMGservicesUsage.pdf".
[TMF LayerRates]	TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA IDL Solution Set, Supporting Document "Layer Rates"*, file "LayerRates.pdf".
[TMF probableCause]	TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA IDL Solution Set, Supporting Document "Specification of probableCause strings"*, file "ProbableCauses.pdf".
[TMF PerformanceParameters]	TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA IDL Solution Set, Supporting Document "Performance Parameters"*, file "PerformanceParameters.pdf".

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses terms defined in other Recommendations. Notes, if any, following the terms describe how the terms are being used in the context of this Recommendation.

3.1.1 agent: See ITU-T Rec. X.701.

3.1.2 event channel: See [ITU-T Q.816] and [OMG 04-10-02].

3.1.3 façade: See [ITU-T X.780.1].

3.1.4 interface: See [ITU-T M.3010].

3.1.5 managed object: See ITU-T Rec. X.700.

NOTE 1 – ITU-T Rec. X.700 restricts the semantics of "resource" to resources which "provide interconnection capabilities" and "allow communications to take place"; it explicitly excludes resources which "provide data storage or processing capabilities". ITU-T Rec. M.3050.1 draws resources from the eTOM's application, computing and network domains.

NOTE 2 – A managed object is characterized in terms of attributes it possesses, operations that may be performed upon it, notifications that it may issue and its relationships with other managed objects.

3.1.6 managed object class: See ITU-T Rec. X.701.

3.1.7 manager: See ITU-T Rec. X.701.

NOTE – Some products that implement the service-oriented framework use this term, or the term "object manager", as a synonym for service-oriented façade.

3.1.8 notification: See ITU-T Rec. X.703.

NOTE – ITU-T Rec. X.703 "Information technology – Open Distributed Management Architecture" provides the ODMA definition, or ODP-RM definition, of notification. Amendment 1 to ITU-T Rec. X.703 "Support using Common Object Request Broker Architecture (CORBA)" relates this definition to the OMG event definition [OMG 04-10-02] (generic or typed event message), and hence to the OMG notification definition [OMG 04-10-13] (generic or typed or structured event message).

3.1.9 notification channel: See [OMG 04-10-13].

3.1.10 operations system: See [ITU-T M.3010].

3.1.11 service-oriented approach/architecture (SOA): See [ITU-T Q.816.2] in harmony with [OASIS].

3.1.12 service-oriented façade: See [ITU-T Q.816.2] in harmony with [OASIS].

3.1.13 service-oriented façade interface: See [ITU-T Q.816.2] in harmony with [OASIS].

3.1.14 service-oriented façade object: See [ITU-T Q.816.2] in harmony with [OASIS].

3.1.15 service-oriented managed object: See [ITU-T Q.816.2] in harmony with [OASIS].

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms:

3GPP	3rd Generation Partnership Project
AI	Adapted Information
AID	Alarm Identifier
AIS	Alarm Indication Signal
ALM	Alarm notification
AP	Access Point
ARC	Alarm Reporting Control
ASA	Alarm Severity Assignment
ASAP	Alarm Severity Assignment Profile
ASN.1	Abstract Syntax Notation One (see X.680-series ITU-T Recommendations)
AVC	Attribute Value Change notification
BSE	Backup Status Event notification
cFD	component Flow Domain
CI	Characteristic Information
CL	ConnectionLess
cLink	component Link
CO	Connection-Oriented
CORBA	Common Object Request Broker Architecture
COS	Current Operational Status
CP	Connection Point
CS	Circuit-Switched

cSN	component SubNetwork
CTP	Connection Termination Point
CTP	Connectivity Termination Point
DIOA	Distributed Interface-Oriented Architecture
DMI	Definition of Management Information
EML	Element Management Layer
EMS	Element Management System
FD	Flow Domain
FDF	Flow Domain Flow
FDFr	Flow Domain Fragment
FDN	Full Distinguished Name
fLink	fibre Link
FM	Fault Management
FP	Flow Point
FPP	Flow Point Pool
FT	File Transfer
FTS	File Transfer Status notification
GDMO	Guidelines for the Definition of Managed Objects (ITU-T Recs X.721 and X.722)
HBE	HeartBeat Event notification
HCPN	Hybrid Circuit/Packet Network
ID	Identifier
IDL	Interface Definition Language
IOR	Interoperable Object Reference
IRP	Integration Reference Point
LC	Link Connection
LF	Link Flow
MFD	Matrix Flow Domain
MO	Managed Object
MOC	Managed Object Class
MTN	Multi-Technology Network
MTNM	Multi-Technology Network Management
MTTM	Multi-Technology Telecommunications Management
NC	Network Connection
NE	Network Element
NF	Network Flow
NML	Network Management Layer
NMS	Network Management System

NSA	Non-Service Affecting
NV pair	Name/Value pair with name being a string and with a "name=value" semantics
NVL pair	Name/Value pair with value being a list of strings
NVS pair	Name/Value pair with value being a string
OCN	Object Creation Notification
ODN	Object Deletion Notification
OMG	Object Management Group
ORB	Object Request Broker
OS	Operations System
PM	Performance Management
PMP	Performance Monitoring Point
PS	Packet-Switched
PTP	Physical Termination Point
RCAI	Root Cause Alarm Indication
RDN	Relative Distinguished Name
RP	Reference Point
Rx	Receive side of unidirectional transmission (sink)
SA	Service Affecting
SCN	State Change Notification
SN	Subnetwork
SNC	Subnetwork Connection
SNFr	Subnetwork Fragment
SO	Service-Oriented
SOA	Service-Oriented Approach/Architecture
SOF	Service-Oriented Façade
TCA	Threshold Crossing Alert notification
TCAPP	Threshold Crossing Alert Parameter Profile
TCP	Termination Connection Point
TFP	Termination Flow Point
TMN	Telecommunications Management Network
TP	Termination Point
TS	Technical Specification
TSN	Technology-Specific Network
TTP	Trail Termination Point
TWM	Threshold Watermark
Tx	Transmit side of unidirectional transmission (source)
UFATN	Unified Functional Approach/Architecture for Transport Networks

UML Unified Modelling Language

UMTS Universal Mobile Telecommunication System

UTC Coordinated Universal Time

VC Virtual Connection/Circuit

VPN Virtual Private Network

5 Conventions

A few conventions are followed in this Recommendation to make the reader aware of the purpose of the text. While most of the Recommendation is normative, paragraphs succinctly 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 preceded by an "O" instead of an "R". For example:

(O) EXAMPLE-2 An example optional requirement.

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

Examples of CORBA IDL are included in this Recommendation and normative IDL specifying the data types, base classes and other service-oriented modelling constructs of the framework (e.g., iterators), is included in Annex A. The IDL is written in a 9-point courier typeface:

```
// Example IDL
   void setAdditionalInfo(
         in globaldefs::NamingAttributes_T objectName,
         inout globaldefs::NVSList_T additionalInfo)
        raises (globaldefs::ProcessingFailureException);
```

Annex A of this Recommendation contains source code that implementers will want to extract and compile. It is normative and should be used by developers implementing systems that conform to this Recommendation. Refer to clause 5.2 of [ITU-T X.780.1] for recommendations with regard to cutting and pasting the IDL of this Recommendation into plain text files that may then be compiled. For example, the entire IDL file "itut_x780_2.idl" can be generated by cutting Annex A from the Microsoft® Word® version of this Recommendation and saving as "Text Only with Line Breaks".

6 Service-oriented interface design considerations

This clause, a companion to clause 6 of [ITU-T Q.816.2], identifies several design considerations that should be addressed by the framework as support for lightweight use of CORBA through service-oriented interfaces is added. While clause 6 of [ITU-T Q.816.2] identifies design considerations for lightweight use of CORBA services, this clause identifies design considerations for lightweight use of IDL repertoire and derived lightweight IDL modelling of service-oriented managed objects and façade objects. Clauses 6.4 to 6.10 are for further study but provide, in the current version of this Recommendation, a brief summary of their intended contents.

CORBA realizes a distributed interface-oriented architecture (DIOA) where location-transparent objects expose one or more interfaces for access to their encapsulated state and behaviour. The concepts of coarse-grained object modelling and service-oriented architecture (SOA) resolve issues with object-oriented analysis and design (OOAD) best practices. Classical OOAD focuses on the class level, i.e., encapsulates behaviour and related data in the same object. This has become known

as a "fine-grained" approach. A general "coarse-grained" approach uses the façade design pattern[1] to separate state and behaviour (and share the state) of some classes, i.e., the behaviour of some (shared) objects is no longer controlled by themselves but by façade objects. In a coarse-grained object-oriented approach data is still encapsulated but data access and control is organized at a separate upper level which results in coarser "grains" being accessed and controlled.

[ITU-T Q.816.1] and [ITU-T X.780.1] use a specific form of the façade design pattern (e.g., an X.780.1 façade can only provide access to a single MOC) and other design considerations to define support for coarse-grained interfaces. This Recommendation, along with [ITU-T Q.816.2], defines a lightweight generic use of the façade design pattern which, together with other lightweight design considerations, generalizes the coarse-grained approach (and therefore indirectly also the fine-grained approach). This novel approach to interface design is called "service-oriented" since it paves the way for introducing SOA principles to the TMN interface specification methodology (M.3020-series of ITU-T Recommendations). CORBA-based service orientation requires the flexibility of application-specific access granularity where well-defined sets of TMN entity types are accessed through assigned façades with IORs (see clause 4.2.1 of [ITU-T X.780]). Refer to clause 12 below and clause 7.3 of [ITU-T Q.816.2] for a concise description of the relationships between the three approaches. Additional information on CORBA IDL modelling is available from [b-Siemens AG].

The service-oriented design considerations related to CORBA IDL repertoire and modelling concern superclasses, naming of managed objects and service-oriented façades, lightweight definition of exceptions, operations and notifications, modelling of multiple telecom transmission technologies according to [ITU-T G.805] and [ITU-T G.809], and standardized modelling of extensions.

6.1 Flexible use of façade design pattern

The coarse-grained framework introduces façades but also requires certain coupling conditions between a façade and the managed objects that are accessible through the façade. A coarse-grained interface is created by first defining a fine-grained interface according to [ITU-T X.780]. Once a fine-grained interface is defined, façade interfaces for each of the managed object interfaces are developed according to [ITU-T X.780.1]. Refer to clause 6.1 of [ITU-T Q.816.2] for further details.

A service-oriented interface can depend on predefined coarse-grained or fine-grained interfaces but usually does not. The service-oriented approach requires the introduction of service-oriented façades, which are CORBA objects, and managed objects, which are accessed and controlled (preferably exclusively) through service-oriented façades[2]. Both types of objects are instantiated at the managed system. Only loose coupling between managed objects and service-oriented façades is required from the outset (see clauses 7.2.1.1 and 10.3) leaving detailed relationships to the

---

[1] The façade design pattern [b-GAMMA] is a structural pattern with the intent to provide a unified interface to a set of interfaces in a subsystem. A façade defines a higher-level interface that makes the subsystem easier to use. A façade promotes loose coupling between the subsystem and its clients. Loose coupling lets one vary the components and interfaces of the subsystem without affecting its clients.

[2] Some products that implement the service-oriented framework use the term "object manager" or just "manager", as a synonym for service-oriented façade. Another synonym sometimes used is "managing object" in contrast with "managed object". Evolving managed objects to "service-oriented managed objects" means separating their state and behaviour, and outsourcing the behaviour to assigned "managing objects" that take a steward role regarding the operations and notifications of their allocated managed objects. This fundamental SOA principle can also be described as separating the definition and storage of data from the delivery of data. Since the terms "managing object" and "manager" may give rise to misunderstandings with regard to the managing role of systems and subsystems, this Recommendation mainly uses the term "service-oriented façade" but may sometimes use the synonyms for explanation purposes.

discretion of the managed system, which may also support dynamic binding of managing systems to one or more service-oriented façades (see clause 7.2.1.2 of [ITU-T Q.816.2]).

While transition from a fine-grained model to a coarse-grained model is defined "bottom-up" through deterministic rules for coarsening of a fine-grained interface, transition from a service-oriented model to a coarse-grained or fine-grained model would be defined (if required someday) "top-down" through rules for stepwise refining of a service-oriented interface according to [ITU-T X.780.1] and [ITU-T X.780]. The service-oriented approach mainly promotes co-existence of fine-grained, coarse-grained and (information model-dependent) service-oriented interfaces, however, and initially does not aim at transition of one TMN interface kind to another.

6.2 Lightweight use of ORB

The OMG CORBA 2 specifications [OMG 98-07-01] [OMG 99-10-07] [OMG 01-02-33] consist of several parts for several aspects of CORBA that shall or should be offered by compliant ORB vendors. Most important for use by the TMN frameworks are the following two aspects of CORBA:

– IDL modelling repertoire of ORB products: These are mainly chapters 3 "OMG IDL Syntax and Semantics" and 5 "Value Type Semantics" of the CORBA 2 specifications.

– Built-in basic CORBA system services of ORB products: These are mainly chapters 4 "ORB Interface" and 11 "The Portable Object Adapter" of the CORBA 2 specifications.

The lightweight use of the first aspect, the IDL repertoire and modelling, is specified in this Recommendation, whilst the lightweight use of the second aspect, the basic CORBA system services, is specified in the companion Recommendation [ITU-T Q.816.2] together with OMG services usage.

6.2.1 IDL repertoire and modelling

The CORBA IDL-related constituents of the service-oriented framework are described in this Recommendation, including the minimum IDL repertoire that should be used, based on the service-oriented object model IDL provided in Annex A (see clauses 8 and 9) and the service-oriented CORBA modelling guidelines for using and extending the object model IDL (see clause 10). As a result, a number of requirements on service-oriented IDL repertoire usage and derived modelling of interfaces, operations, exceptions, attributes, data types and notifications are defined.

A particularly important part is the modelling of managed object classes and service-oriented façade interfaces, which inherit from a set of common attributes `Common_T` (a struct that is often virtual) or from a root service-oriented façade interface called `Common_I` (which is often virtual). The `Common_I` façade interface specifies among other things the setter/mutator functions for the settable common attributes and some getter/accessor functions (see clauses 9.2 and 9.3.2).

6.2.2 Attributes, use of value types, and managed object class specification

A CORBA interface can have constants and attributes which are inherited. Inherited constants can be redefined and attributes are mapped to programming language-specific system accessor/getter and mutator/setter functions with system exceptions; an accessor to retrieve (i.e., get) the value of the attribute and a mutator to modify (i.e., set) the value of the attribute if it is not read-only. Better control of attribute access can be achieved by modelling attributes as explicit IDL operations that are used to read and write the attribute value and have well-defined exceptions (i.e., IDL-specific user accessor and mutator functions). The names of the operations should then indicate the attribute name as well as the type of operation (get or set) and the attribute type should be used as return value and input parameter type. For example, an operation name convention could be "<attribute name>Get" and "<attribute name>Set". Set-valued (i.e., multi-valued) attributes would be modelled as CORBA sequences and should have additional operations to add and remove values to and from

the value set. The fine-grained and coarse-grained frameworks adopt this approach for attribute modelling. Within the service-oriented framework, such accessor and mutator functions are optional and more lightweight modelling rules are recommended.

The collection of all attributes of an interface defines its state[3]. All three frameworks group the state of natively fine-grained managed object interfaces into separate entities, the managed object classes (MOCs), which can be interfaces or be instantiated (by the managed system or the managing system or both, as specified) to "non-CORBA" managed objects that are identifiable by distinguished names. The fine-grained and coarse-grained frameworks require value types as the grouping mechanism while the service-oriented framework allows for more flexibility in TMN interface design and defines four alternative ways of MOC specification:

- a MOC may be an `interface` that has the attributes as CORBA attributes or explicitly defined accessor (and mutator) functions;
- a MOC may be a `struct` that has the attributes as record members;
- a MOC may be a `valuetype` that has the attributes as public state members;
- a MOC may be a CORBA sequence of name/value pairs with `string` names and `any` values, i.e., a MOC may be of type `sequence<struct {string name; any value;};>`, that has the attributes as name/value pairs with a "name=value" semantics.

The `struct` and `valuetype` options can be considered lightweight. The `sequence` option is generic but very heavyweight since it requires the specification of all `name` fields at the interface and the use of type code techniques to properly treat the contents of the `value` fields (or else very detailed implementation agreements between the parties using the interface).

A managed system claiming conformance to the service-oriented framework may choose one of these alternatives to specify MOCs but, for the purpose of transitioning to a harmonized way of MOC definition, with the goal to have only a single alternative for all MOCs in the long term.

6.3 Naming of managed objects and service-oriented façade objects

To achieve the objective of lightweight and SOA-styled use of CORBA through service-oriented interfaces, the resources in a managed system that need to be managed are defined as service-oriented managed objects (also known as second-level or lightweight objects). These managed objects are accessed and controlled through service-oriented façade objects (i.e., CORBA objects). CORBA objects possess an IOR. The managed objects (usually) are not CORBA objects and do not possess an IOR (see clause 6.2.2) and thus relieve the management systems of the burden of storing and maintaining huge numbers of IORs for managed objects. Service-oriented managed objects are identified by names (see clauses 8.5 and 10.3). The CORBA naming of façade objects and the relationship between managed object names and façade object names depends on whether the managed system implements the session service, which allows for a minimalistic use of the naming service. Refer to clauses 7.2.1.1 and 10.3 for more details.

6.4 Creation and deletion of objects

This clause first summarizes creation and deletion of fine-grained and coarse-grained managed objects and coarse-grained façades including OCNs and ODNs. Service-oriented façades cannot be created or deleted by a managing system but are created by the managed system without OCNs during start up and restart. The available façades can be determined by a managing system at any time using the session service or the naming service. Service-oriented managed objects can be created or deleted by the managed system or the managing system or both, and OCNs or ODNs are

---

[3] In the context of state definition, interface constants are considered as initialized read-only attributes. Interface attributes are something akin to public class members though they do not define storage.

generated or not – the actual choices depend on the individual MOC. Creation and deletion of managed objects will modify the set of containment relationships and hence the naming graph (see clauses 6.5 and 10.3.2). The service-oriented framework does not require definition of managed object factories (per MOC). Instead, for each MOC that can be created and deleted by managing systems, the responsible SO façade has to expose create and delete operations.

The completion of this clause is for further study.

6.5 Inheritance and containment

This clause first summarizes the inheritance and containment principles of fine-grained and coarse-grained interfaces. In case of service-oriented interfaces, more lightweight principles are followed. Inheritance is determined by the root classes `Common_T` and `Common_I` (see clauses 8.5 and 9.3) and containment by the service-oriented naming conventions (see clause 10.3.2). The service-oriented framework initially does not require definition of IDL name binding submodules per clause 6.8 of [ITU-T X.780] (with examples from ITU-T Rec. M.3120) to identify the allowable containment relationships but proposes consideration of simplified such modules when progressing the framework.

The completion of this clause is for further study.

6.6 Exceptions

This clause sets out the principles of exceptions as a lead into clause 8.6: the invocation of an operation may result in raising a previously defined exception as listed in the `raises` expression of the operation declaration, or a standard exception defined for the supported ORB version (and not listed under `raises`). An overview is provided of the exceptions defined and used by the fine-grained, coarse-grained and service-oriented frameworks and their different flavours.

The completion of this clause is for further study.

6.7 Retrieval and modification of objects and attributes

This clause leads into clause 9 by identifying, based on clause 6.2.2, the generalizable natures of retrieval and modification (e.g., dealing with individual attributes and attribute selections, dealing with attributes that are lists of parameters such as additionalInfo and transmissionParams, use of iterators, scoping with regard to contained objects, use of a filtering constraint language, unsolicited modifications through state changes) including AVCs and SCNs. It summarizes the principles of clauses 9.2 and 8.7.5.4 and refers to clause 10.1 for more general guidelines.

The completion of this clause is for further study.

6.8 Notifications

This clause leads into clause 8.7 by introducing the principles of notifications arising from CORBA managed objects. It describes the relationship between façades and notifications, provides a concise and consolidated overview of the events of the fine-grained, coarse-grained and service-oriented frameworks including ways for harmonisation/co-existence based on clause 8.3 of [ITU-T Q.816.2], and summarizes service-oriented fault management (with ASAPs and event acknowledgement) and relevant parts of service-oriented performance management (including PMPs and TCAPPs).

The completion of this clause is for further study.

6.9 Conditional capabilities and supported capabilities

The fine-grained and coarse-grained frameworks support, in the context of GDMO translation, a generic concept of conditional support for groups of capabilities called conditional packages and a corresponding packagesGet() operation. The service-oriented framework offers a more lightweight

concept of a two-level capability model with a corresponding getCapabilities() operation to identify the supported operations of a given service-oriented façade (and other capabilities), and with a session service operation for on-demand determination of the supported façades themselves.

The completion of this clause is for further study.

6.10 Additional service-oriented interface design considerations

This clause serves for alignment with the fine-grained and coarse-grained design considerations (see clauses 4 of [ITU-T X.780], 6 of [ITU-T X.780.1], 4 of [ITU-T Q.816] and 6 of [ITU-T Q.816.1]) and oversees further key service-oriented interface design considerations contained in [ITU-T X.780.2] (and [ITU-T Q.816.2]) such as universal parameter list attributes with single-valued (NVS pair), multi-valued (NVL pair) or tagged parameters.

The completion of this clause is for further study.

7 Service-oriented framework and requirements overview

Clause 6 outlined the design considerations that should be resolved as support for service-oriented interfaces is added to the framework. This clause and the rest of this Recommendation provide details on how the framework will be extended to address these issues. This Recommendation defines guidelines for developing or adopting/endorsing information models for service-oriented interfaces including lightweight generic façades to access and control managed objects, while [ITU-T Q.816.2] focuses on the framework support services for service-oriented interfaces. First, an overview of the service-oriented framework is presented, then an overview of its constituents.

7.1 Framework overview

The service-oriented framework consists of two superclasses, three lightweight OMG services, one lightweight ITU-T support service and a small number of conventions (standard data types, notification specifications, etc.). Refer to clause 7.1 of the companion Recommendation [ITU-T Q.816.2] for details. The superclasses and conventions are defined in this Recommendation. Since the service-oriented CORBA framework is capable of supporting any protocol-neutral information model, the potential range of application-specific objects (managed objects and façade objects) that are supported by the framework through inheritance from the superclasses is very large.

7.2 Framework constituents overview

The CORBA services-related constituents of the framework (ORB, OMG services, ITU-T services) are described in the companion Recommendation [ITU-T Q.816.2] and its CORBA IDL-related constituents are described in this Recommendation (superclasses, rules and conventions, naming of managed objects and service-oriented façades, service-oriented CORBA modelling guidelines, IDL repertoire and style guide for use by IDL modellers, modelling of multiple transmission technologies according to [ITU-T G.805] and [ITU-T G.809], standardized modelling of extensions, etc.).

7.2.1 Façade design pattern and service-oriented façade objects

The most significant change to the fine-grained framework required to support coarse-grained interfaces is the way managed objects are accessed and controlled, namely through the use of the façade design pattern (see clause 6.1). Using the façade design pattern, a managed system will support a small number of façade interfaces, at least one but usually no more than a few for each type of managed object on the system. A managing system will then (logically) invoke an operation on a managed object by actually invoking the operation on a façade for that type of managed object on that system. In the façade design pattern, the managed objects do not have to expose a CORBA interface and hence may not have individual IORs. This means a managed system that supports the

façade approach does not need to implement the fine-grained managed object interfaces. These principles of the coarse-grained framework are retained for the service-oriented framework in a more flexible way with additional options.

It is best to think of a service-oriented façade, abbreviated as "SO façade", not as a managed object, though it is managed by a managing system, but as an intermediary object that enables a managing system to manage proper managed objects representing manageable resources. A service-oriented façade is therefore also called a "managing object" or a "manager". The façade object has a CORBA interface and is accessible using CORBA. The proper managed objects, however, usually do not have CORBA interfaces and so are not directly accessible using CORBA. The façade itself does not represent a manageable network resource; its purpose is to enable or facilitate interaction with the objects that do represent manageable resources. A façade exposes behaviour of the managed objects it "manages", i.e., controls and provides access to, but may also expose its own behaviour (e.g., present its capabilities). All façade objects are instantiated by the managed system during startup, restart or during session instantiation and destroyed during shutdown or ending of the session. There are no service-oriented façade factories. Multiple façades for the same type of managed objects may exist on a coarse-grained interface, but usually not on a service-oriented interface. Any managed object shall always be accessible through only one (and always the same) façade. Figure 1 below summarizes the managing and managed entities of the service-oriented framework, i.e., the managing and managed system, the managed objects and the façades.

Figure 1 shows a managing system accessing a managed system that supports the service-oriented approach. The managed system has two façade interface instances that enable the managing system to access two different sets of managed objects. The managed objects at the top of the figure can only be accessed through the SO façade. The managed objects at the bottom also support direct CORBA interfaces and can be accessed either through the façade or directly. They are not service-oriented. Direct CORBA access of managed objects is optional and only specified for reasons of compatibility with the coarse-grained framework. A managed system that supports the real service-oriented façade approach shall provide SO façade interfaces (i.e., managing object classes) for each of its managed object classes and instantiate only SO managed objects.

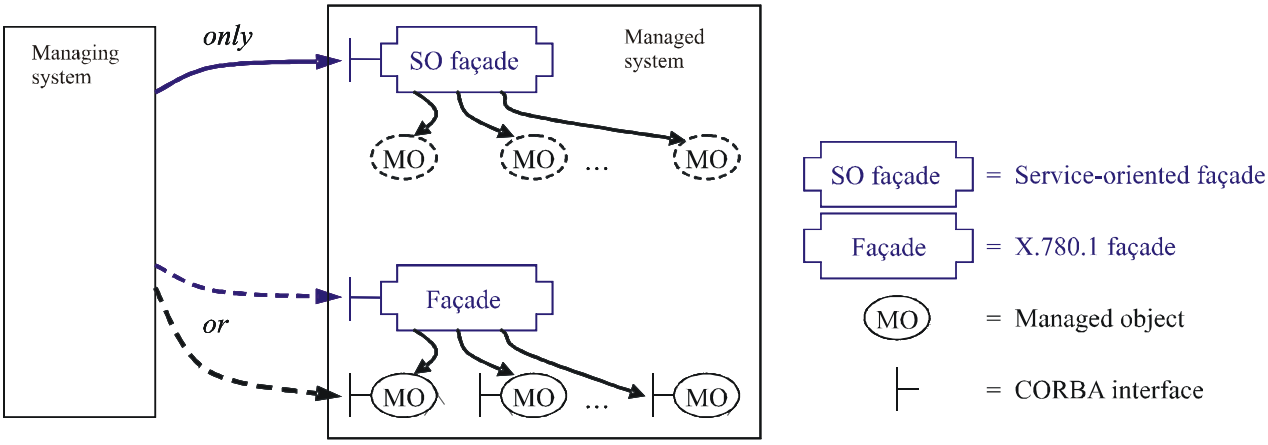

Figure 1 – SO façade and façade of a managed system

A façade may use a managed object's CORBA interface, if available, to invoke an operation on it, or some other implementation-specific means. How SO façades invoke operations on SO managed objects is not exposed at the interface. A managed system, in fact, need not even implement managed objects as individual objects internally. By implementing a CORBA TMN interface based on this framework, however, it will give the illusion that managed objects are internally implemented as objects since they are exposed at the interface as objects of their own.

When an operation is invoked on a managed object through a façade, the façade must then logically invoke the operation on the actual managed object. Because very many managed objects will be accessed through a single façade, the façade must know which managed object is the actual target of the operation. This will be handled by adopting the convention of including the name of the target managed object as the first parameter of every façade operation directed at a managed object. See for example the set operations of the `Common_I` interface in Annex A.

7.2.1.1 Relating managed objects to their service-oriented façade

While managed objects may no longer have unique IORs, they will still have unique names and can still be thought of as individual entities representing manageable resources even though they may not have an individual CORBA IDL interface. It is important that a managing system be able to determine which service-oriented façade to use based only on the managed object's name (see clause 6.2 of [ITU-T Q.816.1]). If it cannot, it will have to query the managed system or persistently associate a façade IOR with every managed object's name. To support the ability to determine a managed object's façade based on only its name, the coarse-grained framework extends the names of managed objects accessible through a façade slightly beyond the names of managed objects not accessible through a façade (see clause 7.2.2 of [ITU-T X.780.1] and clause 8.1.3 of [ITU-T Q.816.1] for details, and clause 10.3.2 for a concise overview).

For reasons of compatibility the service-oriented framework keeps this name extension as an option but also adds a lightweight alternative that allows for simple determination of a managed object's responsible SO façade without the need to change the naming conventions. This option is based on the following rules for service-oriented façade and managed object modelling:

- Each managed object class (MOC) is managed by exactly one service-oriented façade interface (managing object class) but a given service-oriented façade may manage more than one MOC.
- Managed object names and containment relationships are stored, possibly redundantly, by the service-oriented façade that is responsible for the management of the respective managed object (i.e., owns the managed object in a steward role) (see clauses 7.2.2 and 10.2).
- Each service-oriented façade interface is instantiated only once (singleton design pattern[4]).
- From the (full distinguished) name of a managed object, its MOC and its façade steward can be determined unambiguously (deterministic naming – see clause 10.3.2).

As a consequence, the name of any managed object instance will identify uniquely the service-oriented façade instance which is managing that managed object. Refer to clause 10.3 for details on naming conventions for service-oriented CORBA managed objects and façade objects.

7.2.1.2 Relationship to SOA concepts

It is believed that the novel service-oriented approach to CORBA-based TMN interface design can pave the way for introducing SOA principles to ITU-T's management interface specification methodology (M.3020-series of ITU-T Recommendations). Therefore, the most basic SOA principles and artefacts are defined and described in considerable detail, and a mapping between SOA and CORBA concepts is provided. Refer to clause 7.2.1.2 of the companion Recommendation [ITU-T Q.816.2] for details.

---

[4] The singleton design pattern [b-GAMMA] is a creational pattern with the intent to ensure a class only has one instance, and provide a global point of access to it. A singleton class is made itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access its unique instance.

7.2.2 Storage of names and object containment relationships

In the coarse-grained framework, all façade objects have names that are stored in an OMG naming service and the ITU-T containment service is introduced as a central place to store managed object names and containment relationship information for façade-accessible managed objects. The alternative to store managed object names in the façade objects is mentioned but excluded.

In the service-oriented framework, façade objects may be accessed without the OMG naming service if the session service is available, and the façade objects are storage locations of managed object names and their containment relationships if the containment service is not available. These options are a federated and more lightweight means of object name administration that eliminates the threat for performance issues with the containment service similar to those known with the naming service in case of fine-grained interfaces with a gigantic number of objects.

Service-oriented façades provide for the managed objects they own (see clause 7.2.1.1) standard accessor functions to retrieve all names, individual objects by name and, where applicable, contained objects and containing objects. Refer to clauses 9.2, 9.3 and 10.3 for details.

8 Service-oriented object model IDL

Before specifying in clauses 10 and 11 the rules and style idioms for defining service-oriented TMN managed objects and façade objects using CORBA IDL [OMG 98-07-01] [OMG 99-10-07] [OMG 01-02-33], this and the next clause present network management modules containing a set of object interfaces and supporting data structures specified in CORBA IDL that comply with the modelling guidelines and style guide. These IDL modules provide the basic set of IDL definitions on which service-oriented IDL information models are then built. This IDL is included in Annex A.

Clause 7 outlined the constituents of the framework for the service-oriented paradigm covering some of the issues stated in clause 6. This and the next clause continue the description of the extensions of the CORBA framework to support service-oriented interfaces by setting out the considerations covered by the service-oriented object model in IDL provided in Annex A. This clause describes the supporting data types, managed object classes, exceptions and notifications, similar to clause 5 of [ITU-T X.780] for fine-grained interfaces, whilst the next clause describes the façade interfaces, similar to clause 8 of [ITU-T X.780.1] for coarse-grained interfaces.

This clause specifically covers the following key aspects of the service-oriented object model:

– An overview of the very small IDL repertoire required for service-oriented modelling.
– Use of name/value pairs with string values as a principle, and the specific cases:
  - capability support;
  - additional info parameter;
  - transmission parameter;
  - managed object name component (RDN).
– Use of compact list and tabular structures:
  - list of name/value pairs with string values, and the specific cases:
    - list of capabilities with indication of support;
    - list of additional info parameters ("additionalInfo" attribute);
    - list of transmission parameters (untagged "transmissionParams" attribute);
  - name/value pair with list of string values (multi-valued parameter);
  - list of multi-valued parameters;
  - tagged list of name/value pairs with string values (tagged parameters);
  - list of tagged parameters.

– Approach taken to dealing with the representation of parameters for multi-layered objects:
  - layered transmission parameters (i.e., tagged list with transmission Layer rate as tag) (single-layer "transmissionParams" attribute);
  - list of layered transmission parameters (multi-layer "transmissionParams" attribute).
– Approach taken to naming of the objects used in service-oriented interfaces.
– The attributes common to all service-oriented managed object classes:
  - the "name" of the object;
  - a "userLabel" for the object;
  - a "nativeEMSName" for the object;
  - an "owner" attribute;
  - the "additionalInfo" attribute (list of additional info parameters).
– Lightweight means to inherit the common attributes and to add MOC-specific attributes.
– The MOC-specific attributes that characterize single- and multi-layered objects.
– Lightweight concept of operation exception.
– Lightweight approach to notification definition.

This clause provides details on each of the above including IDL fragments taken from the CORBA modules `globaldefs`, `common`, `transmissionParameters`, and `notifications` (see Annex A).

8.1 IDL repertoire and foundation IDL

The foundation IDL of the service-oriented framework (i.e., the SO modelling IDL in Annex A and the SO framework support services IDL in Annex A of [ITU-T Q.816.2]) uses only the following OMG IDL grammatical constructs of CORBA 2 (see chapter 3 of [OMG 98-07-01] [OMG 99-10-07] [OMG 01-02-33]):

• module, interface
• typedef, short, unsigned long, unsigned long long, boolean, any, struct, union, switch, case, enum, sequence, string
• exception, void, oneway, in, out, inout, raises

Figure 2 – Service-oriented IDL repertoire

Using this IDL repertoire, a set of IDL data types, managed object classes, exceptions, notifications, and interfaces have been defined in Annex A and Annex A of [ITU-T Q.816.2] to support service-oriented modelling. Table 1 provides a summary of these IDL specifications.

Table 1 – Service-oriented foundation IDL

IDL	Brief description
Defined in module "globaldefs" in itut_x780_2.idl of Annex A	
NameAndStringValue_T	A structure for representing a name and value pair which uses the string type for the value field (NVS pair)
NVSList_T	A list of name and string value pairs (list of NVS pairs)
NamingAttributes_T	A hierarchical name structure (NVSList_T) representing the full distinguished name of a managed entity that is not instantiated as a first class CORBA object (i.e., does not possess an IOR)
NamingAttributesList_T	A list of naming attribute lists (i.e., a list of lists)
NameAndStringList_T	A structure for representing multi-valued parameters with string values (NVL pair)
NLSList_T	A list of NameAndStringList_T (list of NVL pairs)
TaggedParameters_T	A tagged list of name and string value pairs
TaggedParameterList_T	A list of tagged lists of name and string value pairs
Time_T	A string representing a generalized time (absolute time)
Duration_T	An integer representing a duration of time (relative time)
ExceptionType_T	An enumerated list of possible exception codes
ProcessingFailureException	An exception appended with a string (errorReason) indicating further details about the exception
NamingAttributesIterator_I	A CORBA interface representing an iterator for bulk retrieval of managed object names
IterableManagedEntity_T	A structure representing a generic managed object type which is iterable for bulk retrieval of managed objects
IterableManagedEntityList_T	A list of iterable managed objects
ManagedEntityIterator_I	A CORBA interface representing an iterator for bulk retrieval of managed objects
Defined in module "common" in itut_x780_2.idl of Annex A	
Capability_T	A name and string value pair representing the functionality supported by the EMS across an NMS-EMS interface
CapabilityList_T	A list of capabilities presenting the functionalities supported by the EMS across an EMS-NMS interface
Common_T	A structure defining the mandatory common attributes of each managed object class (including the full distinguished name, user label, native EMS name, owner and additional info parameters) and allowing for extensions by object class-specific attributes
ManagedObjectList_T	A list of generic managed objects (i.e., which have only common attributes)
ManagedObjectIterator_I	A CORBA interface representing an iterator for bulk retrieval of managed objects

Table 1 – Service-oriented foundation IDL

IDL	Brief description
Common_I	A CORBA interface representing a set of services and utilities that is inherited by every service-oriented façade interface, including retrieval of all managed object names, retrieval of all managed objects, retrieval of a specific managed object, setting the native EMS name of a specific managed object, setting the user label of a specific managed object, setting the owner of a specific managed object, getting the capabilities of the manager object (a first class CORBA object) and setting additional information of a specific managed object
Defined in module "transmissionParameters" in itut_x780_2.idl of Annex A	
LayerRate_T	An integer representing the layer of a transport entity
LayerRateList_T	A list of Layer rates
LayeredParameters_T	A structure representing the Layer rate along with the applicable list of transmission parameters
LayeredParameterList_T	A list of Layer rates along with the applicable list of transmission parameters of each of the Layer rates
Defined in module "notifications" in itut_x780_2.idl of Annex A	
NT_OBJECT_CREATION	A constant identifying the OCN notification type
NT_OBJECT_DELETION	A constant identifying the ODN notification type
NT_ATTRIBUTE_VALUE _CHANGE	A constant identifying the AVC notification type
NT_STATE_CHANGE	A constant identifying the SCN notification type
NT_FILE_TRANSFER _STATUS	A constant identifying the FTS notification type
NT_BACKUP_STATUS	A constant identifying the BSE notification type
NT_HEARTBEAT	A constant identifying the HBE notification type
NT_ALARM	A constant identifying the ALM notification type
NT_TCA	A constant identifying the TCA notification type
Event_T	A struct specifying the notification structure for all notification types – this is the OMG structured event
EventList_T	A list of notifications (i.e., OMG structured events)
EventType_T	A struct specifying the event type of a notification – this is the OMG event type consisting of domain_name and type_name
EventName_T	A string holding the OMG event name of the event
NotificationId_T	A string holding the notification ID of the event
ObjectName_T	A name identifying the managed object reporting the event
ObjectType_T	An enum identifying the type of object that is a filterable field attribute of all notifications
ObjectTypeQualifier_T	A string for identifying object types of future releases
EmsTime_T	A Time_T holding the event report time provided by the EMS
NeTime_T	A Time_T holding the event report time provided by the NE
EdgePointRelated_T	A boolean indicating whether an event is related to an edge TP
OCN_T	A struct specifying the OCN notification type

Table 1 – Service-oriented foundation IDL

IDL	Brief description
ODN_T	A struct specifying the ODN notification type
NameAndAnyValue_T	A structure for representing a name and value pair which uses the any type for the value field (NV pair)
NVList_T	A list of name and value pairs in which the values of the value fields are of any type (list of NV pairs)
AVC_T	A struct specifying the AVC notification type
SCN_T	A struct specifying the SCN notification type
FileTransferStatus_T	An enum describing the status of a file transferring process
FTS_T	A struct specifying the FTS notification type
Current_OperationStatus_T	An enum identifying the status of an NE with respect to a database backup operation
BackupStatus_T	A struct specifying the status of a backup operation for an NE
BSE_T	A struct specifying the BSE notification type
HBE_T	A struct specifying the HBE notification type
NativeEMSName_T	A string representing the name of the managed object that reports the event as presented at the EMS GUI
PerceivedSeverity_T	An enum indicating the perceived severity of an alarm
PerceivedSeverityList_T	A list of perceived severities
ServiceAffecting_T	An enum describing the impact of a fault on the monitored entity
AffectedTPList_T	A list of names of TPs affected by an alarm
ProbableCause_T	A string representing the probable cause of an alarm
ProbableCauseList_T	A list of probable causes
ProbableCauseQualifier_T	A string used to fully qualify the probable cause, if needed
NativeProbableCause_T	A string representing the value of a probable cause as portrayed on the EMS user interface
AssignedSeverity_T	An enum indicating an assigned severity of a managed object
AlarmSeverityAssignment_T	A struct specifying the alarm severity assignment (ASA)
AlarmSeverityAssignmentList_T	A list of alarm severity assignment
AdditionalText_T	A string holding additional textual information
AdditionalInfo_T	An NVSList_T that holds additional information
RcaiIndicator_T	A boolean indicating whether an alarm is a raw alarm or a root cause alarm indication (RCAI)
IsClearable_T	A boolean indicating clearability of an event
AcknowledgeIndication_T	An enum indicating the acknowledge state of an event
X733_EventType_T	A string holding the definition of the event type of an alarm according to [ITU-T X.733]
X733_EventTime_T	A Time_T holding the time of generation of an alarm
SpecificProblem_T	A string qualifying a probable cause
SpecificProblemList_T	A list of specific problems
X733_BackupStatus_T	A string specifying whether the managed object reporting an alarm has been backed-up or not

Table 1 – Service-oriented foundation IDL

IDL	Brief description
X733_BackupObject_T	The name of the managed object providing back-up services for the managed object about which an alarm pertains
X733_TrendIndication_T	A string specifying the current perceived severity trend of the managed object reporting an alarm
NotifIDList_T	A list of notification IDs
CorrelatedNotifications_T	A structure identifying the related notifications by pointing to an object and the notifications emitted by the object
CorrelatedNotificationList_T	A list of correlated notification lists
X733_MonitoredAttributes_T	An NVList_T providing attributes of the monitored object
ProposedRepairAction_T	A string describing the proposed repair action
ProposedRepairActionList_T	A list of proposed repair actions
X733_StateChange_T	An NVList_T providing X.731 state transition information
X733_AdditionalInfo_T	An NVList_T providing additional information in alarms
AlarmId_T	A structure uniquely identifying an alarm ID
ALM_T	A struct specifying the ALM notification type
PMParameterName_T	A string holding a PM parameter name
PMLocation_T	A string specifying a PM location
Granularity_T	A string specifying a count period for PM data collection
PMThresholdType_T	An enum describing the threshold types for TCA parameters
TCAId_T	A structure uniquely identifying a threshold crossing alert ID
Value_T	A float holding a PM threshold value
Unit_T	A string holding the unit of measurement of a threshold value
TCA_T	A struct specifying the TCA notification type
AlarmTypeQualifier_T	An enum used to distinguish threshold crossing alert from alarm
AlarmOrTCAIdentifier_T	A switch between an alarm ID or a TCA ID
AlarmAndTCAIDList_T	A list of alarm IDs and TCA IDs (could be a mixed list)
EventIterator_I	A CORBA interface representing an iterator for bulk retrieval of notifications (events, alarms, alerts)
Defined in module "idlVersion" in itut_q816_2.idl of Annex A of [ITU-T Q.816.2]	
Version_I	A CORBA interface that allows the NMS (managing system) to query the current version of the IDL interface (IDL version) implemented by the EMS (managed system)
Defined in module "session" in itut_q816_2.idl of Annex A of [ITU-T Q.816.2]	
Session_I	A CORBA interface that provides capabilities to enable either a client or a server to detect the loss of communication with the associated party and to enable either party to end the association
Defined in module "nmsSession" in itut_q816_2.idl of Annex A of [ITU-T Q.816.2]	
NmsSession_I	A CORBA interface that allows an EMS to notify all connected NMSes about failures to push an event, about the event/alarm loss period being over, about discarding of alarms, TCAs, file transfer statuses, or notifications of other specified types

Table 1 – Service-oriented foundation IDL

IDL	Brief description
Defined in module "emsSession" in itut_q816_2.idl of Annex A of [ITU-T Q.816.2]	
ManagerNames_T	A list of names of service-oriented façades
EmsSession_I	A CORBA interface that allows an NMS to request the manager interfaces that the EMS implements to gain access to a specified manager interface without using the OMG naming service and to gain access to an event channel for receiving notifications
Defined in module "emsSessionFactory" in itut_q816_2.idl of Annex A of [ITU-T Q.816.2]	
EmsSessionFactory_I	A CORBA interface that allows an NMS to obtain the EmsSession_I object from which all managers (i.e., service-oriented façade objects) of the EMS can be obtained

Refer to clause 10 for additional IDL repertoire that is recommended for optional use when extending the foundation IDL for domain-specific applications.

8.2 Use of name/value pairs with string values

As a rule, OMG specifications define properties of entities as name/value pairs with `string` names and `any` values [OMG 98-07-01] [OMG 99-10-07] [OMG 01-02-33] [OMG 04-10-13] [OMG 00-06-22] but also introduce such name/value pairs with `string` values [OMG 04-10-03]. To support and enable extensibility, the service-oriented framework makes vigorous use of free format name/value pairs where the name and the value are both a simple `string` (NVS pairs):

```
module globaldefs
{
...
   struct NameAndStringValue_T
   {
      string name;
      string value;
   };
...
}
```

The rationale for such usage and the "name=value" semantics are set out in clause 10.1. NVS pair structures are available in the IDL where extensibility is recognized as necessary and use of the IDL type `any` is unsuitable for good reasons. There are several cases of use in the service-oriented framework which are detailed below, such as "capability support", "additional info parameter", "transmission parameter", and the "name component" of managed objects.

8.3 Single-valued, multi-valued and tagged parameters

List and tabular structures are used extensively to provide a compact and efficient representation of potentially complex data that is to be conveyed as a single unit. In many cases these structures are built from name/value pairs. IDL describing such cases is provided below (without comments).

```
module globaldefs
{
...
   typedef sequence<NameAndStringValue_T> NVSList_T;
...
   typedef NVSList_T NamingAttributes_T;
...
   typedef sequence<NamingAttributes_T> NamingAttributesList_T;
...
}
```

```
module common
{
...
   typedef globaldefs::NameAndStringValue_T Capability_T;
...
   typedef sequence<Capability_T> CapabilityList_T;
...
   ...
   globaldefs::NVSList_T additionalInfo;
   ...
...
}

module transmissionParameters
{
...
   ...
   globaldefs::NVSList_T transmissionParams;
   ...
...
}

module globaldefs
{
...
   struct NameAndStringList_T
   {
      string name;
      sequence<string> value;
   };
...
   typedef sequence<NameAndStringList_T> NLSList_T;
...
   struct TaggedParameters_T
   {
      unsigned long parameterTag;
      NVSList_T taggedParams;
   };

   typedef sequence<TaggedParameters_T> TaggedParameterList_T;
...
}
```

NVSList_T provides a list of name/value pairs with string values (NVS pairs) that may be used to encode multi-valued attributes as a list of single-valued parameters or components. Examples for such lists are NamingAttributes_T (see clause 8.5), CapabilityList_T (see clause 9.3.2.7), the additionalInfo attribute (see clauses 8.5, 9.2 and 9.3.2.8), and the transmissionParams attribute (see clauses 8.4, 9.2 and 10.4).

String values can be used quite universally to encode list and tabular structures when their use is documented in sufficient detail in a standardized or vendor-specific way. This usage may lead to rather unstructured and complicated contents of the value fields of the NV pairs, however.

More structured and nevertheless lightweight approaches are multi-valued parameters and tagged parameters. NameAndStringList_T generalizes NameAndStringValue_T to name/value pairs where the value is a list of strings (`sequence<string>`) and NLSList_T generalizes NVSList_T to a list of name/value pairs with list of string values (NVL pairs). NLSList_T can be substituted wherever NVSList_T is used in order to provide proper tabular structures: the elements of the sequence of NLSList_T can represent table rows while the elements of the sequence of NameAndStringList_T can represent table columns.

TaggedParameters_T generalizes NVSList_T to a list of name/value pairs which is qualified by a parameter tag, and TaggedParameterList_T is a list of such tagged parameters. The tag type `unsigned long` used here could also be another appropriate IDL type such as `short` or `string`. A

third generalization of NVSList_T would be tagged parameters of type NLSList_T, i.e., the combination of the first and second generalization of NVSList_T (see clause 10.6).

Examples of use of these structures are highlighted throughout clauses 8 and 9.

8.4 Modelling of transmission technologies

Whilst the framework can be applied for the modelling of any OS-OS interface in a service-oriented way, a particularly important application is the case of an NMS-EMS interface where the EMS is in contact with a physical telecom network that may encompass a number of transport, access and aggregation technologies. The framework therefore provides a generic and flexible means to model any of these network technologies down to any required detail and in the context of protocol stacks and multiplexing hierarchies. This approach is called multi-technology capability for short and is based on the layered transport network functional architecture of [ITU-T G.805] and the corresponding management abstractions provided in [ITU-T G.852.2].

The structures LayeredParameters_T and LayeredParameterList_T provide support for layering of the network protocols according to [ITU-T G.805]. A transmission layer per [ITU-T G.805] is determined by a characteristic information and an adapted information which can be identified by a standard tag. This tag is called **Layer rate** and encoded as LayerRate_T. The IDL describing the multi-technology structures as tagged parameters is provided below (without comments).

```
module transmissionParameters
{
...
   typedef short LayerRate_T;
...
   typedef sequence<LayerRate_T> LayerRateList_T;
...
   struct LayeredParameters_T
   {
      LayerRate_T layer;
      globaldefs::NVSList_T transmissionParams;
   };
...
   typedef sequence<LayeredParameters_T> LayeredParameterList_T;
...
}
```

The structures convey both the name or tag of the layer and for each layer a list of attributes. Each attribute is conveyed by the transmissionParams structure as a part of LayeredParameters_T. The transmissionParams structure is a list of name/value pairs that provides a capability to extend definitions of layered managed objects both for proprietary purposes and for standard definitions where inter-release backward compatibility is a concern. In a majority of cases, properties conveyed by LayeredParameters_T can be considered as optional. This structure hence covers all attributes that relate to specific transmission protocol layers as specified by the characteristic information and adapted information concepts of [ITU-T G.805]. The encoding and modelling of such specific layers is out of scope of this generic framework for OS-OS interface design but is left to domain-specific applications and extensions. The structure LayeredParameterList_T encompasses multiple layers and their transmission parameters according to client/server relationships between adjacent layers as specified in [ITU-T G.805]. Refer to clause 10.4 regarding guidelines for the modelling of layered transmission technologies in IDL. In the interest of interoperability, users of the service-oriented CORBA framework for the TMN must cooperate in reaching normative usage of standardized Layer rates (and associated transmission parameters). Details of such cooperation are for further study. The value 0 is not used, and the value 1 is identified as `LR_Not_Applicable` and used in cases where a Layer rate attribute is not applicable. A Layer rate in the range 0 to 9999 is considered as a standardized Layer rate, while the range of numbers 10 000 and above is allocated for vendor-specific usage and a Layer rate in this range is considered as a proprietary Layer rate. The standardized Layer rates specified by MTNM v3.0 [TMF LayerRates] shall be used

as a starting point when an interface specification extends this framework, i.e., the extension shall only define new Layer rates when no value specified by [TMF LayerRates] fits the underlying requirements (see also Table 3).

NOTE – [ITU-T G.805] is concerned with connection-oriented network technologies. The multi-technology capability of the service-oriented framework applies equally well to connectionless network technologies as defined in [ITU-T G.809]. See clause 10.4 for some details.

8.5 Common attributes `Common_T` of service-oriented managed objects

Service-oriented managed object classes and service-oriented façade interfaces shall inherit from a set of common attributes `Common_T`, or from a root service-oriented façade interface called `Common_I` (which is often virtual). The `Common_I` "managing object" class is described in clause 9, and the IDL describing the `Common_T` MOC is provided below (without comments).

```
module common
{
...
   struct Common_T
   {
      globaldefs::NamingAttributes_T name;
      string userLabel;
      string nativeEMSName;
      string owner;
      // add any MOC-specific attributes here
      globaldefs::NVSList_T additionalInfo;
   };
...
}
```

The `Common_T` managed object class has the following attributes:

- **name**: The full distinguished name (FDN) of the managed object (NamingAttributes_T). The name is structured as an ordered list of one or more relative distinguished names (RDNs) or name components. The order of the RDNs reflects the containment relationship of the managed object and each RDN is an NVS pair (NameAndStringValue_T) where the value may itself be substructured (see clause 10.3). The FDN is thus a list of name/value pairs (NVSList_T). This FDN structure is equivalent to the CORBA name structure (see [OMG 04-10-03] and clause 6.4 of [ITU-T Q.816.2]). The FDN is unique within the scope of the EMS and is unchanging through the lifecycle of the managed object, i.e., the name cannot be changed during the life of the object. The name may be chosen by the EMS or NMS depending upon the specific managed object class behaviour. The name is the primary reference used when acquiring or identifying the object.
- **userLabel**: A friendly/familiar name by which the human operator recognizes the object. The userLabel is not guaranteed to be unique but the EMS may support enforcement of uniqueness.
- **nativeEMSName**: A representation of how the managed object is referenced on the EMS displays. The aim is to provide a "nomenclature bridge" to aid the relating of information presented on NMS displays to that on the EMS displays. A mechanism to allow setting of this attribute is available to the NMS. The EMS may reject changes to the value of this attribute. When an object is created by the EMS, the EMS selects the nativeEMSName for the object.
- **owner**: The owner of the managed object as provisioned by the NMS, for example, in the sense of a "used by" relationship. There is no specified behaviour related to this attribute, and it is considered simply as a label.
- **additionalInfo**: A list of name/value pairs that provides a capability to extend the object definition both for proprietary purposes and for standard definitions, in particular where inter-release backward compatibility is a concern. In a majority of cases, property conveyed

by additionalInfo can be considered as optional. Additional information hence covers all attributes that are not specific to a layer instance (see clause 8.4 above) and which are not explicitly modelled at the NML-EML interface.

The `Common_T` structure represents a service-oriented managed object class without explicitly modelled attributes except the common attributes. It is extended to specific managed object classes by simply adding further record members, preferably between owner and additionalInfo, and renaming. This procedure is called lightweight inheritance. It represents the second alternative way of MOC modelling introduced in clause 6.2.2, which is particularly useful for applications that deliberately define very few MOCs. Capabilities of these lightweight managed objects are defined through operations of service-oriented façade instances (see clause 9).

(R) OBJECT-1 Service-oriented resources on a managed system shall be modelled as IDL structs that inherit lightweightly (directly or indirectly) from the `Common_T` struct described above and defined in the CORBA IDL in Annex A. The capabilities described in clause 9 shall be supported.

When `Common_T` is extended to model specific managed object classes, the service-oriented framework recommends to use attributes of type `string` or `NVSList_T` as far as possible and to avoid certain standard IDL types such as `any` and `enum` (refer to clause 10.1 for details). When attributes of type `string` are used, their contents can be standardized in any desired way by appropriate documentation though this weakly typed, lightweight approach does not exploit the type checking capabilities of IDL compilers and ORB runtime environments.

For example, it is recommended to encode an absolute time value (calender date and time of day) as a string with the convention that the contents are encoded according to clause 42 of ITU-T Recommendation X.680 (generalized time type). The IDL describing this useful type is provided below (without comments).

```
module globaldefs
{
...
   typedef string Time_T;
...
}
```

The X.680 string format is "yyyyMMddhhmmss.s[Z|{+|-}HHMm]" (see IDL comments).

It is further recommended to adopt the OMG definition `TimeBase::TimeT` to encode a relative time value (duration). The IDL describing this useful type is provided below (without comments).

```
module globaldefs
{
...
   typedef unsigned long long Duration_T;
...
}
```

NOTE – The OMG time service specification includes the CORBA module `TimeBase` with basic data structures pertaining to time-related CORBA modules (e.g., OMG's `CosNotification`). Relative time values (duration) are encoded as "`typedef unsigned long long TimeT;`" with unit 100 nanoseconds, and absolute time values by the 16-byte struct `UtcT`, which uses `TimeT` with base "15 October 1582 00:00" of the Gregorian calendar. But `UtcT` may also represent relative times depending on context.

The ITU-T module `itut_x780` adopts `TimeBase::UtcT` for the encoding of absolute and relative time values. The above definitions of the service-oriented framework are more lightweight and straightforward, and are independent of OMG's time service specification.

(R) OBJECT-2 When the `Common_T` struct is extended to model specific managed object classes, the types defined in the modules of Annex A shall be used wheresoever possible and reasonable. The admissible contents of strings shall be specified in a much detail as possible.

8.6 Exceptions

The principle of the use of exceptions is set out in clause 10.1. The specific exception types are listed below ExceptionType_T which is an `enum`. The unique exception is augmented using a free format errorReason which is a `string`. The IDL describing the unique exception to be used for all operations of the service-oriented framework is provided below (without comments).

```
module globaldefs
{
...
   enum ExceptionType_T
   {
      // see below for the standard values
   };
...
   exception ProcessingFailureException
   {
      ExceptionType_T exceptionType;
      string errorReason;
   };
...
}
```

Where an operation needs to be rejected, the following values are specified for the exceptionType `enum` and should be used in the response that rejects the operation:

- `EXCPT_NOT_IMPLEMENTED`:

  Used if some IDL operations are optional or not implemented in the specific implementation. If the operation itself is not supported, then errorReason shall be an empty string. If this exception is raised because of the values of specific parameters, then the names of these parameters shall be supplied in errorReason (separated by commas) unless otherwise specified in the operation description.

- `EXCPT_INTERNAL_ERROR`:

  Used to indicate an EMS internal error. Applies to all operations. The errorReason can be used to provide further clarification.

- `EXCPT_INVALID_INPUT`:

  Used if the format of an input parameter is incorrect, for example, if a name which is a 3-level naming attribute is passed as a single level name, then this value may be used in the response that rejects the operation. Also, if a parameter is out of range, this type will be used. The errorReason field will be filled with the parameter that was incorrect.

- `EXCPT_OBJECT_IN_USE`:

  Used to indicate an object already in use where the request to use the object would conflict with the current use.

- `EXCPT_ENTITY_NOT_FOUND`:

  Used if the NMS supplies an object name as a parameter to an operation and the EMS cannot find the object with the given name, then an exception of this type is returned. The errorReason field in the exception will be filled with the name that was passed in as parameter.

- `EXCPT_NOT_IN_VALID_STATE`:

  Used if the client tries to delete an object that is in a state that prevents its deletion.

- `EXCPT_UNABLE_TO_COMPLY`:

  Used as a generic value when a server (EMS) cannot respond to the request and no other exception provides the appropriate clarification. The errorReason string is used to provide further information.

- `EXCPT_NE_COMM_LOSS`:

  Used as a generic value when a server (EMS) cannot communicate with the NE (network element) to which the operation should apply and that prevents the successful completion of the operation. Note that network element here applies to any device to which the server (EMS) will have to communicate to fulfil the operation.

- `EXCPT_CAPACITY_EXCEEDED`:

  Used when an operation will result in resources being created or activated beyond the capacity supported by the NE/EMS.

- `EXCPT_ACCESS_DENIED`:

  Used when an operation results in a security violation.

- `EXCPT_TOO_MANY_OPEN_ITERATORS`:

  Used when an EMS exceeds its internal limit of the number of iterators it can support.

- `EXCPT_USERLABEL_IN_USE`:

  Raised when the userLabel uniqueness constraint cannot be met.

This standard framework exception list may be extended with more focused exceptions that relate to the problem space to which the respective interface using the framework is applied. For example, an exception `EXCPT_UNSUPPORTED_ROUTING_CONSTRAINTS` may be specified where the framework has been used to develop an interface that relates to the problem of routing across a network. Standard OMG exceptions (see clauses 3.15 of [OMG 98-07-01], 3.17 of [OMG 99-10-07] and 4.11 of [OMG 01-02-33]) may also occur, e.g., `NO_IMPLEMENT` if an operation implementation is unavailable, but they should be documented with IDL comments similar to the standard ITU-T exceptions.

Note that modifications of enumerated types such as ExceptionType_T are not backward compatible. Refer to clause 10.5.2.4 for a definition of extensibility for such types and a mechanism how to extend an extensible `enum`. ExceptionType_T becomes extensible when `EXCPT_INTERNAL_ERROR` is considered to be the escape value and errorReason is used to convey new exception values in accordance with the convention "EXCPT_<new exception type>;<errorReason>" similar to the requirement **OBJECT-2** when extending `Common_T`.

8.7 Notifications

The structure and behaviour of notifications of the TMN CORBA framework, no matter whether fine-grained or coarse-grained or service-oriented interfaces are considered, are based on the concepts of the OMG notification service (see clauses 6.5 and 8.3 of [ITU-T Q.816.2]). The fine-grained and coarse-grained frameworks take the 15 event types specified by [ITU-T X.721] and the X.730-series of ITU-T Recommendations as the notification types and define an IDL operation for each type, thereby specifying each type as an OMG typed event (see interface `Notifications` of itut_x780.idl). They require use of structured or typed events, with typed events being the preferred choice. In case of service-oriented interfaces, structured events are the preferred choice and typed events are an option. The service-oriented framework defines nine event types and an IDL struct for each type, thereby fixing all details of the associated structured event in a compilable way (see module `notifications` of itut_x780_2.idl). This clause summarizes the lightweight approach to notification definition.

8.7.1 Notification types and common event fields

The supported standard notification types are defined by constants and, for each type, a three-letter acronym is used as a shortcut. The IDL describing these notification types is provided below.

```
module notifications
{
...
                // Object Creation Notification (OCN)
   const string NT_OBJECT_CREATION = "NT_OBJECT_CREATION";
                // Object Deletion Notification (ODN)
   const string NT_OBJECT_DELETION = "NT_OBJECT_DELETION";
                // Attribute Value Change notification (AVC)
   const string NT_ATTRIBUTE_VALUE_CHANGE = "NT_ATTRIBUTE_VALUE_CHANGE";
                // State Change Notification (SCN)
   const string NT_STATE_CHANGE = "NT_STATE_CHANGE";
                // File Transfer Status notification (FTS)
   const string NT_FILE_TRANSFER_STATUS = "NT_FILE_TRANSFER_STATUS";
                // Backup Status Event notification (BSE)
   const string NT_BACKUP_STATUS = "NT_BACKUP_STATUS";
                // Heartbeat Event notification (HBE)
   const string NT_HEARTBEAT = "NT_HEARTBEAT";
                // Alarm notification (ALM)
   const string NT_ALARM = "NT_ALARM";
                // Threshold Crossing Alert notification (TCA)
   const string NT_TCA = "NT_TCA";
...
}
```

An interface specification that extends this framework may define additional event types according to the rule `const string NT_<notification type> = "NT_<notification type>";`. The encoding of event types as defined by these constants shall be used by the managed system in the common field type_name of all notifications (see below). To guarantee forward compatibility, the managing system should ignore unrecognized notification types.

The attributes of each standard notification type are defined through an appropriate IDL struct in order to fix all standard details in a compilable way (OCN_T, ODN_T, AVC_T, SCN_T, FTS_T, BSE_T, HBE_T, ALM_T, TCA_T). All definitions are specializations of OMG structured events as defined by the OMG notification service specification [OMG 04-10-13], and lists of events (of usually mixed types) are therefore OMG event batches (see clause 8.7.5). These event definitions shall be used for all transport mechanisms such as push/pull model and structured/typed events (see clauses 6.5, 8.3 and 10.2.1 of [ITU-T Q.816.2]). The corresponding IDL is provided below.

```
module notifications
{
...
   typedef CosNotification::StructuredEvent Event_T;
...
   typedef sequence<CosNotification::StructuredEvent> EventList_T;
...
}
```

An interface specification that extends this framework may refine the standard notification types with additional record members. In this context it is also allowed to refactor definitions from the `notifications` module to other CORBA modules (e.g., when adding further PM details to TCA notifications). When name/value pairs are used to define the fields of the filterable event body of structured events, the field names shall be the names of the record members of the struct defining the notification type, and the actual field types shall be the types of these record members, except possibly when such a type is itself a struct, in which case the record members of the nested struct shall be considered directly (e.g., in case of `AlarmId_T` for `ALM_T`).

The common fields of most notification types are eventType, consisting of domain_name and type_name, eventName, notificationId, objectName, objectType, objectTypeQualifier, emsTime, neTime and edgePointRelated. Their IDL data types are either `string` or `boolean` or defined in the

module globaldefs (see above) or newly defined. The IDL describing the new data types for common event fields is provided below (without the tagged comments).

```
module notifications
{
...
   typedef CosNotification::_EventType EventType_T;
...
   enum ObjectType_T
   {
      OT_EMS,
      // OT_<object type>, for each object type supported by a standardized
      //                   interface specification that extends this framework
      OT_AID
   };
...
   typedef string ObjectTypeQualifier_T;
...
}
```

The eventType field of type EventType_T is part of the fixed event header. It consists of the domain_name and the type_name (see clause 8.3 of [ITU-T Q.816.2]), which are both of type string. The values of domain_name are defined by the interface specification that extends this framework. The values of type_name are the constants NT_<notification type> (standard, as defined above, or implementation-specific) that specify the notification types used at the interface.

To comply with the OMG definition of the fixed event header of structured events, each notification has a record member eventName of type string, which is meant to name a specific instance of an OMG structured event where the semantics of the names is defined by the end users of the notification service (i.e., the managed and managing systems). For reasons of interoperability, it is recommended to ignore this field or to have it always set to "".

Each notification has a record member notificationId of type string, which maps to a field of the filterable event body of OMG structured events and contains an implementation-defined free format notification identifier whose semantics is specified in clause 8.1.2.8 of [ITU-T X.733]. The uniqueness and the sequence of the notification ID are not guaranteed. When correlation of notifications according to [ITU-T X.733] is supported (see clause 8.7.3.3), notification IDs must be chosen (by the managed system) to be unique across all notifications from a particular managed object throughout the time the considered correlations are significant. Therefore notification IDs may be reused only when no requirements exist for correlating prior notifications and should be chosen to ensure uniqueness over as long a time as is feasible for the managed system.

The record member objectName of type globaldefs::NamingAttributes_T of an event uniquely identifies the managed object that reports the event (i.e., the event source).

The record member objectType of type ObjectType_T belongs to all notification types that are reported by a managed object. Standardized interface specifications that extend this framework shall define an objectType value for each managed object class that is modelled at the interface (by extending the Common_T structure – see clause 8.5). All standard notification types, except FTS and BSE, have this attribute (see below). The enum avoids any uncertainty in the type of object and allows simple filtering. It cannot be extended in a backward compatible way, however.

A notification must be reported against the correct object type, if this MOC is modelled at the interface. The standard objectType OT_EMS represents the managed system (e.g., EMS). For alarms, the objectType value OT_AID denotes an alarm identifier (AID) and is used to represent the managed system's object types that are not modelled (at the interface) but can emit alarms. Other notification types should not be reported against AIDs. But this objectType value shall also be used for potential new object types in future releases of an interface implementation in order to guarantee backward compatibility of the interface. This convention works as follows.

The objectTypeQualifier record member of type `ObjectTypeQualifier_T` is used to identify managed object types of future releases of standardized interface specifications that extend this framework. It is needed because the `ObjectType_T` enum, once specified for a particular release of an interface specification, cannot be extended for backward compatibility reasons. So when backward compatibility is required (as usual – see clause 10.5), the enum `ObjectType_T` cannot be extended to include future new object types. Therefore, `OT_AID` is used as an "escape value" for the field objectType (see also clause 10.5.2.4), i.e., `OT_AID` may also represent new object types. To identify which of the new object types applies, the filterable field attribute objectTypeQualifier is introduced, which is of type `string` and whose values are as follows:

- `""` – indicates an AID;
- `"OT_<object type>"` – Where <object type> identifies a new object type.

If objectTypeQualifier is not present but objectType has the value `OT_AID`, we have a proper AID.

Most notification types have a record member emstime of type `globaldefs::Time_T`, which holds the time at which the event was reported by the managed system, and a record member neTime of type `globaldefs::Time_T`, which holds the time provided by the NE if the NE reports time and is optional or `""` otherwise. Most notification types have a record member edgePointRelated of type `boolean`, which is `TRUE` if the event relates to a TP that is an edge point or to a managed object that is related to an edge TP, and is `FALSE` or not present otherwise.

8.7.2 OCN, ODN, AVC, SCN, FTS, BSE and HBE notifications

An OCN or ODN holds details of the managed object that is being created or deleted. An AVC or SCN, respectively, holds details of the attribute values or state attribute values of a managed object that have changed. An FTS holds details of a file transfer process. A BSE holds details of a change to a network element's backup status. An HBE holds details of an EMS heartbeat notification. The IDL describing these notification types is provided below (without the tagged comments).

```
module notifications
{
...
   struct OCN_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
      any remainderOfBody;
      // this record member is mapped to the remaining event body of the
      // OMG structured event and holds a copy of the object which has been
      // created; its actual type depends on the value of objectType and is
      // then the struct defining the specified objectType (see clause 8.5)
   };
...
   struct ODN_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
   };
...
```

```
   struct AVC_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
      NVList_T attributeList;
   };
...
   struct SCN_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
      NVList_T attributeList;
   };
...
   struct FTS_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      string fileName;
      FileTransferStatus_T transferStatus;
      short percentComplete;
      string failureReason;
   };
...
   struct BSE_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T meName;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      BackupStatus_T backupStatus;
   };
...
   struct HBE_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      // the value of this record member is always OT_EMS
      globaldefs::Time_T emsTime;
   };
...
}
```

AVC, SCN and ALM notifications have attributes whose type is a list of name/value pairs with `any` values. The IDL describing this type is provided below (without comments).

```
module notifications
{
...
   struct NameAndAnyValue_T
   {
      string name;
      any value;
   };
...
   typedef sequence<NameAndAnyValue_T> NVList_T;
...
}
```

`NVList_T` is equivalent to the NVList standard defined by the OMG to represent (lists of) general properties (see clause 8.2). Refer to the IDL comments in Annex A for a description of usage.

FTS notifications convey the current value of the status associated with a file transfer process. The IDL describing the type of the file transfer status is provided below (without comments).

```
module notifications
{
...
   enum FileTransferStatus_T
   {
      FT_IN_PROGRESS,
      FT_FAILED,
      FT_COMPLETED
   };
...
}
```

Refer to the IDL comments in Annex A for a description of these values and the usage together with the percentComplete and failureReason attributes of FTS notifications.

BSE notifications convey the current value of the operational status associated with a database backup process of a network element (which may well be more sophisticated than a file transfer process). The IDL describing the type of this status is provided below (without comments).

```
module notifications
{
...
   enum Current_OperationStatus_T
   {
      COS_Idle,
      COS_Pending,
      COS_InProgress,
      COS_Completed,
      COS_Aborted
   };
...
   struct BackupStatus_T
   {
      Current_OperationStatus_T opStatus;
      string failureReason;
   };
...
}
```

Refer to the IDL comments in Annex A for a description of the values and record members.

HBE notifications are emitted by the EMS and convey an "I'm alive" absolute time value. Refer to clauses 7.2.5 and 9.1.1 of [ITU-T Q.816.2] and clause 7.5 of [ITU-T Q.816] for information on how this interface may be used or extended to offer a heartbeat service (Q.816.2 session service or Q.816 heartbeat service).

8.7.3 Service-oriented fault management

The lightweight approach to alarm definition covers the mandatory attributes of the alarm record definition of [ITU-T X.721] and [ITU-T X.733], namely managed object class, managed object instance, event type, probable cause and perceived severity, and defines further attributes including all conditional X.733/X.721 attributes. Alarms are specific types of notifications emitted from managed objects concerning detected faults or abnormal conditions. Managed object class (MOC) specifications should include causes of potentially abnormal situations which can then be used to convey with each alarm of the MOC such a cause, called the probable cause of the alarm. Probable causes are used to classify alarm types. When available, additional diagnostic information should be notified (e.g., information related to side effects such as current and past values of state attributes, further qualification of the probable cause). Refer to clause 8.7.3.1 for an overview of the probable causes defined in [ITU-T M.3100], [ITU-T X.721], [ITU-T X.733] and this Recommendation.

[ITU-T X.721] and [ITU-T X.733] introduce five event types for alarms which can also be conveyed with the lightweight alarm structure (see clause 8.7.3.3). Additionally, the service-oriented framework calls the X.733/X.721 alarm with probable cause thresholdCrossed (and event type qualityofService) a threshold crossing alert (TCA) and defines its structure independently of the alarm structure. The TCA structure and behaviour depend on the concepts of service-oriented performance management (PM) outlined in clause 8.7.4. A TCA is an event used across the OS-OS interface to indicate that a PM parameter threshold, associated with a managed object, has been crossed. TCAs have specific PM-related attributes not found in (other) alarms and do not need a probable cause and related attributes, and ALM notifications do not provide any threshold information. TCAs address the requirement of early detection of faults before significant effects have been felt by the user. Degradation of service may be detected by monitoring of error rates. Threshold mechanisms on counters and gauges are a method of detecting such trends and providing a warning to managing systems when the rate becomes, or tends to become, inacceptable (see clause 8.7.4).

An important criterion by which failures of managed objects are to be notified is the level to which the fault degrades the quality of the service that was originally requested by (or promised to) the service user. Malfunctions will range in severity from warning, where there is no impact upon the quality of service offered to the user, to critical, where it is no longer possible to provide the service requested by (or promised to) the service user. The level of severity can be described generically and criteria specified based upon the level of degradation that the fault causes to the service: critical, major, minor, warning, or indeterminate. When conveyed in an alarm, this information is called perceived severity. When used in an MOC specification (on a per probable cause basis), this information is called assigned severity. Refer to clause 8.7.3.2 for details.

A further perceived severity level is cleared, which indicates the clearing of one or more previously reported, clearable alarms or TCAs. A raise/trigger event is called clearable if a clear event can be generated for it after its cause has disappeared. Therefore, the perceived severity attribute allows to distinguish raise/trigger events from clear events. ALM and TCA notifications have a mandatory attribute isClearable of type `boolean` which is an indication as to whether a raise/trigger event will have an associated clear event (or the event is a clear itself). An alarm or TCA is called active if it is not clearable or is clearable but no associated clear notification has been received yet.

NOTE – This framework specifies reportable alarms (i.e., alarm reports). There may well be pending alarms associated with managed objects whose existence can be recognized through an alarm status attribute (see clauses 8.7.3.2 and 8.7.5.3). In the presence of alarm reporting control (ARC) (see [ITU-T M.3100]), i.e., when alarm reporting is inhibited, all alarms are pending. Pending alarms are also considered active.

Alarms have an attribute serviceAffecting of type `ServiceAffecting_T` which specifies the impact of a fault on monitored managed entities on further providing their service, as determined by the managed system. The IDL describing this data type is provided below (without comments).

```
module notifications
{
...
   enum ServiceAffecting_T
   {
      SA_UNKNOWN,
      SA_SERVICE_AFFECTING,
      SA_NON_SERVICE_AFFECTING
   };
...
}
```

Refer to the IDL comments in Annex A for a description of these values.

8.7.3.1 Probable causes

Alarms have a mandatory record member probableCause of type `string` which defines further qualification as to the probable cause of the alarm in the sense of [ITU-T X.733] as stated above. [ITU-T X.733] and [ITU-T X.721] define 57 probable causes, [ITU-T M.3100] defines or reserves 207 probable causes that partly overlap with the X.733/X.721 definitions, and TM Forum's MTNM v3.0 defines 96 probable cause strings [TMF probableCause]. The service-oriented framework does not encode specific probable causes but any interface specification that extends this framework shall specify all probable causes which may occur for any managed object according to the following requirement (e.g., `"AIS"` for alarm indication signal, `"BER_SD"` for bit error rate signal degrade).

(R) NOTIF-1 Probable causes shall be specified as human-readable strings which contain only capital letters, underscores, hyphens (not dashes) and digits. The standard probable cause strings specified by MTNM v3.0 [TMF probableCause] shall be used as a starting point when an interface specification extends this framework, i.e., the extension shall only define new probable causes when no value specified by [TMF probableCause] fits the underlying requirements. When probable causes from [ITU-T X.733], [ITU-T X.721] or [ITU-T M.3100] are being used, they shall be translated into syntactically correct strings in the obvious fashion (e.g., leak detected/leakDetected into `"LEAK_DETECTED"`).

Since probable causes are defined as strings, sets of probable causes are defined as lists of strings:

```
module notifications
{
...
   typedef sequence<string> ProbableCauseList_T;
...
}
```

This type may be used, for example, to include or exclude alarms with specified probable causes in or from result lists of on-demand pull event operations (see clause 8.7.5.4).

8.7.3.2 Perceived and assigned severities

The perceived severity levels provide an indication of how it is perceived that the capability of a managed object reporting an alarm or a TCA has been affected. Four levels represent service affecting (SA) or non-service affecting (NSA) conditions according to the determination by the managed system (see also the serviceAffecting attribute): as stated by [ITU-T X.733], usually critical and major are SA and require immediate or urgent corrective action, while usually minor and warning are NSA or potentially SA and require corrective or diagnostic action.

ALM and TCA notifications have a mandatory attribute perceivedSeverity of type `PerceivedSeverity_T` which holds the X.733-defined perceived severity level. In case of a TCA, this attribute mainly serves as the trigger flag, i.e., allows to distinguish between a trigger/raise TCA (value `PS_INDETERMINATE` or, optionally, some SA or NSA level) and a clear TCA (value `PS_CLEARED`). The IDL describing this data type is provided below (without comments).

```
module notifications
{
...
   enum PerceivedSeverity_T
   {
      PS_INDETERMINATE,
      PS_CRITICAL,
      PS_MAJOR,
      PS_MINOR,
      PS_WARNING,
      PS_CLEARED
   };
...
}
```

Refer to clause 8.1.2.3 of [ITU-T X.733] and Corrigendum 2 to [ITU-T X.733] for the semantics of the values.

Sets of perceived severities are defined as lists of `PerceivedSeverity_T`:

```
module notifications
{
...
   typedef sequence<PerceivedSeverity_T> PerceivedSeverityList_T;
...
}
```

This type may be used, for example, to include or exclude alarms and TCAs with specified perceived severities in or from result lists of on-demand pull event operations (see clause 8.7.5.4).

NOTE 1 – The perceived severity levels are the starting point for [b-ITU-T X.731], [ITU-T X.721] and [ITU-T M.3100] to define the alarm status of managed objects. This status is of interest when considering lifecycles of alarms and TCAs in the context of event acknowledgement. Refer to clause 8.7.5.3 for some details.

The relationship between probable causes and perceived severities (and service affection), as reported by alarms, is fundamental for (service-oriented) alarm management. Therefore, the advance assignment of severities to probable causes per managed object (having regard to service affection) is of interest. For this purpose, the concepts of assigned severity and alarm severity assignment (ASA) are introduced for SA, NSA and service-independent usage (as determined by the managed system). An ASA list provides a listing of all abnormal conditions that may exist in instances of a managed object class. The IDL describing these types is provided below (without comments).

```
module notifications
{
...
   enum AssignedSeverity_T
   {
      AS_INDETERMINATE,
      AS_CRITICAL,
      AS_MAJOR,
      AS_MINOR,
      AS_WARNING,
      AS_NONALARMED,
      AS_FREE_CHOICE
   };
...
   struct AlarmSeverityAssignment_T
   {
      string probableCause;
      string probableCauseQualifier;
      string nativeProbableCause;
      AssignedSeverity_T serviceAffecting;
      AssignedSeverity_T serviceNonAffecting;
      AssignedSeverity_T serviceIndependentOrUnknown;
   };
```

```
...
   typedef sequence<AlarmSeverityAssignment_T> AlarmSeverityAssignmentList_T;
...
}
```

Refer to the IDL comments in Annex A for a description of the values and record members. They comply with the corresponding AlarmSeverityCode/Assignment definitions of [ITU-T M.3100].

NOTE 2 – ASA lists may be used to define the managed object class alarm severity assignment profile (ASAP), as specified by [ITU-T M.3100], which models the (flexible) severity assignment to specified probable causes. ASAPs can then be attached to instances of managed objects by using an "ASAP pointer" attribute that holds the ASAP identifier (i.e., the FDN of the ASAP – see clause 8.5).

8.7.3.3 Coding of X.733 alarm information

The alarm definition of the service-oriented framework provides means to allow for conveyance of fully X.733-compliant alarms or of all X.733-specified details that a managed system may offer to managing systems. This clause presents a concise overview of this capability.

The X.733/X.721 alarmRecord is derived from the X.733/X.710/X.721 eventLogRecord, which in turn is derived from the X.735/X.721 logRecord. Table 2 therefore lists all attributes of logRecord, eventLogRecord and alarmRecord, and maps them to corresponding fields of the ALM and TCA notification types. This is the service-oriented coding of X.733 alarm information.

Table 2 – Mapping of X.733 alarms to service-oriented notifications

X.733 attribute	X.780.2 attribute	Comment
Log record Id	N/A	This applies only to log records.
Logging time	emsTime	Though this really applies to log records (identifying the time at which the record was entered into the log) it is used here as the time when the event was reported by the managed system. In the absence of time synchronization, this time may be greater or less than event time.
Managed object class	objectType and objectTypeQualifier	See clause 8.7.1.
Managed object instance	objectName	This is the name attribute of the `Common_T` struct (see clause 8.5).
Event type	x733_EventType	Valid values are: `"communicationsAlarm"`; `"qualityofServiceAlarm"`; `"processingErrorAlarm"`; `"equipmentAlarm"`; and `"environmentalAlarm"` (see IDL comments).
Event time	neTime or x733_EventTime	x733_EventTime shall only be used if it differs from neTime.
Notification Identifier	notificationId	See IDL comments.
Correlated notifications	x733_CorrelatedNotifications	See IDL comments.
Additional text	additionalText	See IDL comments.
Additional information	additionalInfo or x733_AdditionalInfo	x733_AdditionalInfo (`any` values) shall only be used if additionalInfo (`string` values) cannot be used.

Table 2 – Mapping of X.733 alarms to service-oriented notifications

X.733 attribute	X.780.2 attribute	Comment
Probable cause	probableCause	See **NOTIF-1**.
Perceived severity	perceivedSeverity	See clause 8.7.3.2.
Specific problems	probableCauseQualifier or x733_SpecificProblems	x733_SpecificProblems, a list of strings shall only be used if the string probableCauseQualifier cannot be used (see IDL comments).
Backed-up status	x733_BackUpStatus	Valid values are: `"BACKED_UP"`; and `"NOT_BACKED_UP"`.
Back-up object	x733_BackUpObject	See IDL comments.
Trend indication	x733_TrendIndication	Valid values are: `"LESS_SEVERE"`; `"NO_CHANGE"`; and `"MORE_SEVERE"`.
Threshold information		See clause 8.7.3.
triggered threshold	TCAId_T.pmParameterName	See clause 8.7.4.1.
threshold level	TCAId_T.thresholdType and TCA_T.additionalText	additionaltext is used to convey the hysteresis of a gauge-threshold (see [ITU-T X.721] and [ITU-T X.733]).
observed value	TCA_T.value and TCA_T.unit	See IDL comments.
arm time	TCA_T.additionalText	The syntax is `globaldefs::Time_T` while the semantics is defined by clause 8.1.2.7 of [ITU-T X.733].
State change definition	x733_StateChange	See IDL comments.
Monitored attributes	x733_MonitoredAttributes	See IDL comments.
Proposed repair actions	x733_ProposedRepairActions	See IDL comments.

As shown in Table 2, the X.733 attributes logging time, managed object class, managed object instance, notification Id, additional text, probable cause and perceived severity map to previously defined ALM notification fields. The 12 optional field attributes of the alarm definition that remain to be introduced (event type, event time, correlated notifications, additional information, specific problems, backed-up status, back-up object, trend indication, state change definition, monitored attributes, proposed repair actions, threshold information) are described and specified in Annex A. The IDL describing the related data types is partly provided below (without comments).

```
module notifications
{
...
   typedef string SpecificProblem_T;
...
   typedef sequence<SpecificProblem_T> SpecificProblemList_T;
...
   typedef sequence<string> NotifIDList_T;
...
   struct CorrelatedNotifications_T
   {
      globaldefs::NamingAttributes_T source;
      NotifIDList_T notifIDs;
   };
...
```

```
   typedef sequence<CorrelatedNotifications_T> CorrelatedNotificationList_T;
...
   typedef string ProposedRepairAction_T;
...
   typedef sequence<ProposedRepairAction_T> ProposedRepairActionList_T;
...
}
```

Refer to the IDL comments in Annex A for a description of all X.733-related data types and their usage within ALM and TCA notifications. The type declarations omitted above all use `typedef` to rename previously introduced data types with new semantics. For some X.733-related type declarations, their unnamed equivalents are used in the ALM record definition. This is justified for the sake of readability and since `ALM_T` is mapped to structured events anyway.

(R) NOTIF-2 A managed system shall convey alarm attributes that are defined by [ITU-T X.733] and [ITU-T X.721] with ALM notifications or TCA notifications as specified in Table 2.

Since some X.733 alarm attributes cannot be considered lightweight, the degree of X.733 alarm compliance is left to the discretion of the managed system. But when X.733 features are used with the service-oriented framework, that usage shall comply with **NOTIF-2**.

8.7.3.4 ALM identification

Alarms are uniquely identified by the combination of the objectName, layerRate, probableCause, and probableCauseQualifier fields. These fields are therefore grouped into the `AlarmId_T` struct. The IDL describing this type is provided below (without comments).

```
module notifications
{
...
   struct AlarmId_T
   {
      globaldefs::NamingAttributes_T objectName;
      transmissionParameters::LayerRate_T layerRate;
      string probableCause;
      string probableCauseQualifier; // OPTIONAL
   };
...
}
```

Refer to the IDL comments in Annex A for a description of the record members.

8.7.3.5 ALM notifications

The IDL describing the ALM notification type is provided below (without comments).

```
module notifications
{
...
   struct ALM_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      AlarmId_T alarmId;
      string nativeProbableCause; // OPTIONAL
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
      string nativeEMSName; // OPTIONAL
      PerceivedSeverity_T perceivedSeverity;
      ServiceAffecting_T serviceAffecting;
      globaldefs::NamingAttributesList_T affectedTPList; // OPTIONAL
      string additionalText; // OPTIONAL
      globaldefs::NVSList_T additionalInfo; // OPTIONAL
      boolean rcaiIndicator;
```

```
      boolean isClearable;
      AcknowledgeIndication_T acknowledgeIndication; // OPTIONAL
      string x733_EventType; // OPTIONAL
      globaldefs::Time_T x733_EventTime; // OPTIONAL
      SpecificProblemList_T x733_SpecificProblems; // OPTIONAL
      string x733_BackedUpStatus; // OPTIONAL
      globaldefs::NamingAttributesList_T x733_BackUpObject; // OPTIONAL
      string x733_TrendIndication; // OPTIONAL
      CorrelatedNotificationList_T x733_CorrelatedNotifications; // OPTIONAL
      NVList_T x733_MonitoredAttributes; // OPTIONAL
      ProposedRepairActionList_T x733_ProposedRepairActions; // OPTIONAL
      NVList_T x733_StateChange; // OPTIONAL
      NVList_T x733_AdditionalInfo; // OPTIONAL
   };
...
}
```

Refer to the previous clauses and Annex A for a description of the record members.

8.7.4 Service-oriented performance management

The lightweight approach to performance management (PM) defines targets, subjects, thresholds and results of PM activities such as performance monitoring, threshold supervision and PM data collection. The targets are managed objects such as TPs which are called measurement points and where access points to PM activities are identified by certain transmission Layer rates (see clause 8.4), locations (see clause 8.7.4.2), and granularities (see clause 8.7.4.3). PM parameters (see clause 8.7.4.1) are the subjects of the PM activities on measurement points and may have associated one or more PM thresholds (see clause 8.7.4.4). When such a threshold is being crossed (and threshold supervision is activated), the measurement point emits a threshold crossing alert (TCA) notification with the characteristics of the PM parameter and PM threshold. Such a complete set of threshold crossing characteristics is called a TCA parameter (PM parameter name, PM location, granularity, PM threshold) and can be applied, if meaningful, to one or more (or all) transmission Layer rates (see clause 8.7.4.6).

TCA notifications are the results of threshold supervision but the real results of PM activities on measurement points are the PM data which are stored for a certain holding time as current PM data in collection bins according to the required granularity (e.g., 15 min, 24 h, instantaneous) and made persistent as history PM data records at the end of the granularity period. Currently the service-oriented framework only specifies the TCA-related part of PM. The definition of PM data record structure (PM file) and PM operations (i.e., a service-oriented façade for PM) is for further study.

8.7.4.1 PM parameters

TCAs have an attribute pmParameterName of type `PMParameterName_T` which holds the name of a performance measure (i.e., a performance parameter or a PM parameter for short). It has been defined as a `string` to accommodate backward compatibility and proprietary extension. The IDL describing this type is provided below (without comments).

```
module notifications
{
...
   typedef string PMParameterName_T;
...
}
```

TM Forum's MTNM v3.0 defines 169 PM parameters [TMF PerformanceParameters], which are taken from various standard specifications such as ITU-T Recs G.821, G.826/G.828/G.829, RFCs 2495/2496/2665/2819, [b-af-nm-0020.000]. The service-oriented framework does not encode specific PM parameters but any interface specification that extends this framework shall specify all PM parameters that may occur according to the following requirement in the normative form

"PMP_<performance measure>" (e.g., "PMP_AISS" for alarm indication signal seconds, "PMP_BER" for bit error rate).

(R) NOTIF-3 PM parameters shall be specified as human-readable strings which start with "PMP_" and contain only capital letters, underscores, digits and plus signs. The standard PM parameter strings specified by MTNM v3.0 [TMF PerformanceParameters] shall be used as a starting point when an interface specification extends this framework, i.e., the extension shall only define new PM parameters when really no value specified by [TMF PerformanceParameters] fits the underlying requirements. When PM parameters from [ITU-T Q.822] are being used, they shall be translated into syntactically correct strings in the obvious fashion (e.g., controlled slip second far end (CSSFE) into "PMP_CSSFE").

PM parameters are counters or gauges as defined by [ITU-T X.721] with technology-specific (i.e., layer rate-specific) semantics. A counter is a non-negative integer with initial value 0 that is associated with some internal event and incremented by 1 each time the event occurs until it reaches a specified maximum value when it is wrapped around to its initial value. A gauge is a non-negative integer or real that may increase or decrease by arbitrary amounts and has no initial value and no wrap-around behaviour but a maximum and a minimum value.

Counters are monitored by (counter-)thresholds and gauges are monitored by gauge-thresholds or tide-marks. A threshold consists of one or more integers, called comparison levels, and triggers a TCA every time the counter reaches the comparison level. The TCA triggering may be turned on or off and to each comparison level an integer, called offset value, may be associated such that whenever the counter increases by an interval equal to the offset value the threshold triggers the specified notification. The arm-time of a threshold is the time at which the counter was last initialized or the threshold offset was last reached. A gauge-threshold consists of one or more TCA trigger levels, each consisting of a notify-high value and a notify-low value, and optionally triggers a TCA every time the gauge either reaches or exceeds the notify-high value or falls to or below the notify-low value. A hysteresis mechanism can be provided, based on the difference between the notify-low and notify-high values, to avoid the repeated triggering of TCAs when the gauge makes small oscillations around a threshold value. A tide-mark can be used to record the maximum or minimum value reached by a gauge during a measurement period.

8.7.4.2 Location identification

TCA notifications have an attribute pmLocation of type PMLocation_T which holds the location and orientation/direction for the measurement of the associated PM parameter thereby qualifying the traffic to be monitored, supervised and measured. It has been defined as a string to accommodate extensibility. The IDL describing this type is provided below (without comments).

```
module notifications
{
...
   typedef string PMLocation_T;
...
}
```

The format is "PML_<location>" and the valid standard values are "PML_NEAR_END_Rx", "PML_FAR_END_Rx", "PML_NEAR_END_Tx", "PML_FAR_END_Tx", "PML_BIDIRECTIONAL", "PML_CONTRA_NEAR_END_Rx" and "PML_CONTRA_FAR_END_Rx". They apply to measurement points which are TPs. PM parameters may relate to measurements taken on receive (Rx) or transmit (Tx) traffic either at the named TP (PML_NEAR_END_Rx/Tx) or at the TP at the far end of the trail/connection connected to the named TP (PML_FAR_END_Rx/Tx). Alternatively, the PM parameters may be bidirectional (PML_BIDIRECTIONAL) (e.g., resulting from a second-by-second summation and evaluation of both far and near TPs). Measurements may also be carried out contra-directional to the supervised signal (PML_CONTRA_NEAR/FAR_END_Rx).

The semantics of the location, where performance monitoring takes place, and the orientation/direction of the involved signals refer to the basic concepts of transport processing functions for unidirectional transmission defined in [ITU-T G.805] and [ITU-T G.852.2], in particular the adaptation and trail termination sinks (receive/incoming traffic, decoding, Rx) and the adaptation and trail termination sources (transmit/outgoing traffic, encoding, Tx). The involved signals, the monitored/supervised signal and the monitoring/measuring signal, can be either co-directional (i.e., the sink/source monitoring function relates to information mapped from/to the received/transmitted supervised signal) or contra-directional (i.e., the sink/source monitoring function relates to information mapped to/from the transmitted/received supervised signal).

These G.805-based concepts are visible from the syntax of the standard PM location values, i.e., `<location>` ::= {"" | "CONTRA_"}{"NEAR_END_" | "FAR_END_"}{"Rx" | "Tx"}, which shows that the attribute pmLocation of TCA notifications identifies three different items:

– The physical direction of the monitored signal (i.e., which bitstream is supervised) compared to the monitoring signal, identified by ""/"Contra_", where "" refers to co-directional signals and "Contra_" refers to contra-directional signals.
– The logical direction of the monitored signal (i.e., to which direction the information of the supervised bits of the physical bitstream is related), identified by "Near_End"/"Far_End", where "Far_End" refers to the opposite direction.
– The traffic type of the supervised function whose signal is measured by the monitoring function, identified by "Rx"/"Tx", where "Rx" refers to sink/receive/incoming traffic and "Tx" refers to source/transmit/outgoing traffic.

The precise semantics of the standard values are therefore defined as follows:

– `"PML_NEAR_END_Rx"`:

  Identifies a unidirectional measurement provided by a sink monitoring function that supervises the signal which is related to the sink monitored function (Rx traffic).

  The monitored and monitoring signals are co-directional.

  The direction of the supervised signal is measured.

– `"PML_FAR_END_Rx"`:

  Identifies a unidirectional measurement provided by a sink monitoring function that supervises the signal which is related to the sink monitored function (Rx traffic).

  The monitored and monitoring signals are co-directional.

  The opposite direction of the supervised signal is measured.

– `"PML_NEAR_END_Tx"`:

  Identifies a unidirectional measurement provided by a source monitoring function that supervises the signal which is related to the source monitored function (Tx traffic).

  The monitored and monitoring signals are co-directional.

  The direction of the supervised signal is measured.

– `"PML_FAR_END_Tx"`:

  Identifies a unidirectional measurement provided by a source monitoring function that supervises the signal which is related to the source monitored function (Tx traffic).

  The monitored and monitoring signals are co-directional.

  The opposite direction of the supervised signal is measured.

– "PML_BIDIRECTIONAL":

Identifies a bidirectional measurement. Both types of the supervised signal (Rx and Tx) and both directions (Near_End and Far_End) are taken into account for this measure. There is no distinction between co-directional and counter-directional measurement.

For example, the measurement of the PM parameter may result from a second-by-second summation and evaluation at both the near-end TP and its far-end counterpart.

– "PML_CONTRA_NEAR_END_Rx":

Identifies a unidirectional measurement provided by a sink monitoring function that supervises the signal which is related to the source monitored function (Tx traffic).

The monitored and monitoring signals are contra-directional.

The direction of the supervised signal is measured.

– "PML_CONTRA_FAR_END_Rx":

Identifies a unidirectional measurement provided by a sink monitoring function that supervises the signal which is related to the source monitored function (Tx traffic).

The monitored and monitoring signals are contra-directional.

The opposite direction of the supervised signal is measured.

For rapid provisioning purposes, one may collect PM parameters per PM location, layer rate and granularity in a managed object that can be attached to measurement points (see clause 8.7.4.4).

NOTE – Measurement points of the service-oriented framework are TPs and network elements (i.e., all TPs of a specified NE) and so service-oriented PM is restricted to the NE level. The extension of lightweight PM concepts, such as PM location, to network-level managed objects is for further study.

8.7.4.3 Granularity

TCAs have an attribute granularity of type `Granularity_T` which holds the count period or measurement period (interval or duration) for which PM data may be collected (and subsequently retrieved). The IDL describing this data type is provided below (without comments).

```
module notifications
{
...
   typedef string Granularity_T;
...
}
```

Refer to the IDL comments in Annex A for a description of the normative format. The standard values are `"15min"`, `"24h"` and `"NA"`.

8.7.4.4 PM thresholds

Each TCA notification has an attribute thresholdType of type `PMThresholdType_T` which describes the threshold watermark (TWM) level, or threshold type for short, of the TCA. The IDL describing this data type is provided below (without comments).

```
module notifications
{
...
   enum PMThresholdType_T
   {
      TWM_HIGHEST,
      TWM_HIGH,
      TWM_LOW,
      TWM_LOWEST
   };
...
}
```

Figure 3 provides information on how watermark levels are used for raise/trigger and clear TCAs.

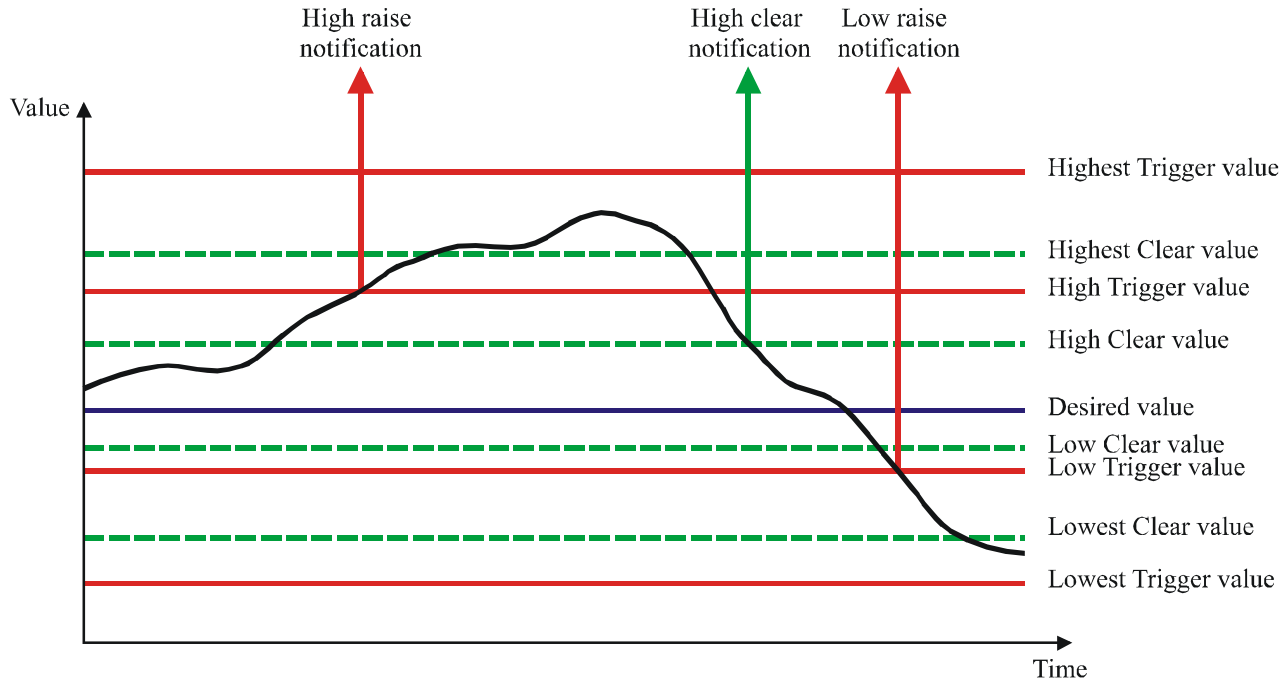

Figure 3 – PM threshold types

Note that the perceivedSeverity attribute (see clause 8.7.3.2) serves (mainly, in case of TCAs) as the trigger flag (i.e., it indicates whether a TCA is a clear or is a trigger/raise with or without severity).

The `TWM_HIGH` and `TWM_HIGHEST` types are used for TCAs that are raised when the measured value goes above the threshold. The `TWM_LOW` and `TWM_LOWEST` types are used for TCAs that are raised when the measured value goes below the threshold; they only apply to gauges since counters do not count down (but may wrap around). When there is one level of TCA notification triggers, only `TWM_HIGH` and/or `TWM_LOW` are used, and when there are two levels of TCA triggers, `TWM_HIGHEST` and/or `TWM_LOWEST` are used in addition.

A PM threshold, as part of a TCA, consists of a threshold type (`PMThresholdType_T`), a trigger flag (`PerceivedSeverity_T`), a threshold value (`float`) and a threshold unit (`string`). Refer to the IDL comments in Annex A for details. For rapid provisioning purposes, one or more PM thresholds may be associated to a PM parameter and such PM parameters with thresholds may be stored – per Layer rate, PM location and granularity – as a managed object (an instance of the MOC performance monitoring point (PMP) derived from `Common_T` – see clause 8.5). Then these managed objects could be configured and attached to measurement points and, for each of them, performance monitoring and/or threshold supervision would be enabled/disabled on demand.

8.7.4.5 TCA identification

TCAs are uniquely identified by the combination of the objectName, layerRate, pmParameterName, pmLocation, granularity and thresholdType fields. That is why these fields are grouped into the `TCAId_T` struct. The IDL describing this type is provided below (without comments).

```
module notifications
{
...
   struct TCAId_T
   {
      globaldefs::NamingAttributes_T objectName;
      transmissionParameters::LayerRate_T layerRate;
```

```
      PMParameterName_T pmParameterName;
      PMLocation_T pmLocation;
      Granularity_T granularity;
      PMThresholdType_T thresholdType;
   };
...
}
```

Refer to the IDL comments in Annex A for a description of the record members.

8.7.4.6 TCA notifications

The IDL describing the TCA notification type is provided below (without comments).

```
module notifications
{
...
   struct TCA_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      TCAId_T TCAId;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
      string nativeEMSName; // OPTIONAL
      PerceivedSeverity_T perceivedSeverity;
      float value; // OPTIONAL
      string unit; // OPTIONAL
      string additionalText; // OPTIONAL
      boolean isClearable;
      AcknowledgeIndication_T acknowledgeIndication; // OPTIONAL
   };
...
}
```

Refer to the previous clauses and Annex A for a description of the record members.

A TCA parameter, as part of a TCA, consists of all threshold crossing-related attributes of the TCA, namely PM parameter name (see clause 8.7.4.1), PM location (see clause 8.7.4.2), granularity (see clause 8.7.4.3), and PM threshold (see clause 8.7.4.4). The TCA parameter is considered to be associated with the specified Layer rate of the managed object emitting the TCA. For rapid provisioning purposes, one or more TCA parameters may be associated to a transmission layer and stored as a (technology-specific) profile (an instance of the MOC TCA parameter profile (TCAPP) derived from `Common_T` – see clause 8.5). Then these profile managed objects could be configured and associated with one or more Layer rates of measurement points, and in-service modification could be allowed for them.

8.7.5 Mixed lists of notifications

Recall from clause 8.7.1 that this framework recommends the use of OMG structured events and event batches for the conveyance of notifications from a managed system to managing systems. The IDL describing these types is defined by the OMG and reused in Annex A as follows.

```
module CosNotification
{
...
   // Define the Structured Event structure
   struct FixedEventHeader {
      _EventType event_type;
      string event_name;
   };
   struct EventHeader {
      FixedEventHeader fixed_header;
```

```
      OptionalHeaderFields variable_header;
   };
   struct StructuredEvent {
      EventHeader header;
      FilterableEventBody filterable_data;
      any remainder_of_body;
   }; // StructuredEvent
   typedef sequence<StructuredEvent> EventBatch;
...
}

module notifications
{
...
   typedef CosNotification::StructuredEvent Event_T;
   typedef sequence<CosNotification::StructuredEvent> EventList_T;
...
}
```

The preceding clauses have shown how `Event_T` is specialized to the standard notification types `OCN_T` etc., and further types can be defined by extensions of this framework. Event batches are therefore lists of structured events whose components are determined by the defined event types (and have either a common domain name such as `"tmf_mtnm"` or different domain names such as IRP document version number strings – see clause 8.3 of [ITU-T Q.816.2]). These lists are called mixed lists of notifications. Moreover, the composite notification ID structures `AlarmId_T` and `TCAId_T` have been introduced for the more sophisticated notification types ALM and TCA.

This clause concludes the description of the `notifications` module of the CORBA framework to support service-oriented interfaces. It summarizes the capabilities provided for use by operations that extend this framework and wish to convey mixed lists of notifications or notification IDs.

8.7.5.1 ALM and TCA notifications

To be able to distinguish the structures `Alarm_T` and `TCA_T` when conveyed as `Event_T`, the union `AlarmOrTCAIdentifier_T` is defined, which holds alarm IDs (`AlarmId_T`) and TCA IDs (`TCAId_T`) and is switched on the enum `AlarmTypeQualifier_T`. The union is then used to define lists of IDs for alarms and TCAs. This IDL is provided below (without comments).

```
module notifications
{
...
   enum AlarmTypeQualifier_T
   {
      ALARM,
      TCA
   };
...
   union AlarmOrTCAIdentifier_T switch (AlarmTypeQualifier_T)
   {
      case ALARM: AlarmId_T alarmId;
      case TCA: TCAId_T tcaId;
   };
...
  typedef sequence<AlarmOrTCAIdentifier_T> AlarmAndTCAIDList_T;
...
}
```

`AlarmAndTCAIDList_T` allows to treat mixed lists of alarm and TCA identifiers in a uniform way (e.g., as operation parameters for acknowledge purposes – see clause 8.7.5.3).

8.7.5.2 Event iterator interface

In order to allow the NMS to deal with retrieval of a large number of events (of mixed types), iterators are used for bulk retrieval of notifications. These iterators are instances of the interface `EventIterator_I`. See clauses 9.4.5 and 9.4.2 for the IDL and details how event iterators are used

with OMG event batches being the iterator batches and OMG structured events being the iterable managed entities. Each such entity encodes one of the standard notification types `OCN_T`, `ODN_T`, `AVC_T`, `SCN_T`, `FTS_T`, `BSE_T`, `HBE_T`, `ALM_T`, `TCA_T` (see the preceding clauses) or an event type defined by an interface specification that extends this framework.

8.7.5.3 Event acknowledgement

Reliability of event delivery from the managed system to managing systems is a very important requirement. When the OMG notification service is configured with EventReliability=Persistent and ConnectionReliability=Persistent (as recommended – see clause 8.3 of [ITU-T Q.816.2]), each event is guaranteed to be delivered to all managing systems registered to receive it at the time it was delivered from the managed system to the notification channel (within expiry limits). If the connection between the channel and a managing system is lost for any reason, the channel will persistently store any events destined for that managing system until either each event times out due to expiry limits or the managing system once again becomes available and the channel is subsequently able to deliver the events to all registered managing systems. In addition, upon restart from a failure, the channel will automatically re-establish connections to all clients that were connected to it at the time the failure occurred. Refer to clause 2.5.5.1 of [OMG 04-10-13] for further details.

Event acknowledgement allows the managing system to confirm the receipt of individual events, thereby enabling the managed system to discard these events before expiry limits are reached or to configure the notification service with EventReliability=BestEffort. It does require a mechanism to uniquely identify events, however, so that the managing system can check for duplicates and give feedback on received events. For this purpose, the Notification Service [OMG 04-10-13] introduces sequence numbers that are non-negative integers with wrap-around behaviour and applied by suppliers to identify unambigously structured events and event batches, and specifies an acknowledge operation for each supplier interface to be used by consumers to convey lists of sequence numbers to suppliers. The QoS properties DeliveryReliability, RetryInterval and Retries may then be used by suppliers to implement reliable event delivery (see clause 3.7.6 of [OMG 04-10-13]).

This framework specifies a lightweight approach to event acknowledgement that is independent of the related notification service capabilities but could be used in a supplementary way together with the notification service. It also allows for best-effort event unacknowledgement. In the first place, it defines a means to indicate, through a corresponding optional attribute, whether an event has been acknowledged or unacknowledged in the managed system. It secondly recommends the definition of an acknowledge operation and an unacknowledge operation to enable acknowledgement and unacknowledgement of events by managing systems. The approach is considered meaningful only for the ALM and TCA notification types since events of this type have a lifecycle.

`AcknowledgeIndication_T` describes the acknowledgement indication of certain events. Active and cleared ALM and TCA notifications have an optional attribute acknowledgeIndication of this type. The IDL describing this data type is provided below (without comments).

```
module notifications
{
...
   enum AcknowledgeIndication_T
   {
      AI_EVENT_ACKNOWLEDGED,
      AI_EVENT_UNACKNOWLEDGED,
      AI_NA
   };
...
}
```

Refer to the IDL comments in Annex A for a description of these values.

The acknowledgement flag indicates the state of an alarm or TCA with regard to acknowledgement as known to the managed system. To enable initiation of state transitions by a managing system interface, specifications that extend this framework may specify an acknowledge operation and an unacknowledge operation that shall comply with the following principles:

– The operation has an input parameter of type `AlarmAndTCAIDList_T` that specifies the events to be acknowledged or unacknowledged.

– The operation has an input parameter of type `globaldefs::NVSList_T` that specifies additional information, for example, a username so that the event can be marked as being attended by the user indicated, or some textual information supplied by the attending user (e.g., to inform other users in a whiteboard-like fashion through notepad entries).

– The operation has an output parameter of type `AlarmAndTCAIDList_T` that specifies the events that failed to become acknowledged or unacknowledged.

Alarm and TCA acknowledgement and unacknowledgement may then be initiated by one or more managing systems and by the managed system itself. For each successfully acknowledged or unacknowledged event, the managed system shall supply a corresponding notification with updated acknowledgeIndication attribute (which is consumed by all registered managing systems).

(O) NOTIF-4 The managed system may support event acknowledgement for alarms and TCAs. In this case it shall implement the acknowledgeIndication attribute for these event types and operations for acknowledgement and unacknowledgement of active events based on alarm IDs and TCA IDs as specified above. When the deployed notification service supports event acknowledgement based on sequence numbers and related QoS properties, the managed system may use this capability for reliable alarm and TCA delivery when the managing systems acknowledge these events.

Alarms and TCAs are either active or cleared (which need not mean terminated). In the presence of event acknowledgement, they can be acknowledged or unacknowledged and any combination with active and cleared is possible but cleared alarms can neither be acknowledged nor unacknowledged. Active events can be distinguished as pending and reportable, and reportable events can be classified according to severity levels. Moreover, event reporting could be inhibited (for example, when a resource is in need of repair) with or without prescribed conditions when it will be turned on again according to a well-defined alarm reporting control (ARC) policy. These considerations have led to the definitions of alarm status and ARC state in [b-ITU-T X.731], [ITU-T X.721] and [ITU-T M.3100], which do not take into account the possibility of event (un)acknowledgement, however.

The progression of the alarm status (without standard state transitions) to an event lifecycle state with standard state transitions that allows for acknowledgement/unacknowledgement and severity changes is for further study including its relationship to the [b-ITU-T X.731], [ITU-T X.721] lifecycle state of managed objects. Currently the perceivedSeverity and acknowledgeIndication attributes collectively provide such information for reportable alarms and TCAs. The admissible states and transitions of an event lifecycle state would depend on the managed system's event processing mode of operation that would specify the degree of acknowledgement support. Such modes would be encoded as capabilities supported across the OS-OS interface as specified in clause 9.3.2.7. For example, `"Supports acknowledgement mode 0"` could mean "The EMS does not support event acknowledgement.", `"Supports acknowledgement mode 1"` could mean "The alarm and TCA list contains only those events which are present in the EMS managed network but which are not acknowledged. Once an event gets acknowledged it is automatically cleared and removed from the list.", `"Supports acknowledgement mode 2"` could mean "The alarm and TCA list contains all events that are currently active (and reportable) in the EMS managed network, whether acknowledged or unacknowledged. An event list record will be removed when the corresponding event is cleared in the network and a cleared notification (with unchanged acknowledgeIndication attribute) has been emitted. If an acknowledged event is raised again since it

changes its perceived severity to a new value <> cleared, it gets automatically unacknowledged (same lifecycle)."

8.7.5.4 On-demand pulling of events

[ITU-T Q.821.1] provides the rationale, some exemplary justifications and itemized requirements for reconciliation or synchronization of active alarms between the managed system and managing systems, and specifies a fine-grained alarm synchronization service, the interface `EnhancedCurrentAlarmSummaryControl`, and a corresponding X.780.1 façade interface, which both offer an iterator-enabled operation for scoped retrieval of current alarms where, according to [ITU-T Q.821], an alarm is termed current if it is active and has not yet been cleared.

This framework specifies a more lightweight approach to alarm and TCA reconciliation based on probable cause and perceived severity exclusions. Any interface specification that extends this framework shall specify getAll`<scope>`ActiveAlarms operations for on-demand pulling of active ALM and TCA notifications that comply with the following principles:

– The operation has an input parameter of type `ProbableCauseList_T` that specifies the probable causes to be excluded from the results list.

– The operation has an input parameter of type `PerceivedSeverityList_T` that specifies the perceived severities to be excluded from the results list.

– The operation uses an event iterator (see clauses 8.7.5.2 and 9.4.5) as described in clause 9.3.2.2 (where the names managed entities have to be replaced by events managed entities).

– The operation has an output parameter of type `EventList_T` that holds the first batch of alarms or TCAs to be returned.

`<scope>` refers to the scope of the targets that are included in the event pulling (e.g., a single NE, all network elements under control of the EMS, all NE-independent events).

If event acknowledgement is supported, getAll`<scope>`UnacknowledgedActiveAlarms operations for on-demand pulling of unacknowledged (see clause 8.7.5.3) active ALM and TCA notifications shall be specified subject to the same principles.

(R) NOTIF-5 The managed system shall support event synchronization for alarms and TCAs. It shall implement the operations for on-demand pulling of active events as specified above. If it supports event acknowledgement according to **NOTIF-4**, it shall also implement the specified operations for on-demand pulling of unacknowledged active events.

The definition of further pulling operations for events is for further study. For example, it could be required to define alarm and TCA pulling operations with inclusion of specified probable causes or perceived severities and to standardize the offered choices for the scope of the pulling operations.

9 Providing service-oriented façade interfaces

As described in clause 7.2.1, the SO framework uses SO façades to provide access to, and generally control, SO managed objects. The IDL details of service-oriented managed object classes are described in clause 8. This clause describes the IDL details of the service-oriented façade interfaces. It also describes auxiliary IDL interfaces used by façade objects. The normative modelling IDL for SO interfaces and MOCs is included in Annex A.

9.1 Façade instantiation

This clause defines the requirements that managed systems shall follow when providing service-oriented façade interfaces and objects for accessing service-oriented managed objects.

(R) SOFAÇADE-1 A managed system shall provide a service-oriented façade interface and at least one service-oriented façade interface instance (object) for each service-oriented managed object class defined according to OBJECT-1. Such MOC is said to be owned by the façade interface. An SO façade interface may own one or more (related) service-oriented MOCs.

(R) SOFAÇADE-2 A managed system should provide only one service-oriented façade object for a given service-oriented MOC (see clause 7.2.1.1). If more than one façade object is provided for a given MOC (e.g., since it can have an extremely large number of instances), managed objects accessible through one façade object shall not be accessible through any other and details on how to identify which façade object to use to access a given managed object need to be specified.

(R) SOFAÇADE-3 All SO façades are created and destroyed by the managed system. An SO façade shall exist for the entire lifecycle of any of the SO managed objects owned by it. No notifications are sent when an SO façade is created or deleted. Therefore, a managing system will become aware of the existence of a new façade when managed objects that are accessible through the new façade begin to appear. If the managed system supports the session service, the available SO façades can be determined at runtime (see SESSION-4 of [ITU-T Q.816.2]).

(R) SOFAÇADE-4 Each SO façade operation that is directed at a specific managed object will contain the name of that managed object in the first parameter of the operation. The façade shall logically invoke that operation on the named managed object and provide the results, including exceptions. If the managed object is not accessible through that façade, the façade shall raise a ProcessingFailureException exception in response to the operation; the exception type in this exception shall be set to the `EXCPT_ENTITY_NOT_FOUND` code defined in the IDL in Annex A.

(R) SOFAÇADE-5 Service-oriented façades shall allow for bulk retrieval of objects and attributes. Such bulk retrieval shall be realized by using in get operations precisely tailored iterators, which are based on the generic iterator approach according to SOFAÇADE-7 and the definition of suitable iterable managed entities and associated iterator batches (see clause 9.4).

9.2 Getting and setting objects and attributes

The approach to setting and getting attributes is tuned to the purpose of the specific attributes. This clause sets out the approach for each attribute discussed in clause 8 above.

The general approach to retrieval of the values of the managed object attributes is a part of the object retrieval (see clause 6.7). The framework allows for operations that retrieve:

– Solely the names of the instances of a particular class or set of classes of managed objects.
  - This does not retrieve any of the attribute values other than the name.
– A managed object with a specified name.
  - This retrieves all of the attributes of the managed object specified by name.
– The set of managed objects satisfying some particular criteria.
  - This retrieves all of the attributes of each of the qualifying managed objects.
– All of the managed objects of a particular class.
  - This retrieves all of the attributes of the managed objects specified by MOC.
– Some defined aspects of each object in a set of managed objects satisfying a specified criteria.
  - This retrieves a defined subset of the attributes of each managed object satisfying the criteria.

The service-oriented framework specifies the following generic accessor functions that can be applied to any service-oriented managed object or set of such objects:

– get all names of the SO managed objects currently being owned by the SO façade;

– get all SO managed objects currently being owned by the SO façade;

– get a managed object (i.e., all its attributes) that is specified by its name.

The framework also allows for:

– use of iterators to handle efficiently potentially large result sets;

– retrieval of containment relationships.

There are several approaches to setting of the values of the managed object attributes. The approach is chosen to suit the specific case of object class, value set and implementation restriction. If the specific approach is not supported for the specific case then the operation will be rejected and an exception will be thrown (see clause 8.6). The approach for any particular managed object class will be some combination of the following cases:

– The value may be set during creation by the managing system (e.g., NMS).

– The value may be set during creation by the managed system (e.g., EMS).

– The value may be modified at some point during the lifecycle of the object by the managed system and this shall be identified by a notification (AVC or SCN, see clause 8.7).

– The value may be modified at some point during the lifecycle of the object by the managing system, using a specific operation to modify the value, and this may be captured by a notification (AVC or SCN, see clause 8.7).

The following cases apply to the attributes identified in clause 8:

– **name**: The name may be chosen during the creation by the EMS or NMS depending upon the specific managed object class behaviour. In the cases where the NMS may choose the name, the EMS may reject the request for object creation if the name is not unique. The name shall not be changed at any stage in the lifecycle of the managed object. Note that there are no operations provided to change the value of the name and hence there is no exception defined.

– **userLabel**: When an object is created by the NMS, the NMS may specify the userLabel for the object and indicate whether the EMS is required to ensure that the userLabel is unique within its visibility. The userLabel may also be "set" by the NMS through the setUserLabel operation of the interface `Common_I` (see clause 9.3.2).

– **nativeEMSName**: When an object is created, the EMS may provide a nativeEMSName. After an object has been created, the nativeEMSName may be changed by the NMS, if the EMS supports this feature, using the setNativeEMSName operation of the interface `Common_I` (see clause 9.3.2).

– **owner**: When an object is created, the EMS may provide an owner. After an object has been created, the owner may be changed by the NMS, if the EMS supports this functionality, using the setOwner operation of the interface `Common_I` (see clause 9.3.2).

– **additionalInfo**: When an object is created by the NMS, the NMS may provide initial attribute values via the additionalInfo structure of the specific create operation. The setting of additionalInfo is best effort (except where specified otherwise for a particular parameter) and the output of the operation identifies the values which were actually set. After an object has been created, the additionalInfo may be adjusted using the setAdditionalInfo operation of the interface `Common_I` (see clause 9.3.2). This operation supports addition to and removal from the list as well as adjustment of the value of a parameter already in the list. The adjustment of additionalInfo values is best effort (except where specified otherwise for a particular parameter).

– **LayeredParametersList_T**: When a layered object is created by the NMS, the NMS may provide initial attribute values via the attribute defined as being of type LayeredParametersList_T in the specific create operation. The setting of an attribute of this type is best effort (except where specified otherwise for a particular parameter) and the output of the operation identifies the values which were actually set. After an object has been created, the attribute may be adjusted using any explicit operation defined for that object provided by a specialized operation via the appropriate interface. The operation must abide by the constraints and specifications of this framework but definition of MOC-specific operations for layered objects is outside the scope of the framework.

Refer to clause 10.1 for general guidelines for lightweight reading and writing of attributes and differentiation from the fine-grained and coarse-grained modelling approaches.

9.3 Service-oriented façade interface base class `Common_I`

This clause describes a service-oriented façade base class that is defined in the IDL in Annex A. All service-oriented façade interfaces provided on a managed system shall inherit, either directly or indirectly, from this superclass interface. The façade superclass provides a set of basic operations/capabilities that all service-oriented façade interfaces shall support to be usable with this framework.

(R) SOFAÇADE-6 The façade interfaces defined for a service-oriented CORBA interface shall inherit (directly or indirectly) from the `Common_I` interface described below and defined in the CORBA IDL in Annex A. The capabilities described below shall be supported. All operations of the façade interfaces shall raise the exception `globaldefs::ProcessingFailureException` and provide IDL comments for the exception types (and error reasons) that may occur.

9.3.1 Common façade basic capabilities

The common capabilities that all service-oriented façade interfaces must support are:

– For each owned managed object class, an operation that provides the names of all currently instantiated managed objects of this MOC.

– For each owned managed object class, an operation that provides all currently instantiated managed objects of this MOC.

– For each owned managed object class, an operation that provides a CORBA struct object containing all of the readable attributes of a managed object (of this MOC) specified by name.

– For each settable attribute of the base managed object class `Common_T`, an operation that modifies the value of this attribute for a managed object specified by name, i.e.:

  • An operation that sets the userLabel of a managed object.

  • An operation that sets the nativeEMSName of a managed object.

  • An operation that sets the owner of a managed object.

  • An operation that sets the additionalInfo of a managed object.

– An operation that provides the capabilities supported by an individual derived service-oriented façade (except the common capabilities listed here) where a capability is specified by either a scoped operation name or in a way that depends on the derived façade interface.

9.3.2 Common façade IDL

The IDL describing the `Common_I` interface (without comments) is provided below.

```
module common
{
...
   interface Common_I
   {
...
   void getAllManagedObjectNames(
         in unsigned long how_many,
         out globaldefs::NamingAttributesList_T nameList,
         out globaldefs::NamingAttributesIterator_I nameIt)
        raises(globaldefs::ProcessingFailureException);
...
   void getAllManagedObjects(
         in unsigned long how_many,
         out ManagedObjectList_T moList,
         out ManagedObjectIterator_I moIt)
        raises(globaldefs::ProcessingFailureException);
...
   void getManagedObject(
         in globaldefs::NamingAttributes_T objectName,
         out Common_T mo)
        raises(globaldefs::ProcessingFailureException);
...
   void setNativeEMSName(
         in globaldefs::NamingAttributes_T objectName,
         in string nativeEMSName)
        raises(globaldefs::ProcessingFailureException);
...
   void setUserLabel(
         in globaldefs::NamingAttributes_T objectName,
         in string userLabel,
         in boolean enforceUniqueness)
        raises(globaldefs::ProcessingFailureException);
...
   void setOwner(
         in globaldefs::NamingAttributes_T objectName,
         in string owner)
        raises(globaldefs::ProcessingFailureException);
...
   void getCapabilities(
         out CapabilityList_T capabilities)
        raises(globaldefs::ProcessingFailureException);
...
   void setAdditionalInfo(
         in globaldefs::NamingAttributes_T objectName,
         inout globaldefs::NVSList_T additionalInfo)
        raises(globaldefs::ProcessingFailureException);

   };

}
```

These interfaces are described in the following clauses and defined in Annex A within the IDL module `common`. The module `common` depends only on the module `globaldefs`, which is stand-alone and uses only the minimum IDL repertoire described in clause 8.1.

9.3.2.1 getAllManagedObjectNames() operation

The getAllManagedObjectNames operation takes the requested maximum first batch size how_many, provides the first batch nameList with how_many, or less, managed object names of type NamingAttributes_T, and also provides either the IOR nameIt of a NamingAttributes iterator to retrieve further batches or `CORBA::Object::_nil` if no more batches are available. The operation may raise the unique exception object ProcessingFailureException (see clause 8.6) with all admissible, non-OMG standard exception types being specified as IDL comments.

9.3.2.2 getAllManagedObjects() operation

The getAllManagedObjects operation takes the requested maximum first batch size how_many, provides the first batch moList with how_many, or less, generic managed objects of type `Common_T`, and also provides either the IOR moIt of a ManagedObject iterator to retrieve further batches or `CORBA::Object::_nil` if no more batches are available. The operation may raise the unique exception object ProcessingFailureException (see clause 8.6) with all admissible, non-OMG standard exception types being specified as IDL comments.

9.3.2.3 getManagedObject() operation

The getManagedObject operation takes the target managed object name objectName and provides the generic managed object mo of type `ManagedObject_T`. The operation may raise the unique exception object ProcessingFailureException (see clause 8.6) with all admissible, non-OMG standard exception types being specified as IDL comments.

9.3.2.4 setNativeEMSName() operation

The setNativeEMSName operation takes the target managed object name objectName and requested nativeEMSName and sets the corresponding attribute of the corresponding object accordingly. The operation may raise the unique exception object ProcessingFailureException (see clause 8.6) with all admissible, non-OMG standard exception types being specified as IDL comments, including `EXCPT_NOT_IMPLEMENTED` if the managed system does not support the operation.

9.3.2.5 setUserLabel() operation

The setUserLabel operation takes the target managed object name objectName and requested userLabel and sets the corresponding attribute of the corresponding object accordingly. In addition, the Boolean parameter enforceUniqueness may be used to indicate whether the managed system is expected to ensure uniqueness of the userLabel within the realm of the managed system. The operation may raise the exception object ProcessingFailureException (see clause 8.6); the potential exception types are specified as IDL comments, including `EXCPT_NOT_IMPLEMENTED` if the managed system does not support the operation. If the managed system determines that the *userLabel* is not unique within its scope of visibility, it is required to reject the operation with the exception type `EXCPT_USERLABEL_IN_USE` (if it supports enforcement of uniqueness).

9.3.2.6 setOwner() operation

The setOwner operation takes the target managed object name objectName and requested owner and sets the corresponding attribute of the corresponding object accordingly. The operation may raise the unique exception object ProcessingFailureException (see clause 8.6) with all admissible, non-OMG standard exception types being specified as IDL comments, including `EXCPT_NOT_IMPLEMENTED` if the managed system does not support the operation.

9.3.2.7 getCapabilities() operation

The getCapabilities operation provides the capabilities supported by this service-oriented façade object. A capability is specified as the name of an NVS pair by either a scoped operation name, with disallowed exception type `EXCPT_NOT_IMPLEMENTED`, or in a way that depends on the purpose of the derived façade interface. The support of the capability is specified by the corresponding value of the NVS pair. Refer to the IDL comments of `Capability_T` for details. The operation may raise the exception object ProcessingFailureException (see clause 8.6) with the identifier `EXCPT_INTERNAL_ERROR` as the only admissible exception type (see IDL comments).

NOTE – Refer to clause 9.1.2.2 of [ITU-T Q.816.2] for the complementary capability features offered by the session service regarding the service-oriented façades supported by the managed system.

9.3.2.8 setAdditionalInfo() operation

The setAdditionalInfo operation takes the target managed object name objectName and requested additionalInfo and sets the corresponding attribute of the corresponding object accordingly, and does this generally on a best effort basis.

As the additionalInfo attribute is a parameter list, the operation supports addition to and removal from the list as well as adjustment of the values of parameters already in the list. As an input, only the list of parameters to be removed, added or changed shall be provided. If an entry is to be removed, "-" shall be specified as a value. If a parameter is specified that is currently not part of the additionalInfo attribute of the specified object, that parameter is added by the managed system with the specified value. If a parameter specified is currently part of the list then the value of that list item is simply updated with the value specified by the request (which must not be "-" as this would indicate removal of the entry). The managed system may reject removal and addition requests by throwing the appropriate exception, which will then be a rejection of the entire operation. The adjustment of additionalInfo values is best effort (except where specified otherwise for a particular parameter) and the response to the request will indicate which values were adjusted. Refer to clause 9.2 and the IDL comments for further semantic details.

The operation may raise the unique exception object ProcessingFailureException (see clause 8.6) with all potential, non-OMG standard exception types being specified as IDL comments, including `EXCPT_NOT_IMPLEMENTED` if the managed system does not support the operation.

9.4 Iterator interfaces

In order to allow the managing system to deal with a large number of entities returned by a retrieval operation, the iterator design pattern[5] is used. Iterators provide a mechanism to retrieve data batch by batch – the ideal size of a batch is specified by the requesting managing system.

In a typical case, a managing system requests a list of managed entities (i.e., managed objects, or attributes, or other manageable information units of a well-defined type) (see, for example, the getAllManagedObjects operation in clause 9.3.2). With this request, the managing system can specify the maximum number of entities it can handle in the first reply, but the number returned by the managed system may be less than this. The managed system gets a snapshot, or a dynamic hit list, of the current entities to be returned: the response will contain a list with (at most) the specified number of entities and a reference to an iterator instance, whose operations can subsequently be used by the managing system to access further batches of the snapshot or dynamic hit list. The iterator may return less than the requested batch size, balancing the efficiency of returning results in a large batch with the possible need to block until more results become available.

Whilst the managed system controls the lifecycle of an iterator, a destroy operation is provided if the managing system wants to stop retrieving results before reaching the last iteration. Upon returning the last result, the iterator usually shall destroy itself. It may also be destroyed by the managed system if it is unused for an unreasonably long period of time. The managed system will usually limit the number of simultaneously instantiated iterators.

(R) SOFAÇADE-7 The iterators used for bulk retrieval according to **SOFAÇADE-5** shall support the capabilities described below. Such iterators shall be based on the generic iterator interface IDL described below and use precisely tailored iterable managed entities and iterator batches.

---

[5] The iterator design pattern [b-GAMMA] is a behavioural pattern with the intent to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. An iterator, also known as a cursor, takes on the responsibility for access and traversal of an aggregate object such as a list and relieves its godchild of the need to expose its internal structure for that purpose. An iterator is instantiated by the managed system, usually together with its godchild or by reusing a member of an iterator pool, and never instantiated by the managing system.

9.4.1 Iterator interfaces basic capabilities

The capabilities that all iterator interfaces must support are:

- An operation that returns the current total number of elements in the iterator.
- An operation that provides the next batch of elements, the maximum batch size being an input parameter, and returns a Boolean value to indicate whether still more elements are available than the actual elements provided (which may be less than the maximum batch size).
- An operation that deletes the iterator object.

NOTE – These descriptions and the descriptions of the operations getAllManagedObjectNames and getAllManagedObjects of the `Common_I` interface (see clauses 9.3.2.1 and 9.3.2.2) collectively demonstrate the generic usage of iterators in a concise way (see also **SOFAÇADE-5** and **SOFAÇADE-7**).

9.4.2 Generic iterator interface IDL

The generic iterator approach defines a generic IDL struct or value type to represent an iterable managed entity type, and uses a sequence of this type, or the IDL type `any`, as the output parameter type for the next batch operation next_n of a generic iterator interface (i.e., as the generic iterator batch type). Specific iterator interfaces are then obtained by adding specific attributes to, or inheriting from, the generic entity type, and inheriting from the generic iterator interface.

The IDL describing the generic iterator interface (without comments) is provided below.

```
module globaldefs
{
...
   struct IterableManagedEntity_T
   {
      // add any managed entity type-specific attributes here
      globaldefs::NVSList_T additionalInfo;
   }
...
   typedef sequence<IterableManagedEntity_T> IterableManagedEntityList_T;
...
   interface ManagedEntityIterator_I
   {
   boolean next_n(
            in unsigned long how_many,
            out IterableManagedEntityList_T managedEntityList)
           raises(globaldefs::ProcessingFailureException);

   unsigned long getLength()
                raises(globaldefs::ProcessingFailureException);

   void destroy()
        raises(globaldefs::ProcessingFailureException);

   };
...
}
```

As an alternative, an `any` type could be used instead of `IterableManagedEntityList_T` to define the output parameter of the next batch operation together with the enforcement that the actual type be the specific `IterableManagedEntityList_T` with all entity-specific attributes:

```
   boolean next_n(
            in unsigned long how_many,
            out any managedEntityList
                // the actual type is a specific IterableManagedEntityList_T
                 )
           raises(globaldefs::ProcessingFailureException);
```

Alternatively, value types could be used instead of structs to define iterable managed entities:

```
valuetype IterableManagedEntity_V
{
   // add the managed entity type-specific attributes here
}

typedef sequence<IterableManagedEntity_V> IterableManagedEntityList_T;
```

The generic iterator interface ManagedEntityIterator_I is described in the following clauses and defined (without alternatives) in Annex A within the CORBA module `globaldefs`.

NOTE – The IDL definitions of the operations getAllManagedObjectNames and getAllManagedObjects of the `Common_I` interface (see clause 9.3.2 and Annex A) provide a generic IDL definition of iterator usage.

Specific iterator interfaces that apply the generic iterator approach to specific iterable managed entity types include the following examples from ITU-T and (one from) OMG:

– `globaldefs::NamingAttributesIterator_I`:

  The managed entities of this iterator are service-oriented names (see clause 9.4.3).

– `common::ManagedObjectIterator_I`:

  The managed entities of this iterator are service-oriented managed objects (see clause 9.4.4).

– `notifications::EventIterator_I`:

  The managed entities of this iterator are OMG structured events (see clause 9.4.5).

– `omg.org::CosNaming::BindingIterator`:

  The managed entities of this iterator are CORBA name, or name component, bindings (see [OMG 04-10-03] and clause 6.4 of [ITU-T Q.816.2]).

– `itut_x780::AttributesGetResultIterator`:

  The managed entities of this iterator are of type `AttributesGetResultType`. These are coarse-grained managed objects, derived from `ManagedObjectValueType`, together with indications for valid attributes, and the iterator batches are used for bulk attribute retrieval (see clause 8.2.8 of [ITU-T X.780.1] and Appendix I of Amendment 1 to [ITU-T X.780.1]).

– `itut_x780::GetResultsIterator`:

  This iterator is used for scoped get operations (see clause 7.4.1.3 of [ITU-T Q.816]).

– `itut_x780::UpdateResultsIterator`:

  This iterator is used for scoped update operations (see clause 7.4.1.4 of [ITU-T Q.816]).

– `itut_x780::DeleteResultsIterator`:

  This iterator is used for scoped delete operations (see clause 7.4.1.5 of [ITU-T Q.816]).

– `itut_x780::NameIterator`:

  The managed entities of this iterator are CORBA names. It is used by the containment service to retrieve containment relationships. Refer to clause 9.6.2 of [ITU-T Q.816.1] for details.

– `itut_q821d1::AlarmSynchronizationDataIterator`:

  The managed entities of this iterator are of type `AlarmSynchronizationDataSeqType`. These are OMG event batches. The iterator is used by the `alarmSynchronization` operations of the fine-grained and coarse-grained alarm synchronization services to return current alarms to the manager. Refer to clause 10.7.3 of [ITU-T Q.821.1] for details.

It should be noted though, that the OMG binding iterator and the ITU-T iterators of the fine-grained and coarse-grained framework modules `itut_x780` and `itut_q821d1` define slightly different operation signatures for the next batch operation and do not know the getLength operation.

9.4.2.1 next_n() operation

The next_n operation takes the requested maximum next batch size how_many, provides the next batch managedEntityList with how_many, or less, managed entities of type `IterableManagedEntity_T`, and returns a boolean value to indicate whether still more entities are available than the actual entities provided this time (and the previous times).

9.4.2.2 getLength() operation

The getLength operation returns the current total number of entities in this iterator.

In some cases, it may not be feasible for the managed system to obtain the total number of entities that can be returned by the iterator. For example, the managed system may have to first fetch all of the data that has been requested (making a local copy) and then count the number of entities being iterated over, which could prove to be particularly inefficient. Or the managed system may be able to obtain a count of the number of entities but, due to concurrent modifications, the actual number may vary over the time of the iteration. Independently of the getLength operation, a managing system can always determine when all data has been collected by repeatedly fetching chunks of data until the iterator indicates that no further data remains. In the cases where the managed system is not able to suitably determine the return value of the getLength operation, the operation shall respond with the `EXCPT_CAPACITY_EXCEEDED` exception type (see clause 8.6).

9.4.2.3 destroy() operation

The destroy operation deletes this iterator object. It shall usually not throw an exception.

9.4.3 NamingAttributes iterator interface NamingAttributesIterator_I

The IDL describing the specific NamingAttributes iterator interface (without comments) is provided below.

```
interface NamingAttributesIterator_I
{
boolean next_n(
         in unsigned long how_many,
         out NamingAttributesList_T nameList)
        raises(globaldefs::ProcessingFailureException);

unsigned long getLength()
              raises(globaldefs::ProcessingFailureException);

void destroy()
     raises(globaldefs::ProcessingFailureException);

};
```

This interface is described below and defined in Annex A within the IDL module `globaldefs`.

The iterable managed entity is a generic service-oriented name defined by:

```
typedef NVSList_T NamingAttributes_T;
```

where NVSList_T is defined in clauses 8.2 and 8.3, and the iterator batch is defined by:

```
typedef sequence<NamingAttributes_T> NamingAttributesList_T;
```

For the next_n operations, the description of clause 9.4.2.1 applies with IterableManagedEntity_T replaced by NamingAttributes_T. For the getLength operation and the destroy operation, clauses 9.4.2.2 and 9.4.2.3 apply unchangedly.

9.4.4 ManagedObject iterator interface ManagedObjectIterator_I

The IDL describing the specific ManagedObject iterator interface (without comments) is provided below.

```
interface ManagedObjectIterator_I
{
boolean  next_n(
          in unsigned long how_many,
          out ManagedObjectList_T moList)
         raises(globaldefs::ProcessingFailureException);

unsigned long getLength()
              raises(globaldefs::ProcessingFailureException);

void destroy()
     raises(globaldefs::ProcessingFailureException);

};
```

This interface is described below and defined in Annex A within the CORBA module `common`.

The iterable managed entity is a generic service-oriented managed object of type `Common_T` (i.e., without any MOC-specific attributes), and the iterator batch is defined by:

```
typedef sequence<Common_T> ManagedObjectList_T;
```

For the next_n operation, the description of clause 9.4.2.1 applies with IterableManagedEntity_T replaced by ManagedObject_T. For the getLength operation and the destroy operation clauses 9.4.2.2 and 9.4.2.3 apply unchangedly.

9.4.5 Event iterator interface EventIterator_I

The IDL describing the specific Event iterator interface (without comments) is provided below.

```
interface EventIterator_I
{
boolean next_n(
         in unsigned long how_many,
         out EventList_T eventList)
        raises(globaldefs::ProcessingFailureException);

unsigned long getLength()
              raises(globaldefs::ProcessingFailureException);

void destroy()
     raises(globaldefs::ProcessingFailureException);

};
```

This interface is described below and defined in Annex A within the module `notifications`.

The iterable managed entity is a CosNotification::StructuredEvent (see clause 8.7) and the iterator batch is defined by:

```
typedef sequence<CosNotification::StructuredEvent> EventList_T;
```

For the next_n operation, the description of clause 9.4.2.1 applies with IterableManagedEntity_T replaced by CosNotification::StructuredEvent. For the getLength operation and the destroy operation clauses 9.4.2.2 and 9.4.2.3 apply unchangedly.

10 Service-oriented CORBA modelling guidelines

[ITU-T X.780] and [ITU-T X.780.1] specify modelling guidelines for defining fine-grained and coarse-grained CORBA TMN interfaces that are strongly inspired from the X.721 model and the X.722 guidelines (GDMO/ASN.1). The modelling guidelines of the service-oriented framework are

independent of the X.721 model and any other information model. The term "modelling" generally refers to all deliverables of a TMN interface specification and encompasses the engineering of:

– requirements/use cases;
– analysis/UML;
– design/CORBA IDL;
– conformance/ICS proformas; and
– supporting documentation (e.g., name/value pairs, layer rates, probable causes, PM parameters) for the purpose of consistent and sustainable interface progression.

Over and above that, in the context of service-oriented TMN interface development, the term "modelling guidelines" also refers to the statement of input requirements for progression and standardized applications of the service-oriented framework. Such guidelines would enable contributions of enhancement proposals for the framework and its applications, which are then discussed and consented/rejected at meetings of the ITU-T and application SDOs/forums (e.g., TM Forum, 3GPP/3GPP2) (see also conformance point 16 in clause 13.2.1). Some examples are:

– A prioritized list/summary of the recommended additional IDL repertoire and foundation IDL beyond Annex A and Annex A of [ITU-T Q.816.2] (see clause 8.1) (e.g., simplest form of `valuetype`, constants beyond those used already – see clause A.4, standardized managed object classes) including the intended context of usage. This significant guideline is for further study.
– Guidelines may be stated as rules or conventions with regard to model objectives or writing style. Refer to clauses 10.1 and 11 for some fundamental principles and style idioms.
– Whilst a managed system claiming conformance to the service-oriented CORBA framework may choose one of the four alternatives to specify MOCs described in clause 6.2, for the purpose of achieving very lightweight use of CORBA, it is strongly recommended that managed objects be represented as IDL `structs`. The façade objects shall be represented as CORBA `interfaces` (i.e., CORBA objects). Refer to clause 10.2 for this guideline.
– Consistent naming of managed objects and façade objects is of prime importance in real life environments. Refer to clause 10.3 for comprehensive general and specific naming guidelines.
– Clauses 8.4 and A.3 provide the generic foundation of the multi-technology capability of the service-oriented CORBA framework but do not advise how to model the telecom transmission technologies of a concrete environment through identification of individual layers, their characteristic and adapted information, and their adaptation and trail termination functions. Refer to clause 10.4 for related concrete guidelines and general considerations.
– Comprehensive guidelines for the modelling of IDL extensions are provided in clause 10.5.
– Additional high-level considerations with regard to modelling objectives of major interest are indicated in clause 10.6 but left for further study. They aim at the development of a two-part service-oriented meta-model for the state and behaviour of all sort of managed objects. Such a meta-model would facilitate the design of any kind of multi-technology OS-OS interface.

The elaborated and indicated CORBA modelling guidelines of this clause provide a convenient starting point for further standardized progression of the service-oriented CORBA framework. While clauses 8 and 9 and the IDL in Annex A are a normative part of the framework (subject to appropriate conformance profiling as stated in clause 13), many parts of this clause are informative. However, the key principles of object naming and IDL extensions are normative as well.

10.1 Rules and conventions

The service-oriented IDL model and interactions are designed primarily to suit the cases where two "intelligent" applications, such as a managing system and a managed system, both with significant data repositories are interacting in an "align and notify of change" pattern, similar to the observer design pattern (see clause 9.1.1 of [ITU-T Q.816.2]) and by using the façade design pattern (see clause 6).

The modelling approach is based upon a number of fundamental principles. The following list of such principles is not exhaustive but is a starting point. Completion and details are for further study.

– Generalized classes:
  - Minimized classification in favour of generalized classes to maximize the survivability of the interface through subtle change.
  - Identification of the boundaries of classes that appear to be long-term persistent (e.g., transmission layer is not a determiner of class).
  - With regard to managed object classes, emphasis of the role of containment and corresponding semantics as opposed to inheritance (i.e., there is a need for a rich containment hierarchy while class inheritance is less important).
– Generalized attributes and extensibility:
  - Use of name/value pairs with a "name=value" semantics wherever meaningful.
  - Use of user-definable transmission layers.
– Patterns in the problem space:
  - Design of coarse-grained classes that minimize the number of instances of objects reported across the interface by encapsulating multiple essential entities into "large" well-structured single classes.
  - While the multiplicity may well be variable, the rules for encapsulation of the entities must be clear and deterministic.
  - The encapsulation of entities in coarse-grained classes should not lose essential information or structure (e.g., layered terminations in a TP).
– Minimized relationships:
  - Design of one-way relationships where the reverse can be determined by the application.
  - Where there are multiple navigations through a set of entities, consideration of reporting of only one navigation where reasonable.
– Minimized reporting of instances:
  - Reporting of only such instances that provide off-normal information.
  - On-demand capabilities for the reporting of normal and detailed information.
– Concrete naming of entities:
  - Naming should be based on concrete entities in the problem space where possible.
  - Naming should not change over time and be deterministic wherever possible.
  - Refer to clause 10.3 for naming conventions that can be applied to any problem space.
– Minimized operations.
– Minimized filters.
– Deterministic procedures for interface progression.

These principles have been applied to design the lightweight use of IDL described in clause 8 "Service-oriented object model IDL" and clause 9 "Providing service-oriented façade interfaces",

which together describe Annex A "Service-oriented modelling IDL". Another example of lightweight CORBA IDL design is provided by clause 9.1 of [ITU-T Q.816.2] "Session service", which describes Annex A of [ITU-T Q.816.2] "Service-oriented framework support services IDL".

The applicability of certain rules and conventions of [ITU-T X.780] and [ITU-T X.780.1], as far as they are independent of the GDMO/ASN.1 model, to the service-oriented framework is for further study.

Further rules for further study could be considerations with regard to getting/accessing and setting/mutating attributes of managed objects through façades (e.g., dependency on the MOC alternative according to clause 6.2.2) and corresponding modelling principles (e.g., differentiation to the sophisticated scoped and filtered approach to attribute access of the fine-grained and coarse-grained frameworks). Such studies could start from clause 9.2 and the introduction to clause 8.

10.2 Superclasses of service-oriented managed objects and façade objects

The service-oriented framework is based on the SOA principle of separating the definition and storage of (shared) data from their delivery and maintenance (see clause 7.2.1.2 of [ITU-T Q.816.2]). Service-oriented managed entities represent manageable resources in terms of shared state and behaviour where state and behaviour are separated through outsourcing of the behaviour to assigned so-called "managing entities" (i.e., service-oriented façades) that take steward roles for the state entities (i.e., managed objects proper). The state entities are accessed and controlled (preferably exclusively) through "their" steward behaviour entities. Consequently, the modelling of managed entities will result in two-part SOA design/IDL models (and two-part analysis/UML models if needed) where the two parts, the state model and the behaviour model, can be progressed widely independently since behaviour entities often need only use state entity identifiers instead of instances or could use the IDL type `any` instead of the proper state entity IDL types.

This framework defines the root classes and inheritance principles for both models in CORBA IDL. It defines a root managed object, called `Common_T`, to possess all the desired basic properties of service-oriented managed objects. All SO managed objects shall use the `Common_T` managed object as a template to ensure lightweight inheritance of the common properties. Refer to clause 8.5 for details, and to clause 6.2.2 for alternative, but less lightweight modelling approaches. Similarly, a root service-oriented façade object, called `Common_I`, is defined to possess all the desired basic capabilities of service-oriented façade objects. All SO façade objects shall subclass from the `Common_I` façade object to inherit the common capabilities. Refer to clause 9 for details.

10.3 Naming conventions for managed objects and façade objects

The objective of this clause is to facilitate consistent naming of managed objects and façade objects in the service-oriented NML-EML interface, more generally OS-OS interface. It defines mandatory naming (i.e., identifier) conventions for the state model and the behaviour model.

This framework does not define specific managed object classes but only the struct `Common_T` which groups the mandatory common attributes of each MOC. This is quite often a "virtual" struct, meant to be renamed and extended whenever a specific MOC should be defined. It could also be used for the modelling of managed objects, all of whose attributes are additional info parameters. Such a managed object without MOC-specific attributes (i.e., of IDL type `Common_T`) is called generic. For the purpose of identifying the sources of notifications by their name and a filterable object type, an interface specification that extends this framework has to list all implemented object types (i.e., MOCs) in the enum `notifications::ObjectType_T` (see clause 8.7.1).

The ITU-T IDL sequence `globaldefs::NamingAttributes_T`, a list of `globaldefs::NameAndStringValue_T`, is used to define identifiers (FDNs) for managed entities

that are not instantiated as CORBA objects and thus do not have CORBA object identifiers (IORs). The name approach chosen is uniqueness, concreteness, transparency and structure, and complies with CORBA names. The FDN is unique within the scope of the EMS and is unchanging through the lifecycle of the managed object, i.e., the name cannot be changed during the life of the object (neither from the EMS nor from an NMS). The semantics of an FDN is defined through the containment relationships of its RDN components. This is explained in clause 10.3.2.

This framework does not define specific façade interfaces but only the interface `Common_I` which provides the basic common capabilities of each façade. This is usually a "virtual" interface, meant to be extended whenever a specific façade should be defined. However, instances of `Common_I` could be used for accessing and controlling generic managed objects (see clause 9.3).

The OMG IDL sequence `CosNaming::Name`, a list of `CosNaming::NameComponent`, is used to define CORBA names for entities that are instantiated as CORBA objects, such as service-oriented façade objects, and thus do have an IOR. The CORBA name is unique within the initial naming context (see clause 6.4 of [ITU-T Q.816.2]). The semantics of a CORBA name is defined through the bindings of its name components. This is explained in clause 10.3.3.

10.3.1 General rules

The name structures of service-oriented managed objects and managing objects (i.e., façades) are syntactically equivalent although semantically different. Formally, `NameAndStringValue_T` (i.e., an NVS pair) can be mapped to `NameComponent` (i.e., an id/kind pair) by mapping `name` to `kind` and `value` to `id`, and vice versa (using the appropriate portable CORBA helper function, for example `CORBA::string_dup` in C++). This equivalence allows statement of general rules that apply to both state model (with ITU-T `globaldefs` names) and behaviour model (with OMG `CosNaming` names), and use of common IDL operations for both models.

For example, the OMG `ORB` interface and the OMG naming service provide IDL operations to stringify (or externalize) and destringify (or internalize) IORs, CORBA names and object URL schemes (see clauses 2.4 of [OMG 04-10-03], 11.6.6 of [OMG 98-07-01] or 13.6.6 of [OMG 99-10-07] [OMG 01-02-33] and 13.6.7 of [OMG 01-02-33]). These can be used not only for service-oriented façades but also for the stringification and destringification of managed object names. In particular, the OMG-defined normative syntax of stringified OMG `CosNaming` names implies the equivalent syntax of stringified ITU-T `globaldefs` names.

Such stringified managed object names may be used, e.g., to encode the attribute nativeEMSName of managed objects (see clause 8.5), and of ALM and TCA notifications (see clause A.4.2). As a general rule, wherever deterministic naming (or an indirect name) (see clause 10.3.2) is used for a managed object, an attribute nativeEMSName will be added, which is a string with an EMS- or NMS-determined format (see IDL comments of `common::setNativeEMSName` and `common::setUserLabel`), that describes what the object is legibly called on user interfaces of the EMS (or hosting NE).

Since names carry a semantics which is defined through relationships between name components, both the set of all managed object names under control of the managed system and the set of all CORBA names under control of the local naming service can be depicted as a naming graph where the nodes, or vertices, represent managed objects, with `globaldefs` names or CORBA objects such as façade/managing objects, with `CosNaming` names, and the edges represent containment relationships or bindings. Since service-oriented managed objects are accessed and controlled through façade/managing objects (see clauses 10.2 and 7.2.1.1), there are also relationships between `CosNaming` nodes and `globaldefs` nodes.

As another general rule, it is required in the interest of lightweight CORBA modelling that both naming graphs are as simple as possible. In particular, both graphs should be rooted trees with a

quite low number of levels. This challenge may lead to the normative specification of a relationship structure for the value fields of name components. Refer to clauses 10.3.2 and 10.3.3 for details.

NOTE – The CORBA naming of façade objects and the relationship between managed object names and façade object names as described in clause 7.2.1.1 depends on whether the managed system implements the session service which allows for a minimalistic use of the naming service.

10.3.2 Structure of naming attribute of managed objects

The `globaldefs::NamingAttributes_T` structure is used to define identifiers (names) for managed entities that are not instantiated as first class CORBA objects and thus do not have a CORBA object identifier (IOR). It represents "the hierarchical name structure" of a second class "non-CORBA" object (managed object); it is an attribute of the managed object that contains its full distinguished name (FDN). The structure of the name is hierarchical and reflects the containment relationships between managed objects in a simple way. It consists of an ordered list (sequence) of components (atomic RDNs) where the preceding component is a containing entity for the following component (e.g., a network element contains a termination point).

Each component is a name/value pair. The name field identifies the type of the managed object. The value field identifies the instance of the managed object. The syntax of the name field and value field are strings (i.e., name components are NVS pairs). The strings used for the name field are case sensitive and may include only uppercase and lowercase letters from the basic 26-letter Latin alphabet ([b-ISO/IEC 646]), digits, underscores, hyphens (not dashes) and plus signs. It is recommended to use only letters though. An interface specification that extends this framework must specify meaningful name field strings for all supported object types (e.g., `"EMS"` to represent the managed system (EMS), `"ManagedElement"` to represent a network element). As a rule the IDL name of the struct defining the object type should be taken without the suffix `"_T"`. In the interest of global interoperability, users of this framework must cooperate in reaching normative usage of standardized name field strings. Details of such cooperation are for further study.

The strings used for the value field of the name/value pairs will be at most 1024 characters long (from [b-ISO/IEC 8859]) with the white space character allowed in the value but with no leading or trailing spaces. For instance, a value could be the string `"the string splendid"` but not the string `"the string exquisite"`. All value field strings are case sensitive. Apart from these rules, any value field is a free format string, assigned exclusively by the managed system (EMS) whose structure and admissible values depend on the object type (i.e., the corresponding name field). Different to name field strings, it also depends on the object type whether an interface specification that extends this framework has to specify the value field strings and if users of this framework must cooperate in reaching normative usage of standardized value field strings.

Cases where standardization of value field strings is required would be object types that allow for recursive containments. Two prominent examples are physical network equipment and logical transmission entities related to complex multiplexing hierarchies. For such managed object types, containment shall be represented in a stringified manner inside the value field strings in order to avoid cycles in the naming graph and to limit the number of name components per name. As a consequence, containment strings must be normatively specified for these object types quite similar to the standardization of stringified CORBA names by the OMG naming service (see clause 2.4 of [OMG 04-10-03]) or [ITU Q.816] (as LDAP DN strings – see 6.1.1 of [ITU-T Q.816]). This includes format constructors such as `"/"` (or `","`) and `"="` (or `"."`) to build substrings like `"/"<name>"="<value>` (or `"/"<id>"."<kind>` or `","<kind>"="<id>`). Determination of relevant object types and details of value field string standardization for stringified containment are for further study.

The fine-grained framework uses the naming service not only to define unique names for managed objects but also to store containment relationships. To this end, a naming context is associated with each managed object which stores bindings for all contained objects and a special binding with kind

value `"Object"` and id value `""` for the object itself (see clause 6.1 of [ITU-T Q.816]). As a result, the full distinguished name of managed objects is extended by a new final name component `"/.Object"` (or `",Object="`). The coarse-grained framework keeps this convention for managed objects not accessible through a façade and extends it for managed objects accessible through a façade only (`"/<façade ID>."`) and for managed objects accessible through a façade as well as directly (`"/<façade ID>.Object"`) where `<façade ID>` denotes an implementation-defined identifier for the façade through which the object is accessed that is unique among all façades exposed by the managed system. Managing systems can therefore gain access to manager objects by only using their names: a managed object name may be resolved to provide the IOR of the object, and/or the identifier of the responsible façade (which in turn allows retrieval of the IOR of this façade – see clause 10.3.3). The service-oriented framework adopts this convention on final name components for service-oriented managed objects according to the naming rules for SO façade name components and with the option to omit the final component in case of singleton façades.

The set of containment relationships between managed objects exposed by a managed system can be depicted as a naming graph that shows name components as nodes/vertices and arrows/edges between nodes representing containment. To enforce unique managed object FDNs per managed system, it will be required that the naming graph consists of one or more rooted trees, preferably only one. Each root represents a management domain under control of the managed system (e.g., a telecom network in case of an EMS). In addition, there may optionally be some isolated nodes from a very small number of MOCs whose names would be unique *per se*. These options provide some flexibility for progression of the naming graph beyond a single rooted tree. An example could be an OS that participates in message-oriented OS-OS communication across a service bus as opposed to the quite rigid managing system/managed system (i.e., client/server) communication implied by a synchronous CORBA IDL interface (despite CORBA's message bus capabilities).

Summarizing the foregoing considerations results in guidelines for managed object naming:

– Managed object names are of type `globaldefs::NamingAttributes_T`, and so a name component is an NVS pair consisting of a `name` field `string` and a `value` field `string`.

– Name field strings are case sensitive and may include only uppercase and lowercase letters, digits, underscores, hyphens and plus signs – preferably only letters. `"Common"` identifies generic managed objects of IDL type `common::Common_T`. Users of this framework must cooperate in reaching normative usage of standardized name field strings.

– Value field strings are case sensitive and may have at most 1024 characters with no leading or trailing spaces. They have free format unless structure and admissible values depend on the object type (i.e., the corresponding name field). Users of this framework must cooperate in reaching normative specifications of value field strings for standard name fields, including agreements on object types to have free format values or to encode stringified containment.

– The final name component of any managed object is equivalent to the final CORBA name component of the service-oriented façade that is responsible for access to and control of the object. If this façade steward is a singleton, the final name component can be omitted.

– Lightweight ITU-T `globaldefs` naming graph for service-oriented managed objects: The naming graph for managed objects that are exposed by a managed system (e.g., an EMS) across a service-oriented OS-OS interface shall be a rooted tree with few levels (up to six, for example) whose edges represent managed object containment, optionally together with further such rooted trees and some isolated nodes from a very small number of object types. Some nodes may encode stringified containment with the same agreed string format.

These naming conventions allow standardized determination of the MOC and the responsible SO façade of a managed object from its name, which is an important system performance advantage. That is why the guidelines for managed object naming are also called deterministic naming.

NOTE – In case of a singleton façade, where the final name component can be omitted, it is understood that the managing system knows from the OS-OS interface specification which managed object classes the façade is actually managing. This includes the possiblity that façades are managing more than one MOC, for example a particular MOC and certain (repeatedly) contained MOCs (e.g., network elements and TPs).

These guidelines also allow for indirect naming of managed objects that are not exposed across the OS-OS interface. Such naming may concern object types which are part of a rooted naming tree with "stand-alone" object instances that do not have real ancestors and so the value fields of their ancestors carry empty strings; these objects are not under control of the managed system exposing their names (i.e., they are "off-network") but the names serve as references "to the other side". Or such indirect naming may concern object types which are not modelled at the OS-OS interface but their identifiers are exposed as names (e.g., to identify alarm sources – see clause 8.7.1).

Users of this framework must also cooperate in reaching normative definition of standardized behaviour of the managed object naming graph through suitable façade operations. This includes a getAll`<object type>`Names operation for each object type (see clause 9.3.2.1) and inspection of the naming graph through classified retrieval of names of contained objects in case of stringified containment. Managing systems will usually not be allowed to modify the naming graph.

Such cooperation could result in a formal specification of the naming graph for service-oriented managed objects. While its normative documentation is a minimum requirement, specification of the managed object naming graph in CORBA IDL would add considerable value to the service-oriented framework. The fine-grained framework recommends the specification of name binding modules for use in create operations and naming contexts of CORBA managed objects, and the coarse-grained framework adopts this usage and extends it to the containment service for all three species of managed objects (only with façade access, with façade access and IOR, only with IOR). Such a name binding module consists of constants and includes at least the scoped names of a containing/superior object type and a contained/subordinate object type, and the unscoped name of the same contained object type to be used as the `kind` field value of the name component of this object type. The following examples of name binding modules stem from ITU-T Rec. M.3120:

```
module NameBinding
{
...
   module EquipmentHolder_EquipmentHolder
   {
      const string superiorClass = "itut_m3120::EquipmentHolder";
...
      const string subordinateClass = "itut_m3120::EquipmentHolder";
...
      const string kind = "EquipmentHolder";
   }; // module EquipmentHolder_EquipmentHolder
...
   module EquipmentHolder_ManagedElement
   {
      const string superiorClass = "itut_m3120::ManagedElement";
...
      const string subordinateClass = "itut_m3120::EquipmentHolder";
...
      const string kind = "EquipmentHolder";
   }; // module EquipmentHolder_ManagedElement
...
}; // module NameBinding
```

While a pure fine-grained implementation uses only the naming service and a sheer coarse-grained implementation uses only the containment service for containment relationship management, when a mixed fine- and coarse-grained implementation is chosen, the managed system will need to handle synchronization issues for the two services since the three species of managed objects give rise to

nine species of containment (see clause 8.1.5 of [ITU-T Q.816.1]). Such handling includes naming contexts for the "only façade" species that do not have a binding with kind value `"Object"` (nor one with kind value `""`). In mixed implementations, changing the species of a managed object implies changing its name which might require re-creation, and so change of species should occur infrequently.

Modification of the naming graphs stored in the naming service and the containment service will usually occur when managed objects are created or deleted. Create operations for fine-grained or coarse-grained managed objects therefore include an input parameter for a scoped name binding module name, an input parameter for the name of the containing object, an input parameter for the requested `id` field value of the name component of the object to be created, and an output parameter for the name of the object to be created. Refer to clause 6.9.1 of [ITU-T X.780] for details.

A naming context implements a table that binds name components to IORs (of application objects or other naming contexts) and the naming context of a managed object stores the name components of its contained objects. The containment service adapts these principles through replacement of IORs by internal references, thereby implementing the naming graph for all managed object species. So for all fine- or coarse-grained implementations, the naming graphs are determined by bindings of name components to references (of other managed objects or naming contexts). The containment service improves the inspection capability of the naming service to list all bindings of a naming context through a scoped get contained names operation (similar to the multiple object operation (MOO) service) where the scope covers either just the named base object (of level 0), or the entire subtree of objects below and including the base object, or the objects at a certain level below the base object, or all of the objects down to a level including the base object and the level. The containment service also allows to restrict the retrieval of contained names to object types of a specified `kind`.

Cooperation of all users of the service-oriented framework for the purpose of globally interoperable progression should consider lightweight adaptation of the approach of the fine-grained and coarse-grained frameworks to containment relationships encoding, taking ITU-T `globaldefs` names instead of OMG `CosNaming` names though, and defining simplified name binding modules. Name binding then means binding of name components of type `NameAndStringValue_T` to internal references, thereby defining the directed edges of the managed object naming graph. The naming graph will be stored neither in a naming service nor in a containment service but in the SO façades that are responsible for one or more MOCs using an element of federation. These façades also provide appropriate create and delete operations for MOCs that may be deleted by managing systems and operations for naming graph inspection while operations for naming graph modifications remain private to be used by the managed system only.

The work of the service-oriented CORBA TMN framework user group for the specification of the service-oriented managed object naming graph in CORBA IDL shall use the managed object naming convention of MTNM v3.0 [TMF objectNaming] as a starting point.

These considerations result in guidelines for IDL specification of managed object naming:

– For each object type, let `<object type>` denote the IDL name of the `struct` defining the object type (by lightweight inheritance from `common::Common_T`) without the trailing `"_T"`, and let `<type scope>` denote the IDL scope of this `struct` (a list of module names separated by `"::"` representing nested IDL modules where the last one contains the `struct`).

– For each pair of object types `<object type-1>` and `<object type-2>` with scopes `<type scope-1>` and `<type scope-2>` having a containment relationship, define the constants `const string containingType = <type scope-1><object type-1>; const string containedType = <type scope-2><object type-2>; const string name = <object type-2>;` and group them into an IDL module with name

`<object type-2>"_"<object type-1>`. Add optional constants if required (e.g., whether managing systems may create objects of type `<object type-2>`, which delete policy applies for objects of type `<object type-2>`). Group all these containment relationship modules into a global IDL module with name `NameBinding`.

– For the definition of specific object types and their containment relationships, take the managed object naming convention of MTNM v3.0 [TMF objectNaming] as a starting point.
– Develop object type-specific create operations and object type-specific semantics of delete operations that consider the name binding information collected in `NameBinding`.
– Develop object type-specific operations for scoped inspection of the naming graph.

It is believed that the guidelines for service-oriented CORBA managed object naming and IDL specification of the managed object naming graph can well assist in the development of normative naming conventions for subsequent progression of the framework.

10.3.3 Structure of CORBA name of façade objects

The `CosNaming::Name` structure is used, as usual, to define names for managed entitities and support entities that are instantiated as first class CORBA objects and thus do have a CORBA object identifier (IOR). Each component is an id/kind pair. The kind field identifies the type of the CORBA entity. The id field identifies the instance of the entity. The syntax of the kind field and id field are strings (i.e., name components are NVS pairs). The strings used for the kind field are case sensitive and may include only uppercase and lowercase letters from the basic 26-letter Latin alphabet [b-ISO/IEC 646] and underscores. The strings used for the value field of the kind/id pairs will be at most 1024 characters long (from [b-ISO/IEC 8859]) with the white space character not allowed. All id field strings are case sensitive. Apart from these rules, any id field is a free format string, assigned exclusively by the managed system (EMS), whose structure and admissible values depend on the entity type (i.e., the corresponding kind field). The name components of CORBA objects are stored by a naming service as nodes of the naming graph with directed edges defined through bindings. Refer to clause 6.4 of [ITU-T Q.816.2] for further details. These principles apply to the fine-grained, coarse-grained and service-oriented frameworks.

For the coarse-grained framework, clause 8.1.2 of [ITU-T Q.816.1] introduces façade identifiers which enable the retrieval of any X.780.1 façade based on its `"/Façade."<façade ID>` name component since this component is bound in a root naming context (e.g., the initial naming context). These façade identifiers are not further specified but are considered implementation-defined. For the service-oriented framework, clause 6.4.3 of [ITU-T Q.816.2] provides an overview of the CORBA object name components, Figure 7 below provides the details, Table 5 specifies the directed edges of the naming graph, and Figure 1 of [ITU-T Q.816.2] depicts an example of a naming graph. The naming graph is in fact a rooted tree with the initial naming context as the root and with other component kinds besides SO façades to support OS-OS interface classes, IDL versioning, session services, event channels, multi-vendor capability and vendor-specific interfaces. Thus the SO façade object naming covers only part of the entire rooted tree. For each OS that registers interface instances in the naming tree, the OS name, as specified by the OS vendor, and the OS vendor company name are used in a normative way as high-level classification aids.

Summarizing the foregoing considerations results in guidelines for façade object naming:

– Façade object names are of type `omg.org::CosNaming::Name`, and so a name component is an NVS pair consisting of a `kind` field `string` and an `id` field `string`.
– Kind field strings are case sensitive and may include only uppercase and lowercase letters and underscores. The admissible values shall be those listed in Figure 7.

– Id field strings are case sensitive and may have at most 1024 characters with the white space character not allowed. They have free format unless structure and admissible values depend on the corresponding kind field as specified normatively in Figure 7.

– Lightweight OMG `CosNaming` naming tree for service-oriented façade objects: The naming graph for service-oriented façade objects that are registered by a managed system (e.g., an EMS) with its local naming service shall be (part of) a rooted tree with few levels whose edges represent bindings and whose root is the initial naming context. The admissible nodes and bindings shall be those specified in Figure 7 and Table 5.

NOTE – The structure of the façade object naming graph, and the entire CORBA naming graph, depends on whether some or all of the managed systems that use the naming service implement the session service (see clause 9.1 of [ITU-T Q.816.2]) since, in the presence of the session service, SO façade registration is not needed from an NMS viewpoint (but may make sense from an EMS viewpoint when numerous façade IORs have to be controlled).

Standardized behaviour of the service-oriented façade object naming graph is defined through operations of the `NamingContext` interface of the (lightweight) naming service. This includes operations to construct and modify the naming graph segment originating from a given context, the `resolve` operation to retrieve the IOR bound to a CORBA name in the given context, and the `list` operation to iterate through the bindings of the given context by using a binding iterator. Managing systems will usually not be allowed to modify the naming graph.

It is believed that the guidelines for service-oriented CORBA façade object naming can well assist in the specification of normative naming conventions for subsequent progression of the framework.

10.4 Service-oriented modelling of transmission technologies

Clauses 8.4 and A.3 provide the generic foundation for the multi-technology capability of the service-oriented CORBA framework but do not at all indicate how to represent the transmission technologies of a concrete telecom environment. This clause provides a number of related concrete guidelines and general considerations for sustainable multi-technology interface progression.

10.4.1 Key aspects of ITU-T Recommendations G.805 and G.852.2

A transport network conveys telecommunications information between locations. In concrete environments this information is a signal with a technology-dependent format and, optionally, a specific transmission rate (bandwidth), which is transmitted or received on network connections or in a connectionless way. [ITU-T G.805] considers each specific signal to be bound to a connection-oriented transmission layer and introduces the term characteristic information (CI) for the signal. It decomposes the transport network into a number of independent single-layer networks with a client/server association between hierachically adjacent layer networks. Each layer network can be separately partitioned in a way that reflects its internal structure or the manner that it will be managed. Thus the concepts of partitioning and layering are orthogonal: a downwards vertical transmission layering is supplemented by a recursive horizontal layer network partitioning. A single layer network describes the generation, transport and termination of a particular CI.

Layer networks can be partitioned into appropriate subnetworks and links between them, and subnetworks and links may be further partitioned. Links represent transmission resources provided to "this" layer by its server layer networks. A network partitioning implies connection partitioning of its network connections (NCs) into tandem connections, which are sequences of contiguous subnetwork connections (SNCs) within subnetworks and link connections (LCs) between NEs or subnetworks. In the client/server relationship of adjacent layer networks, client layer LCs are supported by server layer trails (one-to-one, many-to-one, one-to-many) through an adaptation that modifies the CI of the client layer so that it can be transported over the trail (= NC including trail terminations at both extremities) in the server layer network. A signal that is transmitted or received on a trail is called adapted information (AI). The trail is built from the underlying NC by means of a

trail termination source/sink that generates CI/AI by adding/extracting a trail overhead to/from the information it transmits/receives. The trail overhead mitigates the possible loss of AI, allows reliable monitoring of the trail and ensures the integrity of information transfer.

A one-to-one client/server relationship represents the case of a single client layer LC supported by a single server layer trail. The many-to-one relationship represents the case of concurrent LCs of client layer networks supported by one server layer trail at the same time. Multiplexing techniques are used to combine the client layer signals. The one-to-many relationship represents the case of a client layer LC supported by several server layer trails in parallel. Inverse multiplexing techniques are used to distribute the client layer signal across the supporting server layer trails.

Reference points (RPs) are formed by bindings between inputs and outputs of adaptations and/or trail terminations and/or transport entities. Bindings represent static connectivity that cannot be directly modified by management action. Each transport entity is delimited by accompanying RPs: LCs and SNCs by connection points (CPs) or termination connection points (TCPs), NCs by TCPs, and trails by access points (APs). Figure 3 of [ITU-T G.805] provides an overview of the different pipe and reference point concepts used in connection-oriented layer network functional models.

[ITU-T G.805] classifies transport network layers into service layers, path layers and physical layers. Physical layers belong to some OSI layer 1 technology (see [ITU-T X.200]) and are media-independent transmission path (TPath) layers or media-dependent transmission media (TMedia) layers. TMedia layers are section layers (e.g., multiplex sections, regenerator sections, digital sections, optical channels) or physical media layers (e.g., wired cables, optical fibres, radio frequency channels).

[ITU-T G.805] describes the functional and structural architecture of connection-oriented networks in a generic way, i.e., independently of the underlying networking technologies. Its power is revealed when applying it to the technologies of a concrete telecom environment. The application consists of the identification of the individual layers, their characteristic and adapted information, and their adaptation and trail termination functions. Major traditional examples are PDH layers, SDH layers, SONET layers, OTN layers and ATM layers. While PDH, SDH, SONET and OTN are connection-oriented (CO) and circuit-switched (CS) infrastructures, ATM is a CO and packet-switched (PS) network mode. Table 3 provides an overview of the transmission layers of these technologies.

[ITU-T G.852.2] specifies a set of definitions of management abstractions of G.805 transport network architectural components which are termed transport network enterprise resources. Figure 2 of [ITU-T G.852.2] provides an overview of the corresponding concepts used in connection-oriented TMN functional models. The most important resources are the connection termination point (CTP) and the trail termination point (TTP). The CTP is the assembly of one half of the CP or TCP, and the client-layer part of the accompanying adaptation. The TTP is the assembly consisting of the trail-termination part side of the TCP and the accompanying trail termination together with the server-layer part of the adaptation and hence together with the AP. Figure 2 of [ITU-T G.852.2] shows that, from the enterprise viewpoint, CTPs are the delimiters of link connections and TTPs are the delimiters of trails (as well as corresponding network connections).

Table 3 – Major traditional transmission technologies

Technology (Network mode) Pertinent standard(s)	**Layer type**	**Layers**
Digital	Prime	DS0
PDH (CO-CS) G.702/3/4, G.705	E-Carrier	E1, E2, E3, E4, E5
	T-Carrier	T1, T2, T3
SDH (CO-CS) G.707/Y.1322, G.803, G.774-series, G.783	LOVC TPath	VC-11, VC-11-Xv, VC-12, VC-12-Xv, VC-2, VC-2-Xc, VC-2-Xv, VC-3, VC-3-Xv (X = 1..256; especially 4, 16, 64)
	HOVC TPath	VC-3, VC-3-Xv, VC-4, VC-4-Xc, VC-4-Xv (X = 1..256)
	Multiplex section	STM-n (n = 0, 1, 4, 8, 16, 64, 256)
	Regenerator section	STM-n (n = 0, 1, 4, 8, 16, 64, 256)
SONET (CO-CS) T1.105, GR-253	VT TPath	VT-1.5, VT-1.5-Xv, VT-2, VT-2-Xv, VT-3, VT-6, VT-6-Xc, VT-6-Xv (X = 1..256; especially 3, 12, 48, 192)
	STS TPath	STS-SPE, STS-Xc, STS-Xv, STS-3c-SPE, STS-3c-Xc, STS-3c-Xv (X = 1..256)
	SONET line	STS-n, OC-n (n = 1, 3, 12, 24, 48, 192, 768)
	SONET section	STS-n, OC-n (n = 1, 3, 12, 24, 48, 192, 768)
OTN (CO-CS) G.709/Y.1331, G.798, G.872	Digital path	OPUk, OPUk-Xv, ODUk (k = 1, 2, 3)
	Digital section	OTUk, OTUkV (k = 1, 2, 3)
	Optical path	OCh, OChr
	Optical section	OMSn, OTSn, OPSn ($n \geq 0$)
ATM (CO-PS) I.150, I.326, I.731/2	Path	VC, VP
	TPath	UNI, NNI; layering depends on underlying transport technology

NOTE 1 – Table 3 is not exhaustive. Important missing traditional transmission technologies are, for example, frame relay, legacy voice services and DSL. Whilst ATM offers two multiplexing levels for permanent virtual connections/circuits (PVCs) and ATM cells are identified through their 8-bit VPI and 16-bit VCI, frame relay (FR) offers just one PVC multiplexing level and FR frames are identified through their 10-bit data link connection identifier (DLCI). Switched voice services (POTS, ISDN-BA) are transmitted within DS0 channels that carry signalling information or bearer information and are obtained by channelizing an E1/T1 link into 32/24 voice channels which are configured to be bearer channels or common signalling channels; a single E1 link or 16 E1 links or 24 T1 links are then grouped into what is called a signalling interface (V5.1/G.964 or V5.2/G.965 or GR-303). Leased line services are transmitted within n*DS0 or 1.544 Mbit/s (T1) or 2.048 Mbit/s (E1) channels or even T3/E3 channels. Data services such as FR may be captured at DSR ports (e.g., X.21, V.35) with a configurable bandwidth between 56 kbit/s and 2.048 Mbit/s, for example, or may also start at the end of n*DS0 or T1/E1 channels. The DSL physical layer originally consisted of a physical media dependent (PMD) sublayer and a transmission convergence (TC) sublayer, but later the TC sublayer was renamed to transport protocol specific TC (TPS-TC) sublayer, and the physical media specific TC (PMS-TC) sublayer was split off the PMD sublayer (see G.99x-series of ITU-T Recommendations).

A connection according to [ITU-T G.805] is an architectural component which consists of an associated pair of unidirectional connections capable of simultaneously transferring information transparently within a single-layer network in opposite directions between their respective inputs and outputs. There are SNCs, LCs and NCs. A general (tandem) connection is delimited by CPs or

TCPs or both. A unidirectional NC is delimited by unidirectional TCPs where a unidirectional TCP consists of one of the following bindings: output of a trail termination source to the input of a unidirectional connection; output of a unidirectional connection to the input of a trail termination sink. A TCP consists of a contra-directional pair of co-located unidirectional TCPs and therefore represents the binding of a trail termination to a bidirectional NC. Any TCP has an associated AP (and vice versa), and any NC has an associated trail (and vice versa). Any connection and any trail is characterized by the association/correspondence of its delimiting reference points which has a clearly distinguishable lifetime between connection/trail establishment, multiple data transfer over the connection/trail, and connection/trail release. Consequently, connections and trails as defined in [ITU-T G.805] enable connection-mode transmission and services as defined in [ITU-T X.200].

A single-layer subnetwork (SN) corresponds in the CO layer network functional view to a set of potential or actual CPs and TCPs (or to a set of potential or actual TTPs and CTPs in the CO TMN functional view) that is associated with a (network or element) management function (which may be dependent on a control plane function) in order to facilitate the flexible establishment and release of SNCs between members of this set of reference points. Similarly, a single-layer link corresponds in the CO layer network functional view to two corresponding sets of potential or actual either CPs or TCPs (or to two corresponding sets of potential or actual CTPs in the CO TMN functional view) that is associated with a management function in order to facilitate the flexible establishment and release of LCs between two reference points from different sets.

An edge point of a subnetwork/link is considered here to be initially in the potential state since it may not be created until an SNC/LC is created (with or without activation) having it as an end point. ([ITU-T G.805] therefore uses the term port for the part of the CP or TCP which may exist without SNC/LC. It defines the single-layer subnetwork as a set of ports which are available for the purpose of transferring CI, and the link as a relationship between two specified sets of ports with the purpose of transferring CI according to a certain available transport capacity whose definition is technology-dependent.) An SNC/LC is delimited by actual end points out of the set of potential edge points of its containing subnetwork/link. Whilst potential CPs and TCPs are identified for transferring CI and providing connectivity or transport capacity, actual CPs and TCPs are available for and capable of performing these tasks. Potential reference points either exist and are connectable but not connected, or are not instantiated but identified. Any SNC or LC creation/deletion changes the "connectivity state" of the involved edge points from potential/actual to actual/potential.

NOTE 2 – The G.805-based concepts of potential versus actual connectivity and transmission capacity may serve as an explanation of the semantics of the "SN ← SNC" and "link ← LC" containment. They are not explicitly mentioned in [ITU-T G.805] but are introduced here for clarification reasons.

NOTE 3 – The definitions of TMN resources in [ITU-T G.852.2] have corresponding definitions of MOCs in [ITU-T M.3100]. The virtual termination point (TP) (`terminationPoint`) is unidirectional and single-layered, and its attribute `characteristicInformation` identifies the transmission layer. From TP directly derived are `connectionTerminationPointSource`, `connectionTerminationPointSink`, `trailTerminationPointSource`, `trailTerminationPointSink` and `networkTerminationPoint` (from which `networkCTPSource/Sink`, `networkTTPSource/Sink` and `genericTransportTTP` are derived). Moreover, [ITU-T M.3100] defines a group TP (`gtp`) and a TP pool (`tpPool`) managed object class which have no counterparts in [ITU-T G.852.2]. Their use in the context of service-oriented modelling of transmission technologies, in particular for lightweight decomposition and aggregation, is for further study, and will likely be technology-specific. For example, TP pools consisting of CTPs seem pretty suitable to represent link ends (for all transport technologies) (see clause 6.2.8 of [ITU-T G.852.2]).

The potential versus actual reference point concepts can be further illuminated through descriptions of subnetwork and link decomposition with derived connection decomposition. There are six cases, three cases of partitioning and three cases of fragmentation:

– **Subnetwork partitioning** (or serial/sequential subnetwork decomposition) (based on 5.3.2.1 of [ITU-T G.805], 6.2.11 of [ITU-T G.852.2] and 7.6 of [ITU-T Y.1314])

A subnetwork may be partitioned into a set of "smaller" subnetworks interconnected by links provided its potential connectivity is not changed. This is a composite SN made of component subnetworks (cSNs). The process is recursive until a limit is reached – a smallest/indivisible SN (matrix/fabric within an NE), for example. A subnetwork may also be recursively aggregated from nested sets of "smaller" SNs. The set of (potential or actual) edge points of the partitioned SN is a proper subset of the set of all edge points of all SNs/cSNs at the next level. For each subnetwork its set of edge points remains unchanged under partitioning while edge points are revealed at lower levels which are internal to it.

This form of decomposition generates new reference points. For example, a layer network may have an international portion and national portions where each national portion has regional/access portions with attached sets of potential TCPs (see Figure 6 of [ITU-T G.805] and Figure 7-14 of [ITU-T Y.1314]).

– **Link partitioning** (or serial/sequential link decomposition) (based on clause 5.3.2.2 of [ITU-T G.805] and 6.2.6 of [ITU-T G.852.2])

A link may be partitioned into a serial arrangement of "shorter" links. This is a compound link made of component links (cLinks) that are contiguous (link chain or serial-compound link). Between two component links may or may not be a subnetwork. The cLinks may themselves be further serially partitioned. A link may also be recursively aggregated in a serial fashion from "shorter" links. The capacity of a serial-compound link is less or equal to the largest capacity ("diameter") of its cLinks.

This form of decomposition generates new reference points.

– **Link fragmentation** (or parallel link decomposition) (based on clauses 6.2.6, 6.2.14 and 6.2.8 of [ITU-T G.852.2], 5.3.2.2 of [ITU-T G.805] and 6.3.2.5.3 of [ITU-T G.8010])

A link may be recursively aggregated by bundling a set of links that are equivalent for the purpose of routing. Conversely a link may be fragmented into a parallel arrangement of "thinner" links "of equal length". This is a compound link made of fibre links (fLinks) (called component links in [ITU-T G.852.2] though) that are bundled (link bundle or parallel-compound link). The fLinks may themselves be further partitioned in parallel. The capacity of a bundled compound link is the sum of the capacities ("diameters") of its fibre links. This form of decomposition does not generate new reference points since the fragmentation of a link corresponds to simultaneous partitionings of its (two) sets of (potential or actual) edge points.

– **Subnetwork fragmentation** (or parallel subnetwork decomposition) (adapted from clauses 6.3.2.5.1, 6.3.2.5.2 and Annex A of [ITU-T G.8010] and 7.6 of [ITU-T Y.1314])

A subnetwork may be fragmented into a set of "smaller" subnetworks without interconnecting links by decomposing its set of (potential or actual) edge points without adding any interior reference point. This is a composite SN made of subnetwork fragments (SNFrs). The set of (potential or actual) edge points of the fragmented SN is equal to the set of all edge points of all SNs/SNFrs at the next level.

An edge point of a fragmented SN is considered to belong also to one (and only one) SNFr (since otherwise either the SN would disappear or SNFrs would have no edge points). This process is not recursive. Fragmentation of a subnetwork usually lowers its potential connectivity since no (potential) connectivity is provided between the fragments. This form of decomposition does not generate new reference points. Subnetwork fragmentation may be accompanied by link fragmentation.

– **Connection partitioning** (derived from subnetwork and link partitioning) (also based on clauses 5.3.2.1 of [ITU-T G.805], 6.2.12, 6.2.13 and 6.2.16 of [ITU-T G.852.2])

Subnetwork and link partitioning are related to edge points, in potential or actual connectivity state, and so have a particular impact on the actual edge points. This impact

results in synchronized connection partitioning. The horizontal partitioning of a layer network into SNs and links between them (topology decomposition) implies a partitioning of its NCs and trails into synchronous tandem connections (connectivity decomposition). A synchronous tandem connection is a series of contiguous LCs and/or SNCs (not necessarily alternating) where each LC or SNC is contained by a link or SN of the topology decomposition. It is delimited by actual TCPs or CPs. In the case where there are two TCPs, or two CPs or one TCP and one CP, the corresponding NCs are called, respectively, closed/terminated, or open/unterminated, or half-open. A closed NC consists of synchronously nested open NCs (see Figure 7 of [ITU-T G.805]). An LC or an LC end point corresponds to a "thinnest" fLink or link end (i.e., TP pool) (and vice versa) while an SNC, or the set of end points of an SNC, corresponds to a "smallest" SNFr or SN edge respectively, (and vice versa). This implies a correspondence of the connectivity decomposition with the largest part of the topology decomposition all of whose edge points are actual, and may serve as an explanation of the semantics of the "SN ← SNC" and "link ← LC" containments. Connection partitioning does not generate new reference points. Neither does connection creation generate new reference points – but it changes the connectivity states of the involved SN and link edge points from potential to actual.

– **Connection fragmentation** (derived from subnetwork and link fragmentation)

Subnetwork and link fragmentation are related to edge points, in potential or actual connectivity state, and so have a particular impact on the actual edge points. This impact results in synchronized connection fragmentation. The horizontal fragmentation of the topology decomposition (i.e., subnetwork and link partitioning) of a layer network into SNFrs and fLinks between top-level SNs is called VPN topology decomposition (i.e., subnetwork and link fragmentation to define a network-based virtual private network (VPN) – see [ITU-T Y.1311]). It implies a fragmentation of NCs and trails into synchronous tandem virtual connections (VPN connectivity decomposition). A synchronous tandem virtual connection (VC) is defined as a series of contiguous LCs and/or SNCs (not necessarily alternating) where each LC and SNC is contained by, respectively, an fLink or SNFr of the VPN topology decomposition.

It is delimited by actual TCPs or CPs and has the same structure as a tandem connection. Connection fragmentation does not generate new reference points. Neither does VC creation generate new reference points – but it changes the connectivity states of SNFr and fLink edge points from potential to actual.

NOTE 4 – Since topology management usually happens before and not together with connectivity/connection management, reference points are commonly potential during topology decomposition processes. However, there may be a number of fixed connections whose end points are always in "connectivity state" actual.

NOTE 5 – It can be useful to distinguish transport entities from traffic entities where transport entities are (logical) network or network element resources used for the transmission of traffic entities. When modelling a connection-oriented layer network, connections are considered as transport entities while the CI and AI as well as trail overheads are considered as traffic entities. Moreover, connections are contained by topological components (SNCs by SNs, LCs by links, NCs and trails by the layer network). In real life, for example, a pipe is a transport entity that can transmit (or receive) water, oil, gas, etc., which are traffic entities, and pipes are laid in pipelines or ducts which are topological components.

10.4.2 Key aspects of ITU-T Recommendations G.809 and G.8010/Y.1306

[ITU-T G.809] and [ITU-T G.8010] lay the foundation of a generic common framework for the functional modelling of connection-oriented (CO) and connectionless (CL) layer networks. In its embryonic form, this framework is based on the generalization of the concept of connection to the novel connectivity concepts of flow and fragment, and derived correspondences of reference points and other basic architectural artefacts of the CO and CL network modes of operation. In the

following clauses, these initial stages of the framework are summarized and also enhanced in a convenient manner.

The technology-dependent CI of a CO or CL transmission layer is made of traffic units where a traffic unit is an instance of the CI and a unit of usage with either a circuit nature or a packet nature (based on timeslots, wavelengths, virtual containers, physical links, cells, frames, datagrams, etc.). A flow, according to [ITU-T G.809] is an aggregation of one or more traffic units with an element of common routing. Flows are used to transfer the CI of the transmission layer between reference points. A flow is always unidirectional so that two co-located flows in opposite directions are needed to represent a "bidirectional flow". [ITU-T G.809] defines the adaptation source/sink functions between a CO or CL client layer and a CO or CL server layer, the flow termination source/sink functions for a CL client layer, the CL trail, the CL link (called FPP link though) and four types of reference points: flow point (FP), termination flow point (TFP), access point (AP) and flow point pool (FPP) (as a group of co-located either FPs or TFPs that have a common routing). Flows are delimited by FPs or TFPs, CL trails are delimited by APs, and CL links are delimited by FPPs.

Flows and FPs/TFPs can be defined, decomposed and assembled in a number of ways:

– A flow can contain another flow. This is recursive until, for example, the limit of a single traffic unit is reached (sparsest or atomic kind of flow). Consequently, flows and CL trails as defined in [ITU-T G.809] enable connectionless-mode transmission and services as defined in [ITU-T X.200].

– The aggregation of traffic units to a flow may be spatial or temporal.

– A flow can be defined/identified in terms of one or more parameters such as its CI, the address to which traffic units are directed, or the address from which traffic units have come.

  NOTE 1 – Such n-tuple identifiers <ID1, ID2, …, IDn> define different forms (or equivalence classes) of flows. For example, when the CI is ETH_CI as defined in [ITU-T G.8010], possible ID components are destination MAC address, source MAC address, VLAN ID, priority.

– A flow can be contained by a topological entity: a flow domain (see below), CL link, and layer network can contain FD flows (FDFs) or link flows (LFs) or network flows (NFs), respectively.

– Flows can be multiplexed or inversely multiplexed together, and (inversely) demultiplexed apart, as part of adaptation to or from a server layer network (client/server multiplexing).

– Flows can be multiplexed and demultiplexed in the same layer network (peer multiplexing).

  NOTE 2 – This capability of flows is called "flow-in-flow" nature of flows. It forms the basis for the peer level VPN topology and connectivity decompositions of CL layer networks (see below). While client/server multiplexing results in new CI, peer multiplexing preserves the CI of the decomposed CL layer but "filters" it. Such "flow-in-flow" configurations are impossible for a CO layer.

– An FP/TFP can always be considered as a member of a flow point pool. The FPP is used to describe the architecture of a layer network when aggregate flows are of more interest than individual flows. The FP/TFP is used when individual flows are of interest.

– Flow points and termination flow points can be partitioned to generate new flow points (which may or may not be grouped into a flow point pool).

  NOTE 3 – This process may be described as changing the "connectivity state" of the initial FP from actual to potential, and the states of the resulting co-FPs from potential to actual (see below).

– Flow points (of a flow point pool) can be aggregated to generate a new flow point.

  NOTE 4 – This process may be described as changing the states of the initial co-FPs (and the FPP) from actual to potential and the state of the resulting FP from potential to actual (see below).

Figure 2 of [ITU-T G.809] corresponds to Figure 3 of [ITU-T G.805] and provides an overview of the different flow and reference point concepts used in connectionless layer network functional models.

[ITU-T G.852.2] refers to [ITU-T G.805] and so applies directly only to CO client and server layers. But simply by extending the used G.805 modelling components with the G.809 modelling components, CL layers can be thought of as being considered by [ITU-T G.852.2] as well. It would be convenient, however, to have an amendment or revision of [ITU-T G.852.2] that formalizes these considerations. Thus, the G.852.2 CTP can easily be generalized to encompass an FP or a client TFP, and the G.852.2 TTP may encompass a flow termination and a server TFP. The CTP should be renamed to "connectivity termination point" to cover CO and CL layers. Then client-layer LCs and LFs are both delimited by CTPs and supported by server-layer trails or CL trails, both of which are delimited by TTPs, and [ITU-T G.852.2] will provide the generic CO and CL TMN functional model.

NOTE 5 – When amending or revising [ITU-T G.852.2] to incorporate CL layers according to [ITU-T G.809], the parts of [ITU-T M.3100] that depend on [ITU-T G.852.2] should also be revised and combined in an amendment.

The G.805 concept of single-layer SN corresponds to the G.809 concept of single-layer flow domain (FD) which is a set of potential or actual FPs and TFPs in the CL layer network functional view (or a set of potential or actual TTPs and CTPs in the CL TMN functional view) that is associated with a (network or element) management function (which may be dependent on a control plane function) in order to facilitate the flexible establishment and release of "coarse enough" (see below) FDFs between members of this set of reference points. Similarly, a single-layer CL link corresponds in the CL layer network functional view to two corresponding sets of potential or actual either FPs or TFPs (or to two corresponding sets of potential or actual CTPs in the CL TMN functional view) that is associated with a management function in order to facilitate the flexible establishment and release of "coarse enough" LFs between two reference points from different sets. The two sets of edge points of a CL link are in fact flow point pools (FPPs).

A flow point or TFP is considered to be in the potential state if (and only if) it is not instantiated but is identified for forwarding traffic units (e.g., by a destination MAC address "label"). The direct or indirect creation/deletion of a flow point or TFP changes its "connectivity state" from potential/actual to actual/potential. This means that any instantiated FP or TFP is considered to always be in the actual state: it is always available for and capable of forwarding identified traffic units.

FDFs across an FD, or LFs along a CL link are enabled through actual FPs at the edge of the FD, or the CL link, that now become flow end points. Such an FDF or LF may be as little as a single traffic unit of the layer's CI or may be aggregated over time from all traffic units that meet an FP-specific filter criterion (e.g., a destination MAC address, a VLAN ID) at the ingress FPs of the FDF or LF. Accordingly, FDFs and LFs may be "thin" or "thick" (fine-grained or coarse-grained), and they will, in particular, usually flow non-continuously when aggregated temporally.

The "coarse enough" FDFs ("coarse enough" LFs) that can be addressed by the management function of an FD (or CL link) are defined via flow domain fragments (or CL fibre links). A flow domain fragment (FDFr) is a distinguished subset of the set of actual edge points of a flow domain. A CL fibre link (fLink) consists of corresponding subsets of the two sets of actual edge points of a CL link. Any FDF or LF contained by an FDFr or CL fLink (i.e., having edge points of the FDFr or CL fLink as its end points) is termed "coarse enough".

For the purpose of forwarding traffic units, one may want to provision such "coarse enough" FDFs or LFs by means of the management (or control) function associated with the containing FD or CL link. However, since FDFrs and CL fLinks consist of actual reference points, such provisioning would be equivalent to the provisioning of the containing FDFrs or CL fLinks. The "coarse enough" FDF or LF can be considered as the aggregation of all flows which are not discarded at the ingress

edge ports of the containing FDFr or CL fLink – it will flow across or along the provisioned FDFr or CL fLink independently of management (or control) actions.

These considerations imply that CL connectivity management is concerned with flow domain fragment and CL fibre link provisioning and not with flow provisioning. It follows from the definitions of flow domain fragment and CL fibre link that much of their management may be flow point pool provisioning. The details of FDFr and CL fLink provisioning are for further study.

NOTE 6 – There may well be requirements for flow configuration but not for the purpose of connectivity management. Flows can be configured, for example, with regard to QoS properties of the CL layer under consideration (e.g., Ethernet user priority) or with regard to filter conditions that relate to the client layer (e.g., certain IPv4 or IPv6 addresses of IP packets within Ethernet MAC SDUs). This amounts to tunnelling of FDFs or LFs through an FDFr or an CL fLink without having any impact on connectivity. The abilities to support such requirements are dependent on capabilities of the ingress FPs of FDFrs and CL fLinks.

The potential versus actual FP and TFP concepts can be further illuminated through descriptions of flow domain and CL link decomposition with derived flow decomposition. There are five cases, based on the concepts of partitioning and fragmentation:

– **Flow domain partitioning** (or serial/sequential flow domain decomposition) (based on clause 6.4.2.1 of [ITU-T G.809] and subnetwork partitioning)

  A flow domain may be partitioned into a set of "smaller" flow domains interconnected by CL links provided that its potential connectivity is not changed. This is a composite FD made of component flow domains (cFDs). The process is recursive until a limit is reached – a smallest/indivisible FD (matrix FD (MFD) within an NE), for example. A flow domain may also be recursively aggregated from nested sets of "smaller" FDs. The set of (potential or actual) edge points of the partitioned FD is a proper subset of the set of all edge points of all FDs/cFDs at the next level. For each flow domain, its set of edge points remains unchanged under partitioning while edge points are revealed at lower levels which are internal to it. This form of decomposition generates new reference points. For example, a layer network may have an international portion and national portions where each national portion has regional/access portions with attached sets of potential TCPs (see Figure 4 of [ITU-T G.809] and Figure 7-14 of [ITU-T Y.1314]).

– **CL link partitioning** (or serial/sequential CL link decomposition) (based on clause 6.4.2.2 of [ITU-T G.809] and link partitioning)

  A CL link may be partitioned into a serial arrangement of "shorter" CL links. This is a compound CL link made of CL component links (cLinks) that are contiguous (link chain or serial-compound link). Between two component links may or may not be a flow domain. The CL cLinks may themselves be further serially partitioned. A CL link may also be recursively aggregated in a serial fashion from "shorter" CL links. This form of decomposition generates new reference points.

– **CL link fragmentation** (or parallel CL link decomposition) (based on clause 6.3.2.5.3 of [ITU-T G.8010] and link fragmentation)

  A CL link may be recursively aggregated by bundling a set of CL links that are equivalent for the purpose of routing. Conversely, a CL link may be fragmented into a parallel arrangement of "thinner" CL links "of equal length". This is a compound CL link made of CL fibre links (fLinks) that are bundled (link bundle or parallel-compound link). The CL fLinks may themselves be further partitioned in parallel. This form of decomposition does not generate new reference points since the fragmentation of a CL link corresponds to simultaneous partitionings of its (two) sets of (potential or actual) edge points.

– **Flow domain fragmentation** (or parallel flow domain decomposition) (according to clauses 6.3.2.5.1, 6.3.2.5.2, Annex A of [ITU-T G.8010] and clause 7.6 of [ITU-T Y.1314])

A flow domain may be fragmented into a set of "smaller" flow domains without interconnecting links by decomposing its set of (potential or actual) edge points without adding any interior reference point. This is a composite FD made of flow domain fragments (SNFrs). The set of (potential or actual) edge points of the fragmented FD is equal to the set of all edge points of all FDs/FDFrs at the next level.

An edge point of a fragmented FD is considered to belong also to one (and only one) FDFr (since, otherwise, either the FD would disappear or FDFrs would have no edge points). This process is not recursive. Fragmentation of a flow domain usually lowers its potential connectivity since no (potential) connectivity is provided between the fragments. This form of decomposition does not generate new reference points. Flow domain fragmentation may be accompanied by CL link fragmentation.

– **Virtual connection partitioning and fragmentation**

The general definition of a virtual connection (VC) for CL layer networks is for further study.

It should take the concepts of FDFr and CL fLink into consideration and distinguish between the point-to-point (P2P), point-to-multipoint (P2MP) and multipoint-to-point (MP2P) natures of flows (as well as multipoint-to-multipoint (MP2MP) topologies for bidirectional scenarios). It should also consider the decomposition variants through expansion of adaptation functions, flow termination functions and FPs/TFPs (with or without FPP consideration). The corresponding general definitions of synchronized VC partitioning and fragmentation, derived from FD and CL link partitioning and fragmentation and based on connection partitioning and fragmentation, are also for further study.

NOTE 7 – Since instantiated FPs and TFPs are considered to always be in "connectivity state" actual, CL topology management may be independent of CL connectivity management (e.g., broadcast connectivity).

10.4.3 The lightweight unified functional approach/architecture for transport networks

This clause progresses the initial stages of the common framework for the functional modelling of CO and CL layer networks, as provided by [ITU-T G.809] and [ITU-T G.8010], and summarized and enhanced in clause 10.4.2 from the layer network and TMN functional viewpoints, towards the lightweight unified functional approach/architecture for transport networks (lightweight UFATN). It presents first requirements and modelling principles for this forward-looking approach.

Depending on the nature of the associated traffic units, each transport network technology can be mapped to one of three network modes: connection-oriented circuit-switched (CO-CS), or connection-oriented packet-switched (CO-PS), or connectionless packet-switched (CL-PS). Refer to Annex A of [ITU-T G.809] for a comparison of CO-CS, ATM, IP and Ethernet. MPLS appears in a special role as there are connection-oriented P2P or P2MP LSPs, and connectionless MP2P LSPs with respective dedicated use of a control plane protocol (see Table 5-2 of [ITU-T Y.1314]).

Accordingly, there are nine types of client/server combinations where some combinations are more compatible than others (see Table 5-3 of [ITU-T Y.1314] for some drawbacks). The lightweight UFATN should support all combinations in a unified manner. The following order of priority is proposed:

1)	CO-CS client and CO-CS server (e.g., SDH/SONET-over-OTN).
2)	CO-PS client and CO-CS server (e.g., ATM-over-SDH/SONET).
3)	CO-CS client and CO-PS server (e.g., PDH-over-ATM).
4)	CL-PS client and CO-CS server (e.g., Ethernet-over-SDH/SONET – see Figure 8 of [ITU-T G.809]).
5)	CL-PS client and CO-PS server (e.g., Ethernet-over-P2P-MPLS).
6)	CL-PS client and CL-PS server (e.g., IP-over-Ethernet).
7)	CO-PS client and CO-PS server (e.g., ATM-over-P2P-MPLS).
8)	CO-PS client and CL-PS server (e.g., ATM-over-IP).
9)	CO-CS client and CL-PS server (e.g., PDH-over-IP).

Figure 4 – Network mode client/server combinations

Main examples of these combinations are provided by the concept of hybrid circuit/packet network (HCPN) developed in [ITU-T M.3017] which is based on the concepts of (single-layer) technology-specific network (TSN) and multi-technology network (MTN). Using these concepts, [ITU-T M.3017] specifies seven options of integrated management of HCPNs which rely on semantic mediation between TSNs. This approach assumes different EMSs (implementing different interface information models) for different TSNs which are integrated over one or two levels of NMSs with the help of mediation functions. A fundamental requirement on lightweight UFATN development is to avoid this type of layer integration and understanding of MTN. Instead, it is required to develop a multi-layer management interface which integrates all layers of the MTN. The interface information model of the lightweight UFATN shall offer unified multi-layer managed objects for the MTN instead of different single-layer managed objects for each TSN.

For the case of termination points, some principles on how to form such lightweight managed objects are depicted in Figure 5 which stems from the MTNM functional model [b-TMF layers]:

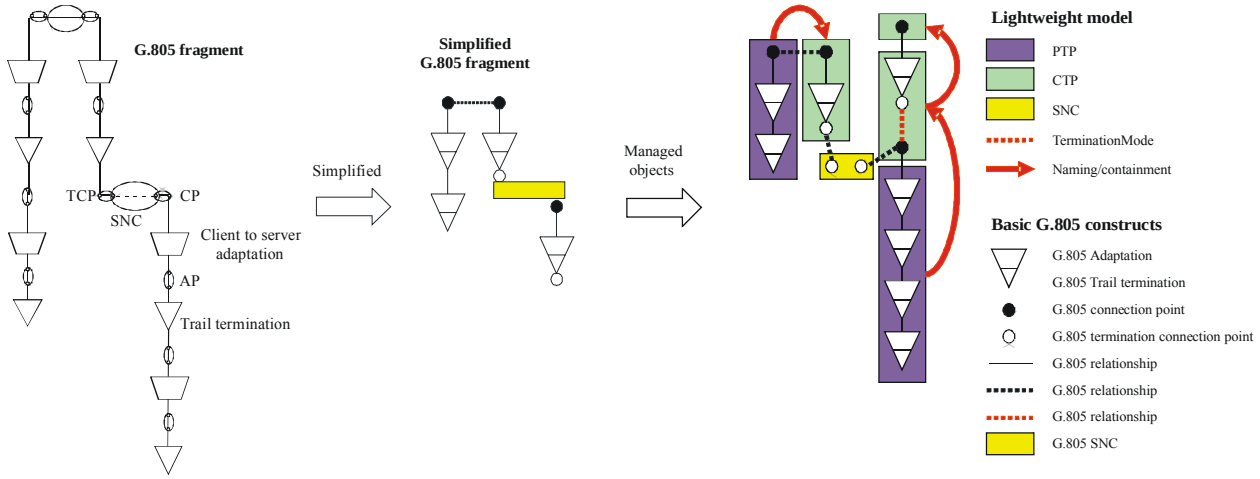

Figure 5 – G.805 transmission layering and lightweight simplification

Figure 5 shows how to encapsulate basic G.805 constructs in lightweight termination points that have a naming/containment relationship and a termination mode to indicate their current ability for providing connectivity. This figure refers to [ITU-T G.805] (as does [b-TMF layers]), i.e., CO layers only, but the lightweight UFATN will also need to develop equivalent encapsulations for the connectionless constructs of [ITU-T G.809] to meet the requirements of Figure 4. This should be seen in the context of G.852.2 revision to provide the generic CO and CL TMN functional model (see clause 10.4.2).

The lightweight UFATN would provide unified definitions of topological components (multi-layer subnetworks and links), connectivity components (multi-layer subnetwork connectivities and fibre links with an identified connection-oriented or connectionless connectivity layer) and edge points (multi-layer termination points, potential and actual, where an actual TP has a termination mode with regard to its connectivity layer) all of which encapsulate the respective G.805 and/or G.809 constructs. The definitions would cover the client/server combinations listed in Figure 4 and the decomposition behaviour described in clauses 10.4.1 and 10.4.2 (partitioning and fragmentation of multi-layer topology and connectivity components). They would provide means to provision VPNs which are dependent on the transport network technologies represented by the layers (including appropriate intra-layer decomposition or sublayering for technologies such as Ethernet).

10.5 Modelling of IDL extensions

When adding new functionality to the service-oriented CORBA TMN interface, it usually becomes necessary to upgrade the existing managed object classes to capture new information, or to create new object types. However, backward and forward compatibility constraints make this and other IDL extensions complicated in some cases. Instead of using a different solution for each individual case, this clause proposes some lightweight extension mechanisms that could be used whenever a complex change needs to be made to IDL definitions. The smart extensibility features apply to both standard extensions and vendor-specific extensions of the framework.

10.5.1 Backward and forward compatibility

The TMN CORBA framework specifies OS-OS interfaces according to [ITU-T M.3010], where one OS takes a client/manager role (e.g., an NMS) and the other OS takes a server/agent role (e.g., an EMS). Compatibility of such interfaces refers to the ability of cooperation of different client and server versions. Backward and forward compatibility can be defined as follows:

– Backward compatibility is a characteristic that allows an upgrade of a client to be made while still being able to interact with the current servers.

– Forward compatibility is a characteristic that allows an upgrade of a server to be made while still being able to interact with the current clients.

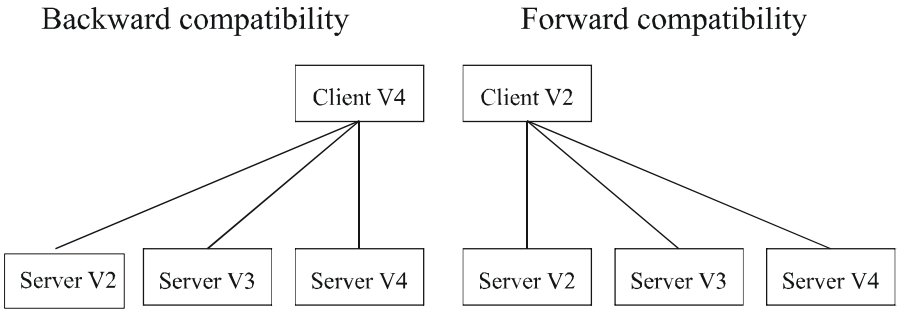

Figure 6 – Examples of backward and forward compatibility

Generally speaking, backward compatibility is easier to achieve than forward compatibility because of semantics considerations. From one version of the interface to the next, functionality increases but existing functionality usually remains unchanged. As a consequence, some concepts representable in version N may not be representable in version N-1, and a server built for version N may not be able to adequately represent its network using version N-1 because of missing concepts. On the other hand, a client built for version N can interact with a server built for version N-1, as all its functionality is still valid in version N.

The consequence of having neither backward nor forward compatibility is that all clients and servers in a network must operate using the same version of the interface. When an upgrade is made, all clients and servers have to be upgraded at the same time. This is clearly an unacceptable limitation.

The consequence of having only forward compatibility (and not backward compatibility) is that a client using version N of the interface would not be able to manage a server using version E < N. This means that an upgrade to a client would first require an upgrade in all servers in the network. This is a significant limitation, as any client upgrade is then dependent on several server upgrades.

The consequence of having only backward compatibility (and not forward compatibility) is that a client using version N of the interface would be unable to interact with a server using version E > N of the interface. This means that an upgrade to a server would first require an upgrade in the clients. Since there is usually a small number of clients, this is a less stringent limitation.

Finally, when both backward and forward compatibility are provided for the versions of an interface specification, then the clients and servers of any deployment become independent of the versions of the interface used in the others. This is obviously a significant advantage.

10.5.1.1 Versioning

Backward compatibility is often achieved in clients by implementing all the versions of the interface to be supported and specific processing depending on the version of the interface used by each server. For example, a client operating with version 3 of the interface would have code to operate with servers using version 3, as well as different code to deal with servers using version 2. In this case, the client must use both version 2 and version 3 of the interface at the same time. This is what is called backward compatible versioning.

Likewise, forward compatible versioning can be achieved in servers by implementing all the versions of the interface to be supported and specific processing depending on the version of the interface used by each client. This can be done by having the server provide several versions of the interface simultaneously. The clients select which one(s) to use through the naming service.

To allow backward and forward compatible versioning, the service-oriented CORBA interface must therefore have the following characteristics:

– The client must be able to determine the IDL version of the interface used by the server.
– The server must be able to provide different IDL versions of the interface simultaneously.
– The client must be able to communicate at the same time with servers using different IDL versions of the interface.
– The interface specification must guarantee to avoid IDL name clashes between versions.

Refer to clause 10.5.4 for details on how the service-oriented CORBA TMN framework allows to meet these requirements and use the naming service for programmatic versioning.

10.5.1.2 Interface compatibility

Backward and forward compatibility can also be achieved through interface backward and forward compatibility. When the interface itself is fully backward compatible, no version-specific code is required in the client to interact with servers using different versions of the interface; the client can

use only its latest version of the interface. Likewise, when the interface itself is fully forward compatible, no version-specific code is required in the server to interact with clients using different versions of the interface; the server can use only its latest version of the interface. Interface backward and forward compatibility is obviously an interesting and useful characteristic of an interface; however it is usually quite difficult to achieve.

To ensure interface backward and forward compatibility, rules should be established regarding what kind of changes of IDL constructs (modules, interfaces, types, attributes, operations) could be made from one version to another of the interface. Such rules are for further study. For example, adding and removing service-oriented façades to and from the interface defined by this framework is backward and forward compatible provided the server implements the session service which offers the getSupportedManager operation for version-independent determination of the actually supported service-oriented façades (see clause 9.1.2.2 of [ITU-T Q.816.2]). The encoding of single-valued, multi-valued and tagged parameters as lists of name/value pairs (see clause 8.3) and the lightweight notification concept (see clause 8.7) are forward compatible examples since clients can (and should) ignore information when they do not understand it.

10.5.1.3 Versioning versus interface compatibility

It is important to note that interface compatibility and versioning address the same issue, but in two completely different ways. Versioning requires each server and client to implement several versions of the interface simultaneously in order to communicate with clients or servers implementing different versions of the interface. In contrast, interface compatibility allows each server and client to implement a single version of the interface while still being able to communicate with other clients or servers that implement different versions of the interface.

Generally speaking, interface compatibility is easier to handle than versioning from a product design perspective. However, achieving interface compatibility generally imposes more requirements on changes than allowing versioning.

10.5.1.4 Guidelines for backward/forward compatibility

There are nine different ways of tackling backward/forward compatibility in an object-oriented interface, which are summarized, with recommendations, in Table 4.

Table 4 – Combinations of backward/forward compatibility

		Backward compatibility		
		None	**Versioning**	**Interface compatibility**
Forward compatibility	**None**			3rd choice
	Versioning			2nd choice
	Interface compatibility			1st choice

For an object-oriented interface, it is to be expected that backward compatibility will be used more than forward compatibility. That is, it should happen more frequently that a new client version is installed before new server versions than the opposite. Indeed, it is not very useful to upgrade a server if the client cannot take advantage of any of the new features provided. For this reason, it is recommended to enforce interface backward compatibility. Backward compatibility could also be achieved through versioning, but this is less interesting and more difficult to manage since it requires that the clients implement every single version of the interface. Using interface backward compatibility tremendously reduces the overall complexity of the interface, as changes can be made without adding new IDL constructs such as types, interfaces, operations. With versioning, any change to an IDL construct requires entirely new additional IDL constructs (see clause 10.5.4).

Forcing interface forward compatibility would be quite nice from a product design perspective. However, interface forward compatibility imposes stringent requirements on top of interface backward compatibility requirements and calls for specific design considerations (see clause 10.5.1.2 for some examples), which makes this task quite difficult. Of course, even if interface forward compatibility were not identified as an absolutely necessary requirement, an interface forward compatible change should still be preferred to a non-forward compatible one, since it simplifies the overall management of different interface versions. Providing forward compatible versioning turns out to be quite complicated as well, and it is not clear that it is required. Forward compatibility could be provided, when required, through versioning in each server.

It is therefore recommended to make every attempt to render both interface backward compatibility and interface forward compatibility (1st choice). However, it is also recognized that this imposes a lot of restrictions on the types of changes that can be made. As a consequence, if at some point the requirements turn out to be too stringent and do not allow proper implementation of a significant feature, then it is recommended to revert to the 2nd choice, which is to force interface backward compatibility, but to allow forward compatibility through versioning. This is recognized to be an optional feature in the servers, since it requires that they implement different versions of the interface and so, as the 3rd choice, it is recommended to provide interface backward compatibility with no forward compatibility. However, the service-oriented interface itself would in all cases allow both backward and forward interface compatibility.

10.5.2 Extension mechanisms

Instead of using a different solution for each individual case, there are extension mechanisms that could be used whenever a change needs to be made to an existing managed object class (MOC) or when new object types need to be added to enhance the functionality of the interface.

Two methods of implementing an extension mechanism for MOCs in the IDL are described here and compared. The first one relies on the `any` type. The second is based on the `valuetype` type.

A backward compatible extension mechanism for the `enum` type is introduced as well.

10.5.2.1 The `any` type

The `any` type is a predefined CORBA type, which has special capabilities. An object of type `any` carries two components:

– a value of any definable CORBA type; and

– a full description of the type of the value, called a type code.

The `any` type is a very powerful tool in CORBA. However, because an object of type `any` carries its type code, it is not very efficient in the cases where the type of the value is actually known in advance. Care must therefore be exercised when using this type. For example, when an operation parameter of type `any` conveys a list of values of another type that is actually known in advance (e.g., a `sequence<Common_T>`), representation of the parameter value by the concrete actual type will result in a type code that includes this type and its subtypes only once, while representation by a `sequence<any>` would include the actual type and its subtypes for all elements of the list.

At first sight, there may be an issue with the `any` proposal since `any` is known to sometimes give rise to performance shortcomings and is not much tested in CORBA environments. But, for a concrete application, the `any` approach just means to specify attachable extensions of managed object classes by using service-oriented names (see clauses 8.5 and 10.3) as the attachment mechanism, where an attachable service-oriented extension is defined as a `struct` declaration that includes `globaldefs::NamingAttributes_T name;` as a mandatory record member. Such declarations usually will not include any record members of type `any`. However, there will be get, iterator and set operations with parameters of type `any` for which it is completely up to the server to

take care for an efficient implementation by carrying an appropriate non-`any` object within the `any` type similar to the usage of actual field types within OMG structured events.

10.5.2.2 The `valuetype` type

The type `valuetype` is a feature of CORBA that allows object inheritance for `struct`-like objects. Before that feature was introduced in CORBA 2.3 [OMG 99-10-07], inheritance in CORBA was only supported for `interface` objects.

An object of type `valuetype` carries one or more components. There is one component for each level of inheritance, and each component identifies the `valuetype` class and the values associated with it. Note that only the identifiers of the value types are required, and not the full descriptions of the value types. This is much more efficient than type codes used for `any` objects but, of course, also provides far less flexibility.

As an example, consider the following declarations:

```
valuetype Class1_V {
  string name;
};
valuetype Class2_V : Class1_V {
  int number;
};
valuetype Class3_V : Class2_V {
  short info;
};
```

An object of type `Class3_V` would contain three components, as follows:

– `Class1_V`: the value of name;

– `Class2_V`: the value of number;

– `Class3_V`: the value of info.

Because that object inherits from `Class2_V` and `Class1_V`, it can be used in the interface anywhere a `Class1_V` or a `Class2_V` object is expected.

Furthermore, if a client expects a `Class2_V` object, for example, and does not know that `Class3_V` exists because it uses an older version of the interface, the `valuetype` mechanism still works because the client ignores the `Class3_V` component.

The `valuetype` feature therefore provides an easy way to ensure backward and forward compatibility. The extension mechanism itself depends on whether a non-`valuetype` object (i.e., a `struct` object) shall be extended (explicit extension via attachment – see clause 10.5.2.1) or a `valuetype` object (implicit extension via inheritance). Further details (e.g., combination with the generic iterator approach of clause 9.4.2) are for further study.

10.5.2.3 Comparison and suggestion

The comparison of the two approaches described above indicates that there are very few advantages to the `valuetype` approach for the extension mechanism. In summary, the main advantage of the `valuetype` approach is that it allows an additional way of extending objects (through inheritance) where such modelling flexibility is needed (but see the "lightweight inheritance" approach using `struct` described in clause 8.5 for cases where, deliberately, very few MOCs are defined), while the main advantages of the `any` approach are its consistency (a single approach for all objects) and its improved efficiency. Based on those observations, it is recommended to use the `any` approach described in clause 10.5.2.1 for the extension mechanism. Note that CORBA 2.3 offers improved type code techniques for introspection and manipulation of the IDL type `any`.

10.5.2.4 Extending `enum` types

The IDL type `enum` is an ordered list of identifiers, which may be named using a `typedef` declaration. Any modification of such a list is not backward compatible. An enumerated type is extensible if one of its enumerators does not convey any particular meaning but is a sort of default identifier; such an enumerator is called escape value. To extend an old attribute whose type is an extensible enum, one defines – in a backward compatible fashion (e.g., by using the additionalInfo attribute of an SO managed object (see clause 8.5)) – a new attribute of type `string` that covers all values of the old `enum` as well as the new value(s). The escape value of the `enum` is used to indicate that an extension is being used. The proper new value is then found in the new attribute. Refer to clause 8.7 for an example (i.e., `notifications::ObjectType_T`).

NOTE – The recommended extension mechanism for `enum` types is really a work-around to deal with situations that are defects. The overall recommendation is to avoid `enum` types wherever possible and to use instead strings whose admissible values are standardized (per normative comments or `const` definitions).

10.5.3 Adding further service-oriented façade interfaces

When using service-oriented CORBA, a TMN interface is specified as one or more service-oriented façade interfaces defined using IDL. Inevitably, TMN interfaces change. Adding a new service-oriented façade interface to a TMN interface version is straightforward. The new service-oriented façade interface simply needs to be defined in a, preferably new, IDL module (by using these guidelines) and added to the specification identifying the service-oriented façade interfaces to be supported on that particular TMN interface. In the presence of the session service, this mechanism is both backward and forward interface compatible (see clause 10.5.1.2). These considerations also apply to fine-grained and coarse-grained TMN interfaces where CORBA interfaces do not represent SO façades but managed objects and X.780.1 façades, respectively (see clause 6.13 of [ITU-T X.780]).

10.5.4 Versioning of CORBA IDL specifications

Updating an existing CORBA interface is a little more sophisticated than adding new interfaces. The fine-grained, coarse-grained and service-oriented guidelines for defining CORBA managed objects and façades place a priority on backward compatibility. Therefore, the IDL naming rules listed below are recommended when extending an existing CORBA interface (i.e., an MOC in case of fine-grained TMN interfaces, an X.780.1 façade in case of coarse-grained TMN interfaces, and an SO façade in case of service-oriented TMN interfaces). Note that these rules apply only to extensions being made to a base interface that do not result in changing the business purpose(s) of the interface. That is, the new interface models or manages the same resource(s) as the old interface, it simply has some additional capabilities (i.e., operations). Refer to clause 10.5.2 for extensions that do change the structure of resources (i.e., managed object classes).

– The name of the new CORBA interface shall be the name of the existing interface with the letter `"R"` and a numeral appended, starting with `"1"`. Here `"R"` signifies "Revision" or "Revised". Subsequent extensions of the TMN interface will increment the numeral.
– The new interface shall be defined within the same module as the existing interface. (CORBA modules are really just name spaces, and may be spread across multiple files.)
– The new interface shall inherit from the existing interface.
– Capabilities inherited from the existing interface cannot be removed or modified in the new interface. If an operation definition needs to be modified, a new operation must be defined whose name shall be the name of the existing operation with the letter `"R"` and a numeral appended, starting with `"1"`. Subsequent revisions of the operation will increment the numeral.

– References to new interfaces should be of the most specific type. (If they are not, the new capabilities cannot be accessed.) CORBA provides some means for determining the actual interface definition of a reference based on information contained in the IOR.

– In case of fine-grained and coarse-grained interfaces, the rules on CORBA name bindings and scoped interface names stated in clause 6.13 of [ITU-T X.780] shall be obeyed.

– A similar approach, appending the name with `"R"` and an incremented number starting with `"1"`, shall be used when other existing IDL constructs are revised such as constant definitions, type definitions (but keeping the suffix `"_T"`) and value type definitions (but keeping the suffix `"_V"`), provided such extensions can be made in a backward compatible manner.

The service-oriented framework adds further lightweight capabilities for IDL versioning that allow to meet the requirements stated in clause 10.5.1.1. These solutions are listed below:

– The client must be able to determine the IDL version of the interface used by the server.

  • The session service provides this capability (see clause 9.1.2.4 of [ITU-T Q.816.2]). The client must use the getVersion operation of the correct server session factory object (see clause 9.1.2.5 of [ITU-T Q.816.2]).

– The server must be able to provide different IDL versions of the interface simultaneously.

  • The session service provides this capability collectively with the lightweight CORBA naming trees (see clauses 9.1.2.5 and 6.4.3 of [ITU-T Q.816.2] and below). The server must register one server session factory object per supported IDL version with the naming service.

– The client must be able to communicate at the same time with servers using different IDL versions of the interface.

  • The session service provides this capability (see clause 9.1.2 of [ITU-T Q.816.2]). The client must maintain a (secure) session (i.e., a pair of associated client session and server session objects) with each provided server session factory of each of the servers to be taken into account.

– The interface specification must guarantee to avoid IDL name clashes between versions.

  • The IDL naming rules specified above and the so-called programmatic versioning described below provide this capability.

The service-oriented framework recommends a lightweight use of the OMG naming service as specified in clauses 6.4 and 8.2 of [ITU-T Q.816.2]. In particular, all CORBA naming graphs should be lightweight trees of the type outlined in clause 6.4 of [ITU-T Q.816.2]. This means that the possible kinds of nodes (i.e., the admissible values of the kind attribute) and their ordering are prescribed. Only the kinds of CORBA name components listed below should be used with the SO framework. This lightweight usage includes a multi-vendor capability a versioning capability, an OS dependency capability and naming rules for server session factories and service-oriented façades as well as event channels. For each lightweight CORBA name component, the syntax of its `id` attribute is listed as well. The tree structure of the lightweight components is specified in Table 5 subsequent to the list.

– "Class":
Denotes an OS-OS interface class that is specified in CORBA IDL. The id is a free format human-readable string. This corresponds to the domain name field in the fixed event header of structured events (see clauses 8.7.1 of this Recommendation and 6.5 of [ITU-T Q.816.2], and Table 2 of [ITU-T Q.816.2] for examples).

– "Vendor":
Denotes an OS vendor. The id is identified as <CompanyName>, which is a free format human-readable string as specified by the OS vendor company.

– "EmsInstance":
Denotes a server OS (e.g., an EMS). The id for this kind is defined as <EMSvalue> ::= { <CompanyName>"/"<EMSname> | <CompanyName>":"<EMSname> }, where <CompanyName> has the same value as the id attribute of the parent component of kind "Vendor", and <EMSname> is a free format human-readable string as specified by this OS vendor company. The separator "/" is the preferred choice but ":" is also allowed since "/" has a reserved usage in the OMG naming service and might require use of escape characters.

– "Version":
Denotes an IDL version of the parent OS-OS interface class. The id has a similar value as the version string associated with the child component of kind "EmsSessionFactory_I" (see clause 9.1.2 and A.1 of [ITU-T Q.816.2]), namely id ::= { <*Release*>"."<*Major*>["."<*Minor*>] | <*Release*>"_"<*Major*>["_"<*Minor*>] }, where Release, Major and Minor are specified as part of idlVersion::Version_I. The separator "." is the preferred choice but "_" is also allowed since "." has a reserved usage in the OMG naming service and might require use of escape characters (depending on the naming service implementation deployed).

– "EmsSessionFactory_I":
Denotes a server session factory (see clause 9.1.2.5 of [ITU-T Q.816.2]). The id value is equal to the <EMSvalue> of the parent component of kind "EmsInstance".

– "Common_I":
Denotes a generic service-oriented façade (i.e., an instance of Common_I – see clause 9.3). The id value is equal to the <EMSvalue> of the parent component of kind "EmsInstance" if the singleton design pattern applies for Common_I (see clause 7.2.1.1), not counting descendents. Otherwise, the id is <EMSvalue>"n" where n enumerates the direct instances of Common_I.

– <SO façade interface name>:
Denotes a specific service-oriented façade (i.e., an instance of a direct or indirect descendant of Common_I – see clause 9.3). The kind value is the IDL name of the SO façade interface, prepended by <module name>"::" if not yet unique. The id is defined as for the "Common_I" component (but considering the singleton pattern for <SO façade interface name>).

– <EventChannel>:
Denotes an OMG event channel (e.g., a notification channel, a NotifyLog). The kind value characterizes the type of event channel according to the OMG notification service and telecom log service specifications: "EventChannel" is a CosEventChannelAdmin::EventChannel, "NotifyChannel" is a CosNotifyChannelAdmin::EventChannel, "EventLog" is a DsEventLogAdmin::EventLog and "NotifyLog" is a DsNotifyLogAdmin::NotifyLog. The id value equals the <EMSvalue> of the parent component of kind "EmsInstance" if the singleton design pattern applies for <EventChannel> (see clause 7.2.1.1), for example, in the presence of the session service (see clause 9.1.2.2 of [ITU-T Q.816.2]). Otherwise, the id is <EMSvalue>"n" where n enumerates the instances of CosEventChannelAdmin::EventChannel.

– <vendor-specific interface name>:
Denotes a vendor-specific interface instance. The kind value is the IDL name of the interface, prepended by <module name>"::" if not unique otherwise. The id is defined as for the "Common_I" component (but considering the singleton pattern for <vendor-specific interface name>). This component type is required to provide backward compatibility and to allow for integration of fine-grained and coarse-grained TMN interfaces.

Figure 7 – Kinds of lightweight CORBA name components

The set of lightweight CORBA name components is hierarchically ordered as follows:

Table 5 – Lightweight CORBA naming tree

Initial naming context → Class → Vendor → EmsInstance → Version → { EmsSessionFactory_I | Common_I | <SO façade interface name> | <EventChannel> | <vendor-specific interface name> }

Each arrow represents a binding; Class, Vendor, EmsInstance and Version represent context IORs (hollow nodes); and all other components represent application IORs (solid/filled nodes) (see clause 6.4 of [ITU-T Q.816.2]). Refer to Figure 1 of [ITU-T Q.816.2] for an example of a lightweight CORBA naming tree.

The application IORs represent the entry points for clients to interact with servers. When all servers involved in a deployment construct a common lightweight CORBA naming tree and all clients use this tree, the tree structure guarantees that different IDL versions will result in different client entry points. This standardized determination of version-specific entry points is called programmatic versioning. It develops its full strength in collaboration with the session service (see clause 9.1 of [ITU-T Q.816.2]).

An example for a physical realization would be two platforms, one running a server OS and the other a client OS. The server OS would implement the session service and instantiate a server session factory object for each supported IDL version in a separate process, and the client OS would instantiate for each server session factory a separate process for session-based communication. Then IDL versions would even be separated by process boundaries on the platforms.

10.5.5 Vendor-specific extensions

The built-in extension mechanisms of this framework enable the deployment of multi-technology network management solutions in multi-vendor (and even multi-customer) networks. They provide the foundation for an agile vendor-independent CORBA data model (with submodels for behaviour and state of managed objects) that offers smart means for vendor-specific extensions – when such extensions are needed in customer projects or for reasons of competitive differentiation – thereby allowing for a very high degree of interoperability.

The CORBA modelling guidelines of the service-oriented façade design pattern (see clauses 7.2.1 and 9) provide standardized flexibility for adding vendor-specific extensions. For example:

– vendor can add new façades and associated managed object classes (see clause 10.5.3);
– vendor can add new name/value pairs in the structure of the naming attribute (see clause 10.3);
– vendor can add functionality without changing the existing IDL (e.g., additional info parameters or layered transmission parameters).

The smart extensibility features guarantee that interface specifications that extend this framework become highly interoperable. Vendors should nevertheless greatly endeavour to standardize their extensions – where vendor extensions are not too company-/technology-specific and not intended for differentiation purposes – thereby contributing to the growth of the framework standard.

10.6 Additional CORBA modelling guidelines and principles

It seems feasible to specify a sort of "service-oriented TMN interface design paradigm" but the benefits of such a major effort should be identified first. This novel paradigm for developing and exposing new TMN objectives in a lightweight fashion could be based, e.g., on a revision of the following service-oriented CORBA modelling principles that includes examples for each step:

– assign new objectives, preferably to existing managed objects;
– introduce new managed object classes (according to clause 8.5) only if the existing classes cannot (easily/naturally) meet the new objectives;
– extend existing managed object classes if it seems advisable and an efficient extension/revision mechanism (for structs) is available (e.g., from clause 10.5);
– assign new managed object classes, preferably to an existing SO façade, since SO façades are intentionally meant to manage groups of several well-related MOCs, and introduce new

SO façades only for the management of vendor-specific or completely new groups of MOCs;

– specify containments and resulting names for the new managed object classes and, if applicable, extensions to existing MOC containments and names;
– model new behaviour of existing managed objects and behaviour of new managed objects through operations of existing (preferred, if reasonable) or new SO façades;
– try to keep façade operations technology-independent, since technology-specific issues are meant to be placed inside managed objects as attribute values (and not attribute names), for example, as parts of object names or within multiple transmission layers, or as additional info parameters;
– for objectives not yet modelled otherwise, define additional info parameters of appropriate managed objects or appropriately extend existing capabilities (e.g., add new PM parameters, add new probable causes, add new layer rates, add new layered transmission parameters, add new common façade operations, add new notification types, add new capability semantics);
– clarify the G.805/G.809 layering for each technology to be supported at the interface; introduce and document new transmission Layer rates and client/server adaptations, if required; for the existing and new transmission layers involved in the new objectives/technologies, specify new transmission parameters or the (possibly extended) use of existing parameters; within the scope of clause 10.4, specify new multi-layer encapsulations of managed objects and introduce new diagrammatic conventions regarding new TP or subnetwork layering varieties, if required;
– adhere in the greatest detail to the CORBA IDL writing style used in Annex A and in Annex A of [ITU-T Q.816.2], and to the further IDL style idioms specified in clause 11;
– ensure consistency/compatibility and sustainability of any model progression.

The development of this paradigm as sound principles, similar to the naming guidelines of clause 10.3, and its applications are for further study. It need not at all be bound to CORBA IDL as the modelling language but could be geared to service-oriented XML (WSDL, XSD, etc.) modelling. The main objective should be to develop a two-part service-oriented CORBA meta-model, based on clause 10.2, for the state and behaviour of all sort of managed objects. Such a meta-model would greatly facilitate the design of any kind of multi-technology OS-OS interface.

It is believed that the novel service-oriented approach to CORBA-based TMN interface design can pave the way for introducing SOA principles to ITU-T's TMN and NGNM interface specification methodology (M.3020-series of ITU-T Recommendations). The most basic SOA principles and artefacts have been defined and described in considerable detail and a mapping between SOA and CORBA concepts has been provided (see clauses 3 and 7.2.1.2 of [ITU-T Q.816.2]). The application of these principles and mapping for the development of SOA modelling guidelines is for further study.

The flexibility of the service-oriented interface design principles allows mapping to other CORBA design approaches chosen by the telecom industry, notably the MTNM model of TM Forum's MTNM team and the IRP model of 3GPP's OAM&P working group SA5. These model mappings are out of scope of the CORBA framework but are an essential part of the proof of M.3010 TMN conformance for these models. The overall guidelines for the construction of such mappings are given by the compliance and conformance rules of clause 13. In case of MTNM, the proof is carried out in ITU-T Rec. M.3170.3, Multi-technology network management: CORBA IDL solution set (TMF814) with implementation statement templates and guidelines (TMF814A). In case of IRP, the proof is for further study.

Revisions of all applicable modelling guidelines of [ITU-T X.780] and [ITU-T X.780.1], which have not yet been considered so far, are for further study with regard to the service-oriented framework (e.g., use of pragmas and more constants, introduction of conditional packages versus enhanced capability model). As a result of such a study, several commonalities and differences of fine-grained and coarse-grained modelling versus service-oriented modelling should become fairly evident.

It has been pointed out in clauses 8.4, 8.7.3.1, 8.7.4.1 and 10.3.2 that, in the interest of interoperability, users of the service-oriented CORBA framework for the TMN must cooperate in reaching normative usage of, for example, standardized layer rates (and associated transmission parameters), probable causes, PM parameters and managed object naming. Whilst details of such cooperation are for further study (but see clause 10.3.2 for some suggestions), it has also been required to build on the available related standardization work of MTNM [TMF LayerRates], [TMF probableCause], [TMF PerformanceParameters], [TMF objectNaming]. The overall recommendation is to foster the foundation of the service-oriented CORBA TMN framework user group as a steering group that coordinates users of this framework who specify, and wish to standardize, interface extensions of the framework.

11 Style guide for lightweight CORBA IDL modelling

This clause extends clause 8 of [ITU-T X.780] "Style idioms for CORBA IDL specifications" to include an IDL style guide for lightweight IDL writing. It takes the following references into consideration:

- applicable parts of clause 8 of [ITU-T X.780];
- OMG's IDL style guide [b-OMG 98-06-073];
- 3GPP's IDL style guide (see Annex D of [ETSI TS 132 150]);
- IDL style of Annex A and Annex A of [ITU-T Q.816.2] (i.e., the IDL style of the service-oriented modelling and framework support services IDL).

The completion of this clause is for further study.

12 Guidelines for redesigning fine-grained and coarse-grained models to service-oriented

This clause complements clause 10 of [ITU-T X.780.1] "Guidelines for translating fine-grained models to coarse-grained" to include guidelines for the transition from fine-grained and coarse-grained IDL model designs to service-oriented IDL models. It consolidates and formalizes the considerations of clauses 6.7 and 7.3 of [ITU-T Q.816.2] and relates them to clause 10 of [ITU-T X.780.1]. It is based on the observation that a coarse-grained design can be considered a service-oriented design, though with a lower degree of lightweightiness. This degree can be improved by chosing, in the course of the design process, suitable options of the service-oriented framework requirements and guidelines.

The completion of this clause is for further study.

13 Service-oriented compliance and conformance

This clause defines the criteria that shall be met by other standards documents claiming compliance to these guidelines and the features of service-oriented IDL modelling that shall be implemented by operations systems claiming conformance to this Recommendation. Since conformance to this Recommendation includes a more flexible conformance to [ITU-T X.780] and [ITU-T X.780.1], this clause also summarizes the compliance points and conformance points of [ITU-T X.780] and [ITU-T X.780.1] in a convenient manner.

13.1 Standards document compliance

Any specification claiming compliance with the service-oriented CORBA framework shall:

1) Provide a description of how its constituent documents, or document sections, and further deliverables (such as CORBA IDL files) can be split between the two aspects "information modelling in IDL" and "ORB and CORBA services usage" of CORBA-based TMN interface specification.

2) Support all the standards document compliance requirements related to the "ORB and CORBA services usage" aspect as stated in [ITU-T Q.816.2].

3) Meet the following standards document compliance criteria related to the "information modelling in IDL" aspect:

   - specify a usage of the CORBA 2 IDL repertoire (see chapter 3 of [OMG 98-07-01], [OMG 99-10-07], [OMG 01-02-33]) that complies with [ITU-T X.780.1] (coarse-grained foundation IDL) or with this Recommendation (service-oriented foundation IDL) (see clause 13.2.1);
   - in case of fine-grained and coarse-grained interfaces, support some or all of the 14 standards document compliance requirements of clause 9.1 of [ITU-T X.780] according to a conformance profile (see clause 13.2.2);
   - in case of coarse-grained interfaces, follow the coarse-grained CORBA modelling guidelines and the fine-grained IDL to coarse-grained IDL mapping rules defined in clauses 9 and 10 of [ITU-T X.780.1];
   - in case of fine-grained and coarse-grained interfaces, specify, along the lines of clauses 6.7 and 7.3 of [ITU-T Q.816.2] as well as clause 12, how the fine-grained or coarse-grained interface design, according to the two previous criteria, can be redesigned gradually to become more and more service-oriented;
   - in case of service-oriented interfaces, specify how the relevant documents, or document sections, and further deliverables (such as CORBA IDL files) are related to the clauses of this Recommendation;
   - in case of service-oriented interfaces, include one or more IDL files with specifications of IDL modules `globaldefs`, `common`, `transmissionParameters` and `notifications` according to clauses 8 and 9 that are guided by the normative IDL of Annex A (see next criterion);
   - in case of service-oriented interfaces, specify how the normative IDL of Annex A, including comments, can be obtained from the provided IDL files (see previous criterion) without any syntactical changes, except for pragmas, a few minor additional IDL constructs that are considered absolutely necessary and do not at all affect the constructs of Annex A, a few omitted IDL constructs that are not (yet) used, and refactoring of definitions between standard IDL modules and IDL modules claiming conformance, and with only moderate and reasonable modifications with regard to (ordinary or formatted) comments (for example, inclusion of references to additional parts of the specification that are out of scope of the service-oriented CORBA framework).

4) Follow the service-oriented interface design rules defined in clause 10.

13.2 System conformance

This clause first summarizes the conformance points of this Recommendation, [ITU-T X.780.1] and [ITU-T X.780] (i.e., the requirements on IDL repertoire usage and derived modelling of notifications and IDL constructs such as interfaces, operations, exceptions, attributes, data types and constants) from the viewpoint of service-oriented TMN interface design, and then combines them in

several conformance profiles (which are not present in [ITU-T X.780.1] and [ITU-T X.780]) that shall be supported by operations systems claiming conformance to [ITU-T X.780.1] or [ITU-T X.780], or this Recommendation.

An implementation claiming general conformance to this Recommendation shall:

1) In case of fine-grained and coarse-grained interfaces, support, according to a conformance profile, the capabilities of the ManagedObject interface described in clause 5.1 of [ITU-T X.780].

2) In case of fine-grained and coarse-grained interfaces, support, according to a conformance profile, the factory-based create operations behaviour described in clause 6.9 of [ITU-T X.780].

3) In case of coarse-grained interfaces, meet, according to a conformance profile, the façade, factory and iterator instantiation requirements specified in clause 8 of [ITU-T X.780.1].

4) In case of service-oriented interfaces, support, according to a conformance profile, the capabilities of the generic `Common_T` managed object class described in clause 8.5 and the capabilities of the root `Common_I` service-oriented façade interface described in clause 9.3.

5) In case of service-oriented interfaces, meet the service-oriented iterator requirements specified in clause 9.4.

6) Implement an IDL interface compliant with the guidelines in this Recommendation, or in [ITU-T X.780.1] or in [ITU-T X.780]. See clause 13.1.

13.2.1 Conformance points

The individual features of service-oriented IDL modelling described earlier in this document (see clauses 8 and 9), and the features of fine-grained and coarse-grained IDL modelling described in [ITU-T X.780] (see clauses 5 and 6 of [ITU-T X.780] as well as optionally clauses 7 of [ITU-T X.780]) and [ITU-T X.780.1] (see clauses 8, 9 and 10 of [ITU-T X.780.1]), are summarized conveniently as conformance points. These conformance points are combined in the next clause in conformance profiles for the three CORBA framework paradigms (fine-grained, coarse-grained and service-oriented).

1) An operations system claiming conformance to the root managed object requirements shall:

   - in case of fine-grained and coarse-grained interfaces, support the ManagedObjectValueType `valuetype` described in clause 6.10 of [ITU-T X.780] and defined in the CORBA IDL module itut_x780 in Annex A of [ITU-T X.780];
   - in case of fine-grained interfaces, support the ManagedObject `interface` described in clause 5.1 of [ITU-T X.780] and defined in the CORBA IDL module itut_x780 in Annex A of [ITU-T X.780];
   - in case of service-oriented interfaces, preferably support the `Common_T` struct described in clause 8.5 and defined in the CORBA IDL module common in Annex A, and alternatively specify MOCs as described in clause 6.2.2 (`interface`, `struct`, `valuetype`, `sequence`) including the root MOC of the fine-grained or coarse-grained interface for the `interface` or `valuetype` choice.

     NOTE 1 – "Support" means that all object types that model resources (managed object classes) shall be derived (directly or indirectly) from the stated IDL constructs. In case of coarse-grained and service-oriented interfaces, this modelling covers only the resource state.

2) An OS claiming conformance to the root façade (managing object) requirements shall:

   - in case of coarse-grained interfaces, support the ManagedObject_F `interface` described in clause 8.2 of [ITU-T X.780.1] and defined in the CORBA IDL module itut_x780 in Annex A of [ITU-T X.780.1];

- in case of service-oriented interfaces, support the `Common_I interface` described in clause 9.3 and defined in the CORBA IDL module common in Annex A.

  NOTE 2 – "Support" means that all object types that model resource behaviour (managing object classes) shall be derived (directly or indirectly) from the stated IDL constructs.

3) An operations system claiming conformance to the managed object naming and containment requirements shall:

- in case of fine-grained interfaces, adhere to the OMG naming service usage requirements specified in clause 6.1 of [ITU-T Q.816] (see also clause 10.3.2 for a concise overview) and specify name binding modules according to clause 6.8 of [ITU-T X.780] for use in create operations and OMG naming contexts;
- in case of coarse-grained interfaces, adhere to the OMG naming service usage requirements specified in clauses 6.1 of [ITU-T Q.816] and 8.1 of [ITU-T Q.816.1], adhere to the ITU-T containment service requirements specified in clause 9.6 of [ITU-T Q.816.1], specify name binding modules according to clause 6.8 of [ITU-T X.780] for use in create operations and naming contexts, and synchronize both CORBA services;
- in case of service-oriented interfaces, adhere to the guidelines for lightweight managed object naming provided in clauses 10.3.2 and 10.5.4 with optional specification of name binding modules.

4) An OS claiming conformance to the managed object creation requirements shall:

- in case of fine-grained and coarse-grained interfaces, support the ManagedObjectFactory `interface` described in clause 5.2 of [ITU-T X.780] and defined in the CORBA IDL module itut_x780 in Annex A of [ITU-T X.780], together with its generic create operation described in clause 6.9 of [ITU-T X.780];

  NOTE 3 – "Support" means to define, for each managed object class that can be instantiated, a factory interface derived (directly or indirectly) from the ManagedObjectFactory interface, to register it with the factory finder service, and to narrow the create operation.

- in case of service-oriented interfaces, support, for each managed object class that can be created by managing systems, managed object creation by the service-oriented façade object(s) that is (are) responsible for access to and control of this managed object class.

  NOTE 4 – "Support" means to define, for each such managed object class, a create operation of the unique façade interface assigned (in a steward role) to this MOC which is MOC-specific (with regard to, for example, delete policy, initialization of writeable and set-by-create attributes, impact on containing managed objects and co-created contained managed objects).

5) An OS claiming conformance to the managed object deletion requirements shall:

- in case of fine-grained and coarse-grained interfaces, support the TerminatorService `interface` described in clause 7.3 of [ITU-T Q.816] and defined in the CORBA IDL module itut_q816 in Annex A of [ITU-T Q.816], together with the generic destroy operation of the root ManagedObject interface or ManagedObject_F interface described in clause 5.1.7 of [ITU-T X.780] or clause 8.2.9 of [ITU-T X.780.1], respectively;
- in case of service-oriented interfaces, support, for each managed object class that can be created by managing systems, managed object deletion by the service-oriented façade object(s) that is (are) responsible for access to and control of this managed object class.

  NOTE 5 – "Support" means to define, for each such MOC, a delete operation of the unique façade interface assigned to this MOC whose semantics complies with the MOC-specific delete policy.

6) An OS claiming conformance to the façade object naming requirements shall:

- in case of coarse-grained interfaces, adhere to the X.780.1 façade identifier and CORBA name component conventions described in clause 8.1.2 of [ITU-T Q.816.1] (see also clause 10.3 for a concise overview);
- in case of service-oriented interfaces, adhere to the guidelines for service-oriented façade object naming provided in clauses 10.3.3 and 10.5.4 (and clause 6.4 of [ITU-T Q.816.2]).

7) An operations system claiming conformance to the standard data types requirements shall:

- in case of fine-grained and coarse-grained interfaces, use, wherever applicable, the basic typedefs and data types defined in the multi-file CORBA IDL module itut_x780 and the definitions for generic attribute types and common types found in clauses 6.3.5 and 6.3.6 of [ITU-T X.780], and state in its compliance documentation references to the CORBA IDL modules from which other generic attributes and common types are used (if applicable);
- in case of service-oriented interfaces, use the service-oriented IDL repertoire (see Figure 2), and the definitions of basic data types and typedefs described in clauses 8.2, 8.3 and 8.5 and defined in the CORBA IDL modules globaldefs and common in Annex A.

8) An operations system claiming conformance to the constants requirements shall:

- in case of fine-grained and coarse-grained interfaces, use, whenever appropriate, the constants defined in the multi-file CORBA IDL module itut_x780 or in one of its submodules (see Annex A of [ITU-T X.780] , Annex B of [ITU-T X.780] and Annex A of [ITU-T X.780.1]) or in the IDL module itut_q816 (see Annex A of [ITU-T Q.816]), and adhere to the rules provided in clause 6.11 of [ITU-T X.780] when defining new constants;
- in case of service-oriented interfaces, use the constants described in clause 8.7 and defined either in the CORBA IDL module notifications in Annex A or as referenced in clause 8.7, and follow the rules provided in clauses 8.7 and 10.1 when defining new constants.

9) An operations system claiming conformance to the exceptions requirements shall:

- in case of fine-grained and coarse-grained interfaces, adhere to the use of the main exception ApplicationError with few exception codes and the few other exceptions with few exception codes as described in clauses 5.5 and 6.3.4 of [ITU-T X.780], 8 of [ITU-T X.780.1], 7 of [ITU-T Q.816] and 9 of [ITU-T Q.816.1], and defined and used in the multi-file IDL modules itut_x780 and itut_q816;
- in case of service-oriented interfaces, adhere to the use of the unique exception ProcessingFailureException with few exception codes as described in clause 8.6 and defined and used in the IDL modules globaldefs and common in Annex A;
- preferably document per IDL operation the (few) standard OMG exceptions that can be expected during regular processing.

10) An operations system claiming conformance to the notifications requirements shall:

- support the notification service requirements specified in clause 10.2.1 of [ITU-T Q.816.2];
- in case of fine-grained and coarse-grained interfaces, support the notifications `interface` described in clause 5.3 of [ITU-T X.780] and defined in the CORBA IDL module itut_x780 in Annex A of [ITU-T X.780];
- in case of service-oriented interfaces, support the notifications module described in clause 8.7 and defined in the CORBA IDL provided in Annex A.

NOTE 6 – "Support" means to meet the mandatory requirements specified in the respective clause(s).

11) An operations system claiming conformance to the capabilities requirements shall:

- in case of fine-grained and coarse-grained interfaces, provide the GDMO-inspired generic concept of conditional support for groups of properties and capabilities called conditional packages with a corresponding generic ManagedObject.packagesGet operation – as defined in [ITU-T X.780];
- in case of service-oriented interfaces, provide the lightweight two-level capability model to identify the supported operations of a given service-oriented façade (and maybe other capabilities) with a corresponding Common_I.getCapabilities operation – as defined in this Recommendation and with the session service operation EmsSession_I.getSupportedManagers for on-demand determination of the supported service-oriented façades themselves – as defined in [ITU-T Q.816.2].

12) An operations system claiming conformance to the multi-technology telecommunications management (MTTM) requirements shall:

- provide the G.805/G.809-based multi-technology capability approach to telecommunications transmission technologies described in clause 8.4 and defined in the CORBA IDL module transmissionParameters in Annex A, and specialize and evolve this generic approach as described in clause 10.4.

13) An operations system claiming conformance to the service-oriented fault management requirements shall:

- support the MTTM requirements (see conformance point 12 above);
- support the service-oriented FM features described in clauses 8.7.3 and 8.7.5.

14) An operations system claiming conformance to the service-oriented performance management requirements shall:

- support the MTTM requirements (see conformance point 12 above);
- support the service-oriented PM features described in clause 8.7.4.

15) An OS claiming conformance to the special rules and conventions requirements shall:

- in case of fine-grained and coarse-grained interfaces, support further features of [ITU-T X.780]/[ITU-T X.780.1] (and [ITU-T Q.816]/[ITU-T Q.816.1]) not included in the previous conformance points (e.g., follow the GDMO to IDL mapping rules provided in clause 7 of [ITU-T X.780] if the IDL model is a translation from GDMO, adhere to not yet considered conventions specified in clauses 6 and 8 of [ITU-T X.780] for defining CORBA TMN managed objects and writing IDL, use of macros for identifying notifications per MOC, follow the fine-grained IDL to coarse-grained IDL mapping rules provided in clause 10 of [ITU-T X.780.1]);
- in case of service-oriented interfaces, support further features of this Recommendation (and [ITU-T Q.816.2]) not included in the previous conformance points (e.g., be compliant with the IDL style guide of clause 11).

16) An OS claiming conformance to the service-oriented modelling guidelines shall:

- for the features not covered by other conformance points, provide documents, or document sections, that can well serve as contributions of enhancement proposals for the service-oriented CORBA TMN framework standard (this Recommendation [ITU-T Q.816.2]) and its applications, and can be discussed and consented/rejected at ITU-T meetings or at meetings of working teams of relevant SDOs/forums (e.g., TM Forum, 3GPP/3GPP2) with subsequent submission to the ITU-T.

13.2.2 Conformance profiles

This clause combines the conformance points of the previous clause in conformance profiles for [ITU-T X.780], [ITU-T X.780.1] and this Recommendation.

An OS claiming conformance to the X.780 core profile (fine-grained) shall support:

1) the root managed object requirements (see conformance point 1);
2) the managed object naming and containment requirements (see conformance point 3);
3) the managed object creation requirements (see conformance point 4);
4) the managed object deletion requirements (see conformance point 5);
5) the exceptions requirements (see conformance point 9);
6) the notifications requirements (see conformance point 10).

An OS claiming conformance to the X.780 basic profile (fine-grained) shall support:

1) the X.780 core profile;
2) the standard data types requirements (see conformance point 7);
3) the constants requirements (see conformance point 8).

An OS claiming conformance to the X.780 advanced profile (fine-grained) shall support:

1) the X.780 basic profile;
2) the capabilities requirements (see conformance point 11);
3) the special rules and conventions requirements (see conformance point 15).

An OS claiming conformance to the X.780.1 core profile (coarse-grained) shall support:

1) the X.780 core profile (coarse-grained);
2) the root façade (managing object) requirements (see conformance point 2);
3) the façade object naming requirements (see conformance point 6).

An OS claiming conformance to the X.780.1 basic profile (coarse-grained) shall support:

1) the X.780 basic profile (coarse-grained);
2) the root façade (managing object) requirements (see conformance point 2);
3) the façade object naming requirements (see conformance point 6).

An OS claiming conformance to the X.780.1 advanced profile (coarse-grained) shall support:

1) the X.780.1 basic profile;
2) the capabilities requirements (see conformance point 11);
3) the special rules and conventions requirements (see conformance point 15).

An OS claiming conformance to the X.780.2 core profile (service-oriented) shall support:

1) the root managed object requirements (see conformance point 1);
2) the root façade (managing object) requirements (see conformance point 2);
3) the managed object naming and containment requirements (see conformance point 3);
4) the managed object creation requirements (see conformance point 4);
5) the managed object deletion requirements (see conformance point 5);
6) the façade object naming requirements (see conformance point 6);
7) the exceptions requirements (see conformance point 9);
8) the notifications requirements (see conformance point 10).

An OS claiming conformance to the X.780.2 basic profile (service-oriented) shall support:

1) the X.780.2 core profile;
2) the standard data types requirements (see conformance point 7);
3) the capabilities requirements (see conformance point 11);
4) the MTTM requirements (see conformance point 12);
5) the service-oriented FM requirements (see conformance point 13);
6) the service-oriented PM requirements (see conformance point 14).

An OS claiming conformance to the X.780.2 Advanced Profile (service-oriented) shall support:

1) the X.780.2 basic profile;
2) the constants requirements (see conformance point 8);
3) the special rules and conventions requirements (see conformance point 15);
4) the service-oriented modelling guidelines (see conformance point 16).

The definition of further conformance points and conformance profiles with regard to CORBA IDL repertoire usage and derived modelling of notifications and IDL constructs is for further study.

13.3 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 will not change. Instead, a new IDL file will simply be added. Simply stating the name of an IDL module in a conformance statement, therefore, will not suffice to identify a set of IDL interfaces. The conformance statement must identify a document and month of publication to make sure the right version of IDL is identified.

A standards document claiming compliance to the service-oriented framework may specify a lightweight IDL file structure where modules are not allowed to be split into multiple files (and, therefore, updates of IDL modules always result in updates of IDL files).

```
/**
```

Annex A (Normative)

Service-oriented modelling IDL

```
*/

/* This IDL code is intended to be stored in a file named "itut_x780_2.idl"
located in the search path used by IDL compilers on your system. Most comments
are formatted to be parsed by an IDL-to-HTML converter. */

#ifndef ITUT_X780_2_IDL
#define ITUT_X780_2_IDL


// ********************************
// *                              *
// * itut_x780_2.idl              *
// *                              *
// ********************************


/*

This file defines an extensible, service-oriented NML-EML interface
in CORBA IDL, more generally an OS-OS interface according to Rec. M.3010,
where one OS takes a client/manager role (e.g., an NMS) and the other OS
takes a server/agent role (e.g., an EMS). The OS in a client role is called
managing system, and the OS in a server role is called managed system.

The IDL is organized into the following modules, interfaces,
operations, exceptions, attributes, data types, and constants.
```

module	interface	operation
globaldefs	NamingAttributesIterator_I	next_n()
		getLength()
		destroy()
	ManagedEntityIterator_I	next_n()
		getLength()
		destroy()
common	Common_I	getAllManagedObjectNames()
		getAllManagedObjects()
		getManagedObject()
		setNativeEMSName()
		setUserLabel()
		setOwner()
		getCapabilities()
		setAdditionalInfo()
	ManagedObjectIterator_I	next_n()
		getLength()
		destroy()
transmissionParameters	-	-
notifications	EventIterator _I	next_n()
		getLength()
		destroy()

module	exception, attribute	data type, constant
globaldefs	ProcessingFailureException	Duration_T
		ExceptionType_T
		IterableManagedEntity_T
		IterableManagedEntityList_T
		NameAndStringList_T
		NameAndStringValue_T
		NLSList_T
		NVSList_T

		NamingAttributes_T
		NamingAttributesList_T
		TaggedParameters_T
		TaggedParameterList_T
		Time_T
common	-	Capability_T
		CapabilityList_T
		Common_T
		ManagedObjectList_T
transmissionParameters	-	LayerRate_T
		LayerRateList_T
		LayeredParameters_T
		LayeredParameterList_T
notifications	-	NT_OBJECT_CREATION
		NT_OBJECT_DELETION
		NT_ATTRIBUTE_VALUE_CHANGE
		NT_STATE_CHANGE
		NT_FILE_TRANSFER_STATUS
		NT_BACKUP_STATUS
		NT_HEARTBEAT
		NT_ALARM
		NT_TCA
		Event_T
		EventList_T
		EventType_T
		EventName_T
		NotificationId_T
		ObjectName_T
		ObjectType_T
		ObjectTypeQualifier_T
		EmsTime_T
		NeTime_T
		EdgePointRelated_T
		OCN_T
		ODN_T
		NameAndAnyValue_T
		NVList_T
		AVC_T
		SCN_T
		FileTransferStatus_T
		FTS_T
		Current_OperationStatus_T
		BackupStatus_T
		BSE_T
		HBE_T
		NativeEMSName_T
		PerceivedSeverity_T
		PerceivedSeverityList_T
		ServiceAffecting_T
		AffectedTPList_T
		ProbableCause_T
		ProbableCauseList_T
		ProbableCauseQualifier_T
		NativeProbableCause_T
		AssignedSeverity_T
		AlarmSeverityAssignment_T
		AlarmSeverityAssignmentList_T
		AdditionalText_T
		AdditionalInfo_T
		RcaiIndicator_T
		IsClearable_T
		AcknowledgeIndication_T
		X733_EventType_T
		X733_EventTime_T
		SpecificProblem_T
		SpecificProblemList_T
		X733_BackupStatus_T
		X733_BackupObject_T
		X733_TrendIndication_T
		NotifIDList_T
		CorrelatedNotifications_T

```
                    |                              | CorrelatedNotificationList_T
                    |                              | X733_MonitoredAttributes_T
                    |                              | ProposedRepairAction_T
                    |                              | ProposedRepairActionList_T
                    |                              | X733_StateChange_T
                    |                              | X733_AdditionalInfo_T
                    |                              | AlarmId_T
                    |                              | ALM_T
                    |                              | PMParameterName_T
                    |                              | PMLocation_T
                    |                              | Granularity_T
                    |                              | PMThresholdType_T
                    |                              | TCAId_T
                    |                              | Value_T
                    |                              | Unit_T
                    |                              | TCA_T
                    |                              | AlarmTypeQualifier_T
                    |                              | AlarmOrTCAIdentifier_T
                    |                              | AlarmAndTCAIDList_T

*/

//Include list
#include "OMGidl/CosNotification.idl"

#pragma prefix "itu.int"

/**
```

A.1 Module `globaldefs`

```
*/

  /**
   * <p>This module defines common types and other IDL constructs
   * used by all CORBA modules of the NML-EML interface.
   * It is intended to be used as a common repository for definitions
   * that need to be exported across modules.</p>
   **/

module globaldefs
{
  /**
   * <p>The NameAndStringValue_T structure is provided here as a replacement of
   * the NVList construct defined by the OMG. In consideration for performance
   * and the cost associated with the marshalling of the any type, it is decided
   * to use the type string for the value field instead of the any type.
   * When used for name components the structure is equivalent to the
   * CosNaming::NameComponent structure of the OMG Naming Service.</p>
   **/
   struct NameAndStringValue_T
   {
      string name;
      string value;
   };

  /**
   * <p>A list of (name=string, value=string) tuples, i.e.,
   * name/value pairs with string values (NVS pairs).</p>
   *
   * <p>For example, the additional info parameters of any service-oriented
   * managed object (see module common) and the transmission parameters of a
   * Termination Point (see module transmissionParameters) use this structure.
   * When NVS pairs are used to specify parameters for usage in attributes of
   * managed objects, a standardized naming scheme is adopted between the NMS
   * and the EMS to identify the name and the value fields of NVS pairs.</p>
   **/
   typedef sequence<NameAndStringValue_T> NVSList_T;

  /**
```

```
 * <p>The NamingAttributes_T structure is used to define identifiers
 * (names) for managed entities that are not instantiated as first class
 * CORBA objects and thus do not have a CORBA object identifier (IOR).
 * It represents "the hierarchical name structure" of a second class
 * "non-CORBA" object (managed object); it is an attribute of the
 * managed object that contains its full distinguished name (FDN). The
 * structure of the name is hierarchical and reflects the containment
 * relationship between managed objects in a simple way. It consists of
 * an ordered list (sequence) of components where the preceding component
 * is a containing entity for the following component (e.g., a network
 * element contains a termination point).</p>
 *
 * <p>Refer to clause 10.3.2 "Structure of naming attribute of managed
 * objects" of Rec. X.780.2 for further details.</p>
 **/
 typedef NVSList_T NamingAttributes_T;

/**
 * <p>A list of NamingAttributes_T. It is a list of lists.</p>
 **/
 typedef sequence<NamingAttributes_T> NamingAttributesList_T;

/**
 * <p>The NameAndStringList_T structure is provided here as a
 * standard means to model multi-valued parameters with string values,
 * i.e., parameters whose values are lists of strings (NVL pairs).</p>
 **/
 struct NameAndStringList_T
 {
    string name;
    sequence<string> value;
 };

/**
 * <p>A list of (name=string, value=sequence&lt;string&gt;) tuples,
 * i.e., name/value pairs with list of string values (NVL pairs).</p>
 **/
 typedef sequence<NameAndStringList_T> NLSList_T;

/**
 * <p> The TaggedParameters_T structure relates a parameter list
 * (i.e., list of NVS pairs) with a parameter tag.</p>
 **/
 struct TaggedParameters_T
 {
    unsigned long parameterTag;
    NVSList_T taggedParams;
 };

/**
 * <p>A list of tagged parameters.</p>
 **/
 typedef sequence<TaggedParameters_T> TaggedParameterList_T;

/**
 * <p>Time_T is a string holding a generalized time string as defined in
 * clause 42 of ITU-T Rec. X.680 "Information technology - Abstract Syntax
 * Notation One (ASN.1): Specification of basic notation". This string
 * represents an absolute time value (calender date and time of day).</p>
 *
 * <p>The format is "yyyyMMddhhmmss.s[Z|{+|-}HHMm]" where:<br>
 * <pre>      yyyy   "0000".."9999"  year
 *       MM     "01".."12"      month
 *       dd     "01".."31"      day
 *       hh     "00".."23"      hour
 *       mm     "00".."59"      minute
 *       ss     "00".."59"      second
 *       .s     ".0"..".9"      tenth of second (set to ".0" if EMS
                             or NE cannot support this granularity)
 *       Z      "Z"             indicates UTC (rather than local time)
 *       {+|-}  "+" or "-"      delta from UTC
```

```
 *       HH     "00".."23"      time zone difference in hours
 *       Mm     "00".."59"      time zone difference in minutes</pre></p>
 *
 * <p>For example,<br> "19851106210627.3Z" would be 6 minutes, 27.3 seconds
 * after 9 p.m. on November 6th, 1985 and indicating UTC time,<br>
 * "19851106210627.3" would be local time,<br>
 * "19851106210627.3+0500" would be local time specifying a +5 hour time
 * zone difference from UTC,<br> "19851106210627.3-0530" would be local
 * time specifying a -5.5 hour time zone difference from UTC.</p>
 **/
 typedef string Time_T;

/**
 * <p>Duration_T represents a relative time value (duration) with unit
 * 100 nanoseconds. For example, 18 * 10**9 represents 30 minutes.</p>
 **/
 typedef unsigned long long Duration_T;

/**
 * <p><b>Exception Type Definitions</b></p>
 *
 * <p>As per CORBA policies agreement, only one exception object
 * is defined to capture all of the possible exception types
 * defined in the <b>ProcessingFailureException</b> exception.</p>
 *
 * <p><ul>
 * <li> EXCPT_NOT_IMPLEMENTED
 * <br>If some IDL operations are optional or not implemented,
 * then this value may be used for this purpose. If the operation
 * itself is not supported, then errorReason shall be an empty string.
 * If this exception is raised because of the values of specific
 * parameters (i.e., NV pairs), then the names of these parameters
 * shall be supplied in errorReason (separated by commas), unless
 * otherwise specified in the operation description.</li>
 *
 * <li> EXCPT_INTERNAL_ERROR
 * <br>To indicate an EMS internal error. Applies to all operations.
 * The errorReason can be used to provide further clarification.</li>
 *
 * <li> EXCPT_INVALID_INPUT
 * <br>If the format of a parameter is incorrect (e.g., if a name
 * which is a 3-level NamingAttributes_T is passed as a single level
 * name), then this type will be used. Also if a parameter is out of
 * range, this type will be used. The reason field will be filled
 * with the parameter that was incorrect.</li>
 *
 * <li> EXCPT_OBJECT_IN_USE
 * <br>To indicate that an object is already in use.</li>
 *
 * <li> EXCPT_ENTITY_NOT_FOUND
 * <br>In general, if the NMS supplies an object name as a parameter
 * to an operation and the EMS cannot find the object with the given
 * name, then an exception of this type is returned. The errorReason
 * field in the exception will be filled with the name that was passed
 * in as a parameter.</li>
 *
 * <li> EXCPT_NOT_IN_VALID_STATE
 * <br>Used if the client tries to delete an object that is
 * in a state that prevents its deletion.</li>
 *
 * <li> EXCPT_UNABLE_TO_COMPLY
 * <br>This value is used as a generic value when a server (EMS)
 * cannot respond to the request and no other exception provides
 * the appropriate clarification. The errorReason string is used
 * to provide further information.</li>
 *
 * <li> EXCPT_NE_COMM_LOSS
 * <br>This value is used as a generic value when a server (EMS) cannot
 * communicate with the NE (Network Element) to which the operation
 * should apply and that prevents the successful completion of
 * the operation. All operations that involve communication with
```

```
 * the NE may throw this particular exception type. Note that NE
 * here applies to any device to which the server (EMS) will have
 * to communicate to fulfill the operation.</li>
 *
 * <li> EXCPT_CAPACITY_EXCEEDED
 * <br>Raised when an operation will result in resources being created
 * or activated beyond the capacity supported by the NE/EMS.</li>
 *
 * <li> EXCPT_ACCESS_DENIED
 * <br>Raised when an operation results in a security violation.</li>
 *
 * <li> EXCPT_TOO_MANY_OPEN_ITERATORS
 * <br>Raised when the EMS exceeds its internal limit of the number
 * of iterators it can support.</li>
 *
 * <li> EXCPT_USERLABEL_IN_USE
 * <br>Raised when the userLabel uniqueness constraint cannot be met.</li>
 * </ul></p>
 *
 * <p>This standard framework exception list may be extended with more
 * focussed exceptions that relate to the problem space to which the
 * respective interface using the framework is applied. For example,
 * an exception EXCPT_UNSUPPORTED_ROUTING_CONSTRAINTS may be specified
 * where the framework has been used to develop an interface that
 * relates to the problem of routing across a network. Standard OMG
 * exceptions may also occur, e.g., NO_IMPLEMENT if an operation
 * implementation is unavailable, but they should be documented
 * with IDL comments similar to the standard ITU-T exceptions.</p>
 *
 * <p>Refer to clause 10.5.2.4 of Rec. X.780.2 for the general
 * procedure, and to notifications::ObjectType_T as an example,
 * of how to extend enum types. ExceptionType_T becomes extensible
 * when EXCPT_INTERNAL_ERROR is considered to be the escape value
 * and errorReason is used to convey new exception values
 * according to the convention
 * <code>"EXCPT_&lt;new exception type&gt;;&lt;errorReason&gt;"</code>.</p>
 **/
 enum ExceptionType_T
 {
    EXCPT_NOT_IMPLEMENTED,
    EXCPT_INTERNAL_ERROR,
    EXCPT_INVALID_INPUT,
    EXCPT_OBJECT_IN_USE,
    EXCPT_ENTITY_NOT_FOUND,
    EXCPT_NOT_IN_VALID_STATE,
    EXCPT_UNABLE_TO_COMPLY,
    EXCPT_NE_COMM_LOSS,
    EXCPT_CAPACITY_EXCEEDED,
    EXCPT_ACCESS_DENIED,
    EXCPT_TOO_MANY_OPEN_ITERATORS,
    EXCPT_USERLABEL_IN_USE
 };

/**
 * <p>A coarse grain approach is adopted for capturing exceptions
 * as well. This has the advantage of making the catching of exceptions
 * fairly generic. Since CORBA does not allow as in the Java language
 * to subclass exceptions, it is recommended to reduce the number
 * of exceptions a client may catch as far as reasonably possible.
 * On the down side, a client may need to write explicit code when
 * an exception is thrown by the server (i.e., a client may need to
 * have a switch statement on the exceptionType attribute).</p>
 *
 * ExceptionType_T <b>exceptionType</b>:
 * See description of ExceptionType_T.<br>
 * string <b>errorReason</b>: A string indicating further details about
 * the exception. It is a free format string filled by the EMS.<br>
 **/
 exception ProcessingFailureException
 {
    ExceptionType_T exceptionType;
```

```
      string errorReason;
   };

   /**
    * <p>In order to allow the NMS to deal with retrieval of a large
    * number of names, iterators are used for bulk retrieval of
    * managed object names.</p>
    *
    * <p>Refer to clause 9.4 "Iterator interfaces" of Rec. X.780.2
    * for information on how iterators are used in this interface.</p>
    **/
    interface NamingAttributesIterator_I
    {
    boolean next_n(
             in unsigned long how_many,
             out NamingAttributesList_T nameList)
            raises(globaldefs::ProcessingFailureException);

    unsigned long getLength()
             raises(globaldefs::ProcessingFailureException);

    void destroy()
         raises(globaldefs::ProcessingFailureException);

    };  // end of interface

   /**
    * <p>The IterableManagedEntity_T struct represents a generic managed
    * entity type which is iterable (e.g., a managed object type).</p>
    **/

    struct IterableManagedEntity_T
    {
       // add any managed entity type-specific attributes here
       globaldefs::NVSList_T additionalInfo;
    };

   /**
    * <p>The IterableManagedEntityList_T structure represents a generic
    * iterator batch consisting of generic iterable entities.</p>
    **/
    typedef sequence<IterableManagedEntity_T> IterableManagedEntityList_T;

   /**
    * <p>In order to allow the NMS to deal with retrieval of a large
    * number of entities, iterators are used for bulk retrieval of
    * managed entities.</p>
    *
    * <p>Refer to clause 9.4 "Iterator interfaces" of Rec. X.780.2
    * for information on how iterators are used in this interface.</p>
    *
    * <p>The ManagedEntityIterator_I is a generic iterator.</p>
    **/
    interface ManagedEntityIterator_I
    {
    boolean next_n(
             in unsigned long how_many,
             out IterableManagedEntityList_T managedEntityList)
            raises(globaldefs::ProcessingFailureException);

    unsigned long getLength()
                  raises(globaldefs::ProcessingFailureException);

    void destroy()
         raises(globaldefs::ProcessingFailureException);

    };  // end of interface

};  // end of module
```

```
/**
```

A.2 Module common

```
*/

  /**
   * <p>This module contains the definitions of the Common_T structure
   * and the Common_I interface of the NML-EML interface.</p>
   **/

module common
{
  /**
   * <p>A Capability_T value is used to identify a functionality supported by
   * the EMS across the NML-EML interface. It is a name/value pair, in which
   * the name represents the feature/capability name and the value represents
   * the support or non-support of the specified feature/capability.</p>
   *
   * <p>The EMS capabilities include individual IDL operation support.
   * The feature/capability name part is used to identify an IDL
   * operation using the following convention (scoped operation name):
   * "<i>module_name</i>::<i>interface_name</i>::<i>operation_name</i>".</p>
   *
   * <p>Applications of this framework may define a number of other
   * specifiable capabilities in addition to the operation-oriented
   * capabilities, e.g., regarding connectivity management
   * or fault management.</p>
   *
   * <p>The currently defined values are as follows:<br>
   * <ul>
   * <li><b>"Supported"</b>: The specified feature/capability is
   * fully or partially supported across the NML-EML interface,
   * where an operation is partially supported if not all values
   * of all parameters are supported.</li>
   * <li><b>"Unsupported"</b>: The specified feature/capability is
   * not supported at all across the NML-EML interface.</li>
   * </ul></p>
   *
   * <p> Other capabilities may be added with the approval of an amendment
   * to Rec. X.780.2, or through bilateral agreements. </p>
   **/
   typedef globaldefs::NameAndStringValue_T Capability_T;

  /**
   * <p> Set of Capability_T. Used to represent the full set of
   * capabilities of a service-oriented façade. Any capability
   * that is not listed is considered unsupported. </p>
   */
   typedef sequence<Capability_T> CapabilityList_T;

  /**
   * <p>The struct Common_T defines the mandatory common
   * attributes of each managed object class (MOC). This is
   * often a "virtual" struct, meant to be renamed and extended
   * whenever a specific MOC should be defined. It could also
   * be used for the modelling of managed objects all of whose
   * attributes are additional info parameters. Such a managed
   * object without MOC-specific attributes (i.e., of
   * IDL type Common_T) is called generic.</p>
   *
   * <p>The mandatory common attributes of managed objects are:<br>
   * - globaldefs::NamingAttributes_T <b>name</b>:<br>
   *   The full distinguished name (FDN) of the managed object.<br>
   * - string                         <b>userLabel</b>:<br>
   *   A friendly name that the operator wants to place for the
   *   managed object and get it used subsequently by the NMS.<br>
   * - string                         <b>nativeEMSName</b>:<br>
   *   A representation how the managed object is referred to
   *   on EMS displays; its aim is to provide a "nomenclature
```

```
 *   bridge" to aid relating information presented on NMS
 *   displays to EMS displays (via GUI cut through).<br>
 * - string                          <b>owner</b>:<br>
 *   The owner of the managed object as provisioned by the NMS,
 *   e.g., in the sense of a "used by" relationship.<br>
 * - globaldefs::NVSList_T           <b>additionalInfo</b>:<br>
 *   Additional information which is not explicitly modelled
 *   at the NML-EML interface but needs to be communicated
 *   from the EMS to the NMS or vice versa. Such information
 *   may be standardized or vendor-specific.<br>
 * </p>
 **/
 struct Common_T
 {
    globaldefs::NamingAttributes_T name;
    string userLabel;
    string nativeEMSName;
    string owner;
    // add any MOC-specific attributes here
    globaldefs::NVSList_T additionalInfo;
 };

/**
 * <p> List of Common_T. Used to represent a batch of generic managed
 * objects (i.e., which have only common attributes).</p>
 */
 typedef sequence<Common_T> ManagedObjectList_T;

/**
 * <p>In order to allow the NMS to deal with retrieval of a large number
 * of objects, iterators are used for bulk retrieval of managed objects.</p>
 *
 * <p>Refer to clause 9.4 "Iterator interfaces" of Rec. X.780.2
 * for information on how iterators are used in this interface.</p>
 **/
 interface ManagedObjectIterator_I
 {
 boolean next_n(
          in unsigned long how_many,
          out ManagedObjectList_T moList)
         raises(globaldefs::ProcessingFailureException);

 unsigned long getLength()
          raises(globaldefs::ProcessingFailureException);

 void destroy()
      raises(globaldefs::ProcessingFailureException);
 };

/**
 * <p>The interface Common_I is a set of services and utilities
 * that is inherited by every service-oriented façade interface.<p>
 *
 * <p>It allows to retrieve the capabilities of the respective
 * concrete façade and to set the (settable) common attributes
 * of any managed object that is being managed by the façade.</p>
 **/
 interface Common_I
 {
/**
 * <p>This allows an NMS to request the names of all of the generic
 * Managed Objects that are under the control of this Common_I.</p>
 *
 * <p>In order to allow the NMS to deal with a large number of objects,
 * this operation uses an iterator (see 9.4/X.780.2).</p>
 *
 * @parm unsigned long <b>how_many</b>: Maximum number of MO names
 * to return in the first batch.
 *
 * @parm globaldefs::NamingAttributesList_T <b>nameList</b>:
 * First batch of MO names.
```

```
 *
 * @parm globaldefs::NamingAttributesListIterator_I <b>nameIt</b>:
 * Iterator to retrieve the remaining MO names.
 *
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * EXCPT_TOO_MANY_OPEN_ITERATORS - Raised when the maximum number of
 *  iterators that the EMS can support has been reached<br>
 * </dir>
 **/
 void getAllManagedObjectNames(
       in unsigned long how_many,
       out globaldefs::NamingAttributesList_T nameList,
       out globaldefs::NamingAttributesIterator_I nameIt)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>This operation has exactly the same behaviour as
 * getAllManagedObjectNames, but instead of returning
 * the names of the objects, this operation returns
 * their entire object structures.</p>
 *
 * <p>This allows an NMS to request details of all of the generic
 * Managed Objects that are under the control of this Common_I.</p>
 *
 * <p>In order to allow the NMS to deal with a large number of objects,
 * this operation uses an iterator (see 9.4/X.780.2).</p>
 *
 * @parm unsigned long <b>how_many</b>: Maximum number of MOs
 * to report in the first batch.
 *
 * @parm ManagedObjectList_T <b>moList</b>: First batch of MOs.
 *
 * @parm ManagedObjectIterator_I <b>moIt</b>:
 * Iterator to retrieve the remaining MOs.
 *
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * EXCPT_TOO_MANY_OPEN_ITERATORS - Raised when the maximum number of
 *  iterators that the EMS can support has been reached<br>
 * </dir>
 **/
 void getAllManagedObjects(
       in unsigned long how_many,
       out ManagedObjectList_T moList,
       out ManagedObjectIterator_I moIt)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>This service returns the Managed Object for the given
 * generic managed object name.</p>
 *
 * @parm globaldefs::NamingAttributes <b>objectName</b>:
 * Name of the MO to retrieve.
 *
 * @parm Common_T <b>mo</b>: The retrieved MO.
 *
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * EXCPT_INVALID_INPUT - Raised when objectName does not reference a
 *  generic managed object<br>
 * EXCPT_ENTITY_NOT_FOUND - Raised when objectName references an MO
 *  object that does not exist<br>
 * EXCPT_NE_COMM_LOSS - Raised when communications to the managed
 *  element containing or hosting objectName is lost<br>
 * </dir>
 **/
 void getManagedObject(
```

```
       in globaldefs::NamingAttributes_T objectName,
       out Common_T mo)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>The nativeEMSName is owned by the EMS. It represents how an EMS user
 * addresses an object on the EMS GUI. The EMS may or may not support
 * changing this value through this service.</p>
 *
 * <p>When an object is created by the EMS, the EMS selects the nativeEMSName
 * for the object.</p>
 *
 * <p>When an object is created by an NMS, the NMS specifies the userLabel
 * for the object. If the EMS supports setting of native EMS names, the
 * nativeEMSName should be set to the same value as the userLabel. If the
 * EMS does not support setting of native EMS names, or if the nativeEMSName
 * has constraints that the userLabel does not satisfy, the EMS selects
 * the nativeEMSName for the object.</p>
 *
 * <p>After an object has been created, the nativeEMSName may be changed by
 * the NMS, if the EMS supports this functionality, using this operation.</p>
 *
 * @parm globaldefs::NamingAttributes_T <b>objectName</b>: Name of the object
 * for which to change the native EMS name.<
 *
 * @parm string <b>nativeEMSName</b>: New native EMS name
 * to assign to the object.
 *
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_NOT_IMPLEMENTED - Raised if the EMS does not support this service<br>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * EXCPT_INVALID_INPUT - Raised when objectName is incorrectly formed<br>
 * EXCPT_ENTITY_NOT_FOUND - Raised when objectName references an object
 *  which does not exist<br>
 * EXCPT_UNABLE_TO_COMPLY - Raised when the nativeEMSName cannot be set
 *  for the specified object<br>
 * EXCPT_NE_COMM_LOSS - Raised when communications to the managed element
 *  containing or hosting objectName is lost
 * </dir>
 **/
 void setNativeEMSName(
       in globaldefs::NamingAttributes_T objectName,
       in string nativeEMSName)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>The userLabel is owned by the NMSes. It is a string assigned by
 * an NMS to an object. A difference between the userLabel and the
 * NamingAttributes_T name is that the userLabel is an attribute of
 * the objects that may be changed by an NMS through this service.</p>
 *
 * <p>When an object is created by an NMS, the NMS specifies the userLabel
 * for the object.</p>
 *
 * <p>When an object is created by the EMS, the EMS sets the userLabel
 * to the nativeEMSName.</p>
 *
 * <p>Once an object is created, the userLabel may only be changed by
 * an NMS through this operation.</p>
 *
 * @parm globaldefs::NamingAttributes_T <b>objectName</b>: Name of the object
 * for which to change the user label.
 *
 * @parm string <b>userLabel</b>: New user label to assign to the object.
 *
 * @parm boolean <b>enforceUniqueness</b>: Specifies whether or not
 * userLabel should be checked for uniqueness amongst objects of
 * the same class within the EMS. If TRUE, then the operation will fail
 * if userLabel is already in use.
 *
```

```
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_NOT_IMPLEMENTED - Raised if the EMS does not support this service<br>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * EXCPT_INVALID_INPUT - Raised when objectName is incorrectly formed<br>
 * EXCPT_ENTITY_NOT_FOUND - Raised when objectName references an object
 *  which does not exist<br>
 * EXCPT_UNABLE_TO_COMPLY - Raised when the userLabel cannot be set
 *  for the specified object<br>
 * EXCPT_NE_COMM_LOSS - Raised when communications to the managed element
 *  containing or hosting objectName is lost<br>
 * EXCPT_USERLABEL_IN_USE - Raised when the userLabel uniqueness constraint
 *  is not met<br>
 * </dir>
 **/
 void setUserLabel(
       in globaldefs::NamingAttributes_T objectName,
       in string userLabel,
       in boolean enforceUniqueness)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>This service sets the owner attribute of the specified object.</p>
 *
 * @parm globaldefs::NamingAttributes_T <b>objectName</b>: Name of the object
 * for which to change the owner.
 *
 * @parm string <b>owner</b>: New owner to assign to the object.
 *
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_NOT_IMPLEMENTED - Raised if the EMS does not support this service<br>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * EXCPT_INVALID_INPUT - Raised when objectName is incorrectly formed<br>
 * EXCPT_ENTITY_NOT_FOUND - Raised when objectName references an object
 *  that does not exist<br>
 * EXCPT_UNABLE_TO_COMPLY - Raised when the owner cannot be set
 *  for the specified object<br>
 * EXCPT_NE_COMM_LOSS - Raised when communications to the managed element
 *  containing or hosting objectName is lost<br>
 * </dir>
 **/
 void setOwner(
       in globaldefs::NamingAttributes_T objectName,
       in string owner)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>This service retrieves the capabilities of the manager.
 * All non-specified capabilities are assumed to be unsupported.</p>
 *
 * @parm CapabilityList_T <b>capabilities</b>: The retrieved capabilities.
 *
 * @raises globaldefs::ProcessingFailureException<dir>
 * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
 *  failure<br>
 * </dir>
 **/
 void getCapabilities(
       out CapabilityList_T capabilities)
      raises(globaldefs::ProcessingFailureException);

/**
 * <p>This service sets the additional info attribute of the object identified
 * by objectName. This operation should be used to set both vendor-specific
 * attributes as well as the attributes that are formally defined in
 * standardized interface specifications that extend this framework.</p>
 *
 * <p>As an input only the list of parameters to be changed, removed,
 * or added shall be provided. If an entry is to be removed, "-" shall
 * be specified as a value. If a parameter is specified that is currently
```

```
   * not part of the additionalInfo attribute of the specified object
   * that parameter is added by the EMS with the specified value.
   * The EMS may reject removal and addition requests, however.</p>
   *
   * <p>The operation is best effort except where specified otherwise
   * for a particular parameter where this parameter is defined (i.e.,
   * in a standardized or vendor-specific NML-EML interface specification
   * that extends this framework). The output specifies the values which
   * were actually set.</p>
   *
   * @parm globaldefs::NamingAttributes_T <b>objectName</b>: The managed object
   * whose additional info parameters are intended to get modified.
   *
   * @parm globaldefs::NVSList_T <b>additionalInfo</b>: List of parameters
   * to be changed, added, or removed (input), updated to provide the
   * actually changed or added parameters (output).
   *
   * @raises globaldefs::ProcessingFailureException<dir>
   * EXCPT_NOT_IMPLEMENTED - Raised if the EMS does not support this service<br>
   * EXCPT_INTERNAL_ERROR - Raised in case of non-specific EMS internal
   *  failure<br>
   * EXCPT_INVALID_INPUT - Raised when objectName is incorrectly formed,
   *  raised when an input parameter is syntactical incorrect, and raised
   *  when a parameter is identified as settable only by using a "specific
   *  operation" defined in a standardized NML-EML interface specification
   *  that extends this framework<br>
   * EXCPT_ENTITY_NOT_FOUND - Raised when objectName references an object
   *  that does not exist<br>
   * EXCPT_NE_COMM_LOSS - Raised when communications to the managed element
   *  containing or hosting objectName is lost<br>
   * EXCPT_UNABLE_TO_COMPLY - Raised when the EMS is unable to execute the
   *  request because at least one of the parameters although valid can not
   *  be set and that parameter is identified as "not best effort" in a
   *  standardized interface specification that extends this framework<br>
   * </dir>
   **/
   void setAdditionalInfo(
         in globaldefs::NamingAttributes_T objectName,
         inout globaldefs::NVSList_T additionalInfo)
        raises(globaldefs::ProcessingFailureException);

   };  // end of interface

};  // end of module


/**
```

A.3 Module `transmissionParameters`

```
*/

  /**
   * <p>This module contains the generic definitions of Layer rates, which
   * identify telecom transmission technologies, and of single-layer and
   * multi-layer transmission parameters. These definitions may be used
   * for defining attributes of transmission-related managed object classes
   * of the NML-EML interface (such as termination points and subnetwork
   * connections), which specify these transmission characteristics
   * for the application-specific MOC to be defined.</p>
   *
   * <p>Whilst the framework can be applied for the modelling of any OS-OS
   * interface in a service-oriented way, a particularly important application
   * is the case of an NMS-EMS interface where the EMS is in contact with a
   * physical telecom network that may encompass a number of transport, access
   * and aggregation technologies. The framework therefore provides a generic
   * and flexible means to model any of these network technologies down to any
   * required detail and in the context of protocol stacks and multiplexing
   * hierarchies. This approach is called <b>multi-technology capability</b>
   * for short and is based on the layered transport network functional
```

```
   * architecture of Recommendations G.805 and G.809, and the corresponding
   * management abstractions provided in Recommendation G.852.2.</p>
   **/

module transmissionParameters
{
  /**
   * <p>The LayerRate_T value is used to identify the characteristic or
   * adapted information of a telecom transmission layer according to Rec.
   * G.805 (and Rec. G.809). Managed object class definitions for entities,
   * whose features depend on such layer(s), may include an attribute of
   * this type, or a list of this type, to identify the supported or
   * provisioned Layer rate(s). For example:<br>
   * - the list of layers at which an NE may establish cross-connects;<br>
   * - the (connectable) layer of a TTP or CTP
   *   (as defined by Rec. G.852.2 and Rec. M.3100);<br>
   * - the Layer rate of a subnetwork connection.</p>
   *
   * <p>The type of the LayerRate_T has been made a <code>short</code>
   * rather than an (extensible) <code>enum</code> to allow new rates to
   * be added without changing the IDL interface. This generic framework
   * for OS-OS interface design does not encode specific Layer rates,
   * however, but leaves this task to domain-specific applications and
   * extensions. In the interest of interoperability users of the
   * service-oriented CORBA framework must cooperate in reaching
   * normative usage of standardized Layer rates (and associated
   * transmission parameters) (see 8.4/X.780.2).</p>
   *
   * <p>The value 0 is not used, and the value 1 is identified as
   * LR_Not_Applicable and used in cases where a Layer rate attribute is
   * not applicable. A Layer rate in the range 0 to 9999 is considered
   * as a <b>standardized Layer rate</b>, while the range of numbers 10.000
   * and above is allocated for vendor-specific usage and a Layer rate in
   * this range is considered as a <b>proprietary Layer rate</b>.</p>
   **/
   typedef short LayerRate_T;

  /**
   * <p>Set of LayerRate_T. The LayerRateList_T struct represents
   * multiple layers according to client/server relationships between
   * adjacent layers as specified in Recs. G.805 and G.809.</p>
   **/
   typedef sequence<LayerRate_T> LayerRateList_T;

  /**
   * <p>The LayeredParameters_T struct includes the Layer rate with
   * the applicable list of transmissions parameters. It may be used
   * to define attributes of single-layered managed entities.</p>
   *
   * <p>LayerRate_T <b>layer</b>:
   * Represents the layer to which the parameters apply.<br>
   *
   * globaldefs::NVSList_T <b>transmissionParams</b>:
   * Name/value pair list for the supported Layer rates as specified in
   * applications and extensions of this framework (see 8.4/X.780.2).</p>
   **/
   struct LayeredParameters_T
   {
      LayerRate_T layer;
      globaldefs::NVSList_T transmissionParams;
   };

  /**
   * <p>Set of LayeredParameters_T. It may be used for defining
   * attributes of multi-layered managed entities.</p>
   **/
   typedef sequence<LayeredParameters_T> LayeredParameterList_T;

};  // end of module
```

```
/**
```

A.4 Module `notifications`

```
*/

  /**
   * <p>This module contains the definition of the notification portions
   * of the NML-EML interface. Refer to clause 8.3 of Rec. Q.816.2 for
   * details on how this interface uses the OMG Notification Service.</p>
   *
   * <p>The standard notification types supported by this framework are
   * listed as constants. For each standard notification type a struct
   * is defined that groups all attributes of this type.</p>
   *
   * <p>An interface specification that extends this framework may define
   * additional notification/event types according to the rule
   * <code>const string NT_&amp;lt;notification type&amp;gt; =
   * "NT_&amp;lt;notification type&amp;gt;";</code>.
   * It may also refine the standard notification types with additional
   * field attributes. In this context it is also allowed to refactor
   * definitions from this module to other CORBA modules (e.g.,
   * when defining PM details in a separate PM module).</p>
   *
   * <p>The NMS should ignore notifications with unrecognised event type.
   * This is required to provide forward compatibility.</p>
   **/

module notifications
{
  /** Object Creation Notification (OCN) */
   const string NT_OBJECT_CREATION = "NT_OBJECT_CREATION";

  /** Object Deletion Notification (ODN) */
   const string NT_OBJECT_DELETION = "NT_OBJECT_DELETION";

  /** Attribute Value Change notification (AVC) */
   const string NT_ATTRIBUTE_VALUE_CHANGE = "NT_ATTRIBUTE_VALUE_CHANGE";

  /** State Change Notification (SCN) */
   const string NT_STATE_CHANGE = "NT_STATE_CHANGE";

  /** File Transfer Status notification (FTS) */
   const string NT_FILE_TRANSFER_STATUS = "NT_FILE_TRANSFER_STATUS";

  /** Backup Status Event notification (BSE) */
   const string NT_BACKUP_STATUS = "NT_BACKUP_STATUS";

  /** Heartbeat Event notification (HBE) */
   const string NT_HEARTBEAT = "NT_HEARTBEAT";

  /** Alarm notification (ALM) */
   const string NT_ALARM = "NT_ALARM";

  /** Threshold Crossing Alert notification (TCA) */
   const string NT_TCA = "NT_TCA";

  /**
   * <p>The structure of any notification is an <b>OMG Structured Event</b>.</p>
   *
   * <p>This module defines the attributes of each standard notification
   * type through an appropriate struct in order to fix all standard
   * details in a compilable way (OCN_T, ODN_T, AVC_T, SCN_T, FTS_T,
   * BSE_T, HBE_T, ALM_T, TCA_T). All definitions are specialisations
   * of Structured Events. They shall be used for all transport mechanisms
   * such as push/pull model and structured/typed events (see 6.5/Q.816.2,
   * 8.3/Q.816.2, and conformance point 4) of 10.2.1/Q.816.2).</p>
   *
   * <p>In particular, when name/value pairs are used to define the fields of
   * the filterable event body of Structured Events, the field names shall be
```

```
 * the names of the record members of the struct defining the notification
 * type and the actual field types shall be the types of these record members,
 * except possibly when such a type is itself a struct in which case the
 * record members of the nested struct shall be considered directly (e.g.,
 * in case of AlarmId_T for ALM_T).</p>
 **/
 typedef CosNotification::StructuredEvent Event_T;

/**
 * <p>A list of OMG Structured Events. This is the <b>OMG Event Batch</b>.</p>
 **/
 typedef sequence<CosNotification::StructuredEvent> EventList_T;

/**
 * <p>The <b>eventType</b> of a notification is part of the fixed
 * event header. It consists of the <b>domain_name</b> and the
 * <b>type_name</b> (see 8.3/Q.816.2). The values of domain_name
 * are defined by the interface specification that extends this
 * framework. The values of type_name are the constants
 * <code>NT_&lt;notification type&gt;</code> (standard,
 * as defined above, or implementation-specific) that specify
 * the notification types used at the interface.</p>
 **/
 typedef CosNotification::_EventType EventType_T;

/**
 * <p>To comply with the OMG definition of the fixed event header of
 * structured events, each notification has a record member <b>eventName</b>
 * of type string, which is meant to name a specific instance of an
 * OMG structured event where the semantics of the names is defined
 * by the end users of the Notification Service (i.e., the managed and
 * managing systems). For reasons of interoperability it is recommended
 * to ignore this field respectively to have it always set to "".</p>
 **/
 typedef string EventName_T;

/**
 * <p>Each notification has a record member <b>notificationId</b> of type
 * string, which maps to a field of the filterable event body of structured
 * events and contains an implementation-defined free format notification
 * identifier whose semantics is specified in 8.1.2.8/X.733. The
 * uniqueness and the sequence of the notification ID are not guaranteed.
 * When correlation of notifications according to Rec. X.733 is supported
 * (see below), notification IDs must be chosen to be unique across all
 * notifications from a particular managed object throughout the time
 * the considered correlations are significant. Therefore notification IDs
 * may be reused only when no requirements exist for correlating prior
 * notifications and should be chosen to ensure uniqueness over as long
 * a time as is feasible for the managed system (EMS).</p>
 **/
 typedef string NotificationId_T;

/**
 * <p>Most notification types have a record member <b>objectName</b> of
 * type globaldefs::NamingAttributes_T, which uniquely identifies the
 * managed object that reports the event (i.e., the event source).</p>
 **/
 typedef globaldefs::NamingAttributes_T ObjectName_T;

/**
 * <p>ObjectType_T is the type of the filterable field attribute
 * <b>objectType</b>, which belongs to all notifications that are reported
 * by a managed object. Standardized interface specifications that
 * extend this framework shall define an objectType value for each managed
 * object class that is modelled at the interface (by extending the
 * common::Common_T structure - see 8.5/X.780.2). All standard notification
 * types, except FTS and BSE, have this attribute. The enum
 * avoids any uncertainty in the type of object and allows simple filtering.
 * It cannot be extended in a backward compatible way, however.</p>
 *
 * <p>A notification must be reported against the correct object type,
```

```
 * if this object type is modelled at the interface.</p>
 *
 * <p>The standard object type OT_EMS represents the managed system (EMS).</p>
 *
 * <p>An interface specification that extends this framework may define
 * additional object types according to the rule
 * <code>OT_&lt;object type&gt;</code> where
 * <code>&lt;object type&gt;</code> is an acronym or a short phrase
 * that describes the object type and may consists only
 * of capital letters and underscores.</p>
 *
 * <p>For alarms, the object type value OT_AID denotes an Alarm Identifier
 * (AID) and is used to represent the EMS object types that are not modelled
 * (at the interface) but can emit alarms. Other notification types should
 * not be reported against AIDs. But this object type value shall also
 * be used for potential new object types in future releases in order to
 * guarantee backward compatibility of the interface.</p>
 **/
 enum ObjectType_T
 {
    OT_EMS,
 // OT_&lt;object type&gt;, for each object type supported by a standardized
 //                   interface specification that extends this framework
    OT_AID
 };

/**
 * <p>The ObjectTypeQualifier_T is used to identify object types of
 * future releases of standardized interface specifications that extend
 * this framework. It is needed because the ObjectType_T enum, once
 * specified for a particular release of an interface specification,
 * cannot be extended for backward compatibility reasons.</p>
 *
 * <p>So when backward compatibility is required (as usual - see
 * 10.5/X.780.2), the enum ObjectType_T cannot be extended to include
 * future new object types. Therefore OT_AID is used as an "escape
 * value" for the field objectType. Thus OT_AID may also represent
 * new object types. To identify which object type applies, the
 * filterable field attribute <b>objectTypeQualifier</b> is introduced,
 * which is of type string and whose values are as follows:<br>
 * <ul><li><code>""</code> indicates an AID;</li>
 * <li><code>"OT_"&lt;object type&gt;</code> where
 * &lt;object type&gt; identifies a new object type.</li></ul></p>
 *
 * <p>If objectTypeQualifier is not present but objectType
 * has the value OT_AID, we are dealing with a proper AID.</p>
 **/
 typedef string ObjectTypeQualifier_T;

/**
 * <p> Most notification types have a record member <b>emsTime</b>
 * of type globaldefs::Time_T, which holds the time at which the event
 * was reported by the managed system (EMS).</p>
 **/
 typedef globaldefs::Time_T EmsTime_T;

/**
 * <p> Most notification types have a record member <b>neTime</b>
 * of type globaldefs::Time_T, which holds the time provided by the NE
 * with the notification if the NE reports time, and is optional
 * or "" otherwise.</p>
 **/
 typedef globaldefs::Time_T NeTime_T;

/**
 * <p> Most notification types have a record member <b>edgePointRelated</b>
 * of type boolean, which is TRUE if the event relates to a TP that is
 * an edge point or to a managed object that is related to an edge TP,
 * and is FALSE or optional otherwise (i.e., may be not present).</p>
 **/
 typedef boolean EdgePointRelated_T;
```

```
/**
```

A.4.1 OCN, ODN, AVC, SCN, FTS, BSE and HBE notifications

```
*/

  /**
   * <p>Defines the OCN notification type. Holds details of the managed
   * object which is being created.</p>
   *
   * <p>The record member <b>remainderOfBody</b> is mapped to the
   * remaining event body of the OMG structured event and holds a
   * copy of the object which has been created; its actual type
   * depends on the value of objectType (and objectTypeQualifier)
   * and is then the struct defining the specified objectType
   * (which extends common::Common_T - see 8.5/X.780.2).</p>
   **/
   struct OCN_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
      any remainderOfBody;
   };

  /**
   * <p>Defines the ODN notification type. Holds details of the managed
   * object which is being deleted.</p>
   **/
   struct ODN_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      ObjectTypeQualifier_T objectTypeQualifier;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      boolean edgePointRelated;
   };

  /**
   * <p>The NameAndAnyValue_T structure is used when an any value is
   * needed to represent a general property of a notification (i.e.,
   * the recommended globaldefs::NameAndStringValue_T cannot be used
   * for the property) (see also 8.2/X.780.2).</p>
   **/
   struct NameAndAnyValue_T
   {
      string name;
      any value;
   };

  /**
   * <p>A list of (name=string, value=any) tuples. It is equivalent to
   * the NVList standard defined by the OMG to represent lists of general
   * properties (see also globaldefs::NVSList_T for properties with string
   * values). It is used for AVC, SCN and ALM notifications.</p>
   *
   * <p>In an AVC or SCN notification, the transmissionParams attribute's
   * value may not contain the complete value of the attribute. Rather,
```

```
 * it only indicates transmission parameters that have changed. The
 * value always contains a complete list of the Layer rates; but for
 * each layer, only those transmission parameters that have been deleted,
 * changed, or added are specified. A deleted transmission parameter is
 * indicated by a "-" value; a changed or added transmission parameter
 * has its new value specified. All transmission parameters not listed
 * have not changed.</p>
 *
 * <p>Similarly in an AVC or SCN notification, the additionalInfo
 * attribute's value may not contain the complete value of the
 * attribute. Rather, it only indicates NameAndStringValue_Ts that
 * have been deleted, changed, or added. A deleted NameAndStringValue_T
 * is indicated by a "-" value; a changed or added NameAndStringValue_T
 * has its new value specified. All NameAndStringValue_Ts not listed
 * have not changed.</p>
 **/
 typedef sequence<NameAndAnyValue_T> NVList_T;

/**
 * <p>Defines the AVC notification type. Holds details of the attribute
 * values of a managed object that have changed.</p>
 **/
 struct AVC_T
 {
    EventType_T eventType;
    string eventName;
    string notificationId;
    globaldefs::NamingAttributes_T objectName;
    ObjectType_T objectType;
    ObjectTypeQualifier_T objectTypeQualifier;
    globaldefs::Time_T emsTime;
    globaldefs::Time_T neTime;
    boolean edgePointRelated;
    NVList_T attributeList;
 };

/**
 * <p>Defines the SCN notification type. Holds details of the state
 * attribute values of a managed object that have changed.</p>
 **/
 struct SCN_T
 {
    EventType_T eventType;
    string eventName;
    string notificationId;
    globaldefs::NamingAttributes_T objectName;
    ObjectType_T objectType;
    ObjectTypeQualifier_T objectTypeQualifier;
    globaldefs::Time_T emsTime;
    globaldefs::Time_T neTime;
    boolean edgePointRelated;
    NVList_T attributeList;
 };

/**
 * </p>Describes the file transfer (FT) status.</p>
 *
 * <p>The FT status values are defined as follows:
 * <ul><li><b>FT_IN_PROGRESS</b>: The EMS notifies that
 * the FT is in progress including the percentage of
 * completion (in the range 0..100).</li>
 * <li><b>FT_FAILED</b>: The EMS notifies that the FT
 * has failed including the failure reason and, optionally,
 * the percentage of completion.</li>
 * <li><b>FT_COMPLETED</b>: The EMS notifies that
 * the FT is 100 % completed.</li></ul></p>
 *
 * <p>The number and rate of FTS notifications indicating FT_IN_PROGRESS
 * is implementation-defined, and may be zero. However, for each FT
 * process initiated (directly or indirectly) by the EMS there shall be
 * either an FTS with FT_COMPLETED or an FTS with FT_FAILED.</p>
```

```
 **/
 enum FileTransferStatus_T
 {
    FT_IN_PROGRESS,
    FT_FAILED,
    FT_COMPLETED
 };

/**
 * <p>Defines the FTS notification type. Holds details of a
 * file transfer (FT) process.</p>
 *
 * <p>string <b>fileName</b>:<br>
 * The name of the file being transferred including the path name
 * and exactly as specified when the FT request was initiated.<br>
 *
 * FileTransferStatus_T <b>transferStatus</b>:<br>
 * Holds the FT status value.
 *
 * short <b>percentComplete</b>:<br>
 * Indicates percentage of FT completion in the range 0..100.<br>
 *
 * string <b>failureReason</b>:<br>
 * Indicates the failure reason in case of FT_FAILED.</p>
 **/
 struct FTS_T
 {
    EventType_T eventType;
    string eventName;
    string notificationId;
    string fileName;
    FileTransferStatus_T transferStatus;
    short percentComplete;
    string failureReason;
 };

/**
 * <p>This enum identifies the status of a managed element with
 * respect to the current database backup operation. Initially
 * when the EMS is started the Current Operational Status (COS)
 * will be set to COS_Idle.</p>
 *
 * <p>The current operational status values are defined as follows:
 * <ul><li><b>COS_Idle</b>: No database backup operation has been
 * performed since the EMS last (re)started (booted).</li>
 * <li><b>COS_Pending</b>: A backup operation has been requested
 * but has not yet started.</li>
 * <li><b>COS_InProgress</b>: A backup operation is being performed.</li>
 * <li><b>COS_Completed</b>: Last backup operation was successful.</li>
 * <li><b>COS_Aborted</b>: Last backup operation failed.</li></ul></p>
 **/
 enum Current_OperationStatus_T
 {
    COS_Idle,
    COS_Pending,
    COS_InProgress,
    COS_Completed,
    COS_Aborted
 };

/**
 * <p>This data structure identifies the status of backup operation
 * for a managed element. The failure reason should be present if the
 * operation status indicates a failure (i.e., in Abort status).</p>
 *
 * <p>Current_OperationStatus_T <b>opStatus</b>:<br>
 * Indicates the current operational status of the backup.<br>
 *
 * <br>string <b>failureReason</b>:<br>
 * A free form text string provided if the opStatus value is COS_Aborted
 * to explain the reason for the abort (e.g., "Comms loss with NE").</p>
```

```
   **/
   struct BackupStatus_T
   {
      Current_OperationStatus_T opStatus;
      string failureReason;
   };

  /**
   * <p>Defines the BSE notification type. Holds details of a change
   * to a network element's backup status.</p>
   **/
   struct BSE_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T meName;
      globaldefs::Time_T emsTime;
      globaldefs::Time_T neTime;
      BackupStatus_T backupStatus;
   };

  /**
   * <p>Defines the HBE notification type. Holds details of an
   * EMS heartbeat notification. The value of the record member
   * objectType is always OT_EMS.</p>
   *
   * <p> Refer to 7.2.5/Q.816.2, 9.1.1/Q.816.2 and 7.5/Q.816
   * for information on how this interface may be used or extended
   * to offer a heartbeat service (Q.816.2 Session Service
   * or Q.816 Heartbeat Service).</p>
   **/
   struct HBE_T
   {
      EventType_T eventType;
      string eventName;
      string notificationId;
      globaldefs::NamingAttributes_T objectName;
      ObjectType_T objectType;
      globaldefs::Time_T emsTime;
   };


/**
```

A.4.2 ALM and TCA notifications

```
*/

  /**
   * <p>ALM and TCA notifications have an optional record member
   * <b>nativeEMSName</b> of type string which represents the name of
   * the managed object, which emitted the alarm resp. which is the
   * measurement point whose threshold (of one of its PM parameters) has
   * been crossed, as presented on the EMS GUI. The native EMS name
   * attribute of a managed object is owned by the EMS which may or
   * may not support changing this value (see 8.5/X.780.2).</p>
   **/
   typedef string NativeEMSName_T;

  /**
   * <p>PerceivedSeverity_T values are consistent with the definitions
   * of Recs. X.733 and X.721. Refer to clause 8.1.2.3 of Rec. X.733 and
   * Rec. X.733/Cor.2 for the semantics of the values. Alarms and TCAs
   * have a mandatory attribute <b>perceivedSeverity</b> of this type.</p>
   *
   * <p>The Perceived severity levels provide an indication of how it is
   * perceived that the capability of a managed object reporting an alarm
   * or a TCA has been affected. Four levels represent service affecting
   * (SA) or non-service affecting (NSA) conditions according to the
```

```
 * determination by the managed system (EMS) (see ServiceAffecting_T):
 * as stated by Rec. X.733, usually PS_CRITICAL and PS_MAJOR are SA and
 * require immediate resp. urgent corrective action, while usually PS_MINOR
 * and PS_WARNING are NSA resp. potentially SA and require corrective
 * resp. diagnostic action. The semantics of the PS_CLEARED severity
 * level depends on the values of the x733_SpecificProblems and
 * x733_CorrelatedNotifications field attributes (if present).</p>
 *
 * <p>In case of a TCA the perceivedSeverity attribute mainly serves as
 * the trigger flag, i.e., allows to distinguish between a trigger/raise
 * TCA (value PS_INDETERMINATE or, optionally, some SA or NSA level)
 * and a clear TCA (value PS_CLEARED).</p>
 **/
 enum PerceivedSeverity_T
 {
    PS_INDETERMINATE,
    PS_CRITICAL,
    PS_MAJOR,
    PS_MINOR,
    PS_WARNING,
    PS_CLEARED
 };

/**
 * <p>List of PerceivedSeverity_T values.</p>
 **/
 typedef sequence<PerceivedSeverity_T> PerceivedSeverityList_T;

/**
 * <p>ServiceAffecting_T describes the impact of a fault on monitored
 * managed entities on further providing their service, as determined
 * by the managed system (EMS). Alarms have a mandatory record member
 * <b>serviceAffecting</b> of this type. The following SA values are defined:
 *
 * <ul><li><b>SA_UNKNOWN</b>: The EMS cannot determine whether
 * the condition affects service or does not.</li>
 *
 * <li><b>SA_SERVICE_AFFECTING</b>: The EMS determines that the condition
 * affects service (e.g., interrupts payload traffic).</li>
 *
 * <li><b>SA_NON_SERVICE_AFFECTING</b>: The EMS determines that
 * the condition does not affect service.</li></ul></p>
 **/
 enum ServiceAffecting_T
 {
    SA_UNKNOWN,
    SA_SERVICE_AFFECTING,
    SA_NON_SERVICE_AFFECTING
 };

/**
 * <p>Each alarm (ALM notification) has an optional record member
 * <b>affectedTPList</b> of type globaldefs::NamingAttributesList_T which
 * holds a list of affected TPs (ordered by TP names). It shall indicate
 * in case of alarms on equipment the physical TPs (ports) implemented
 * by the alarmed equipment and affected by the equipment failure
 * (irrespective of whether the alarm is service affecting or not).</p>
 **/
 typedef globaldefs::NamingAttributesList_T AffectedTPList_T;

/**
 * <p>Alarms have a mandatory record member <b>probableCause</b> of type
 * string which defines further qualification as to the Probable cause
 * of the alarm in the sense of Rec. X.733. Probable causes are used
 * to classify alarm types.</p>
 *
 * <p>The framework does not encode specific Probable causes but
 * any OS-OS interface specification that extends this framework
 * shall specify all Probable causes that may occur according to
 * the rules laid down in 8.7.3.1/X.780.2 "Probable causes"
 * (e.g., "AIS" for Alarm Indication Signal, "BER_SD"
```

```
 * for Bit Error Rate Signal Degrade).</p>
 *
 * <p>For example, Recs. X.733 and X.721 define 57 Probable causes,
 * Rec. M.3100 defines or reserves 207 Probable causes that partly
 * overlap with the X.733/X.721 definitions, and TM Forum's MTNM v3.0
 * defines 96 Probable cause strings (see 8.7.3.1/X.780.2).</p>
 *
 * <p>Refer to the standardized OS-OS interface specifications
 * that extend this framework for the normative specification and
 * description of the Probable cause strings available with the
 * respective release of the interface.</p>
 **/
 typedef string ProbableCause_T;

/**
 * <p>Set of Probable causes. This type may be used, for example,
 * in case of on-demand pulling of alarms in order to include
 * or exclude specified Probable causes.</p>
 **/
 typedef sequence<string> ProbableCauseList_T;

/**
 * <p>Alarms have an optional record member <b>probableCauseQualifier</b>
 * of type string which is meant to be used together with objectName,
 * layerRate and probableCause to fully and uniquely identify the alarm.
 * If this attribute is an empty string (or not present in the OMG
 * Structured Event), the other three attributes must provide
 * uniqueness. Otherwise this attribute contains information such that
 * any clear of that alarm would correlate to the correct alarm.</p>
 **/
 typedef string ProbableCauseQualifier_T;

/**
 * <p>Alarms have an optional record member <b>nativeProbableCause</b>
 * of type string which shall represent the value of the Probable cause
 * as portrayed on the EMS user interface.</p>
 **/
 typedef string NativeProbableCause_T;

/**
 * <p>The relationship between Probable causes and Perceived severities
 * (and Service affection) as reported by ALM notifications is fundamental
 * for alarm management. Therefore the advance assignment of severities to
 * Probable causes per managed object (having regard to Service affection)
 * is of interest. For this purpose the concepts of Assigned severity and
 * Alarm Severity Assignment (ASA) are introduced for SA, NSA and service
 * independent usage (as determined by the managed system (EMS)).</p>
 *
 * <p>AssignedSeverity_T adapts PerceivedSeverity_T for managed
 * object-specific advance severity assignments to Probable causes.
 * Its values are consistent with the AlarmSeverityCode definition
 * of Rec. M.3100.</p>
 *
 * <p>The Assigned severity values are defined as follows:
 *
 * <ul><li>AS_INDETERMINATE: The Indeterminate level indicates that
 * the assigned severity level cannot be determined.</li>
 *
 * <li>AS_CRITICAL: The Critical level indicates that a usually
 * SA condition has occurred and an immediate corrective action
 * is required. Such a severity can be assigned, for example,
 * when an object would become totally out of service and its
 * entire capability must be restored.</li>
 *
 * <li>AS_MAJOR: The Major level indicates that a usually SA condition
 * has developed and an urgent corrective action is required. Such a
 * severity can be assigned, for example, when there would be a severe
 * degradation in the capability of the managed object and therefore
 * its full capability need be restored.</li>
 *
 * <li>AS_MINOR: The Minor level indicates the existence of a usually NSA
```

```
 * fault condition and that corrective action should be taken in order
 * to prevent a more serious (e.g., SA) fault. Such a severity can be
 * assigned, for example, when the detected alarm condition would not
 * currently degrade the capacity of the managed object.</li>
 *
 * <li>AS_WARNING: The Warning level indicates the detection of a
 * usually potential or impending SA fault before any significant
 * effects have been felt. Action should be taken to further
 * diagnose (if necessary) and correct the problem in order to prevent
 * it from becoming a more serious (e.g., SA) fault.</li>
 *
 * <li>AS_NONALARMED: The Nonalarmed level indicates that no alarm
 * should be raised.</li>
 *
 * <li>AS_FREE_CHOICE: The Free choice level indicates that the managed
 * system (EMS) or network element is free to make a choice of the
 * Perceived severity level based upon its local criteria.</li></ul></p>
 **/
 enum AssignedSeverity_T
 {
    AS_INDETERMINATE,
    AS_CRITICAL,
    AS_MAJOR,
    AS_MINOR,
    AS_WARNING,
    AS_NONALARMED,
    AS_FREE_CHOICE
 };

/**
 * <p>AlarmSeverityAssignment_T provides an Alarm Severity Assignment
 * (ASA), or rather three ASAs, to be used for managed object-specific
 * assessment of faults (which is out of scope of this framework).
 * It defines three values for severity to cover the cases of service
 * affecting conditions, non-service affecting conditions, and conditions
 * where the service impact is unknown. The structure fully identifies
 * the specific alarm type that it applies to by using the three available
 * Probable cause identifiers in combination. It is consistent with
 * the AlarmSeverityAssignment definition of Rec. M.3100.</p>
 *
 * <p>string <b>probableCause</b>:<br>
 * The Probable cause of the alarm to which the severities are assigned.
 * This is a normative string defined by the interface specification.<br>
 *
 * <br>string <b>probableCauseQualifier</b>:<br>
 * A qualifier of the Probable cause that is used to achieve uniqueness
 * in such cases where the Probable cause alone would not be sufficient.
 * This is a free format string which often is empty.<br>
 *
 * <br>string <b>nativeProbableCause</b>:<br>
 * The Probable cause representation as used on the user interface of
 * the managed system (EMS) or network element. This is a human-readable,
 * implementation-defined free format string.<br>
 *
 * <br>AssignedSeverity_T <b>serviceAffecting</b>:<br>
 * Severity assigned to the Probable cause when service affecting.<br>
 *
 * <br>AssignedSeverity_T <b>serviceNonAffecting</b>:<br>
 * Severity assigned to the Probable cause when not service affecting.<br>
 *
 * <br>AssignedSeverity_T <b>serviceIndependentOrUnknown</b>:<br>
 * Severity assigned to the Probable cause when the service impact is
 * not known or is known to be service independent.</p>
 **/
 struct AlarmSeverityAssignment_T
 {
    string probableCause;
    string probableCauseQualifier;  // OPTIONAL
    string nativeProbableCause;  // OPTIONAL
    AssignedSeverity_T serviceAffecting;
    AssignedSeverity_T serviceNonAffecting;
```

```
    AssignedSeverity_T serviceIndependentOrUnknown;
 };

/**
 * <p>AlarmSeverityAssignmentList_T provides a listing of all abnormal
 * conditions that may exist in instances of a managed object class. Each
 * element of this sequence specifies a Probable cause and, optionally,
 * a probableCauseQualifier and/or a nativeProbableCause as well as
 * the three severities to be assigned to the Probable cause.</p>
 **/
 typedef sequence<AlarmSeverityAssignment_T> AlarmSeverityAssignmentList_T;

/**
 * <p>Alarms have an optional record member <b>additionalText</b> of type
 * string which allows, in an X.733-consistent way, a free form text
 * description to be reported in order to convey additional textual
 * information on the alarm. Such information can range from a simple note
 * (e.g., "Unit is mismounted") to an entire notepad (e.g., details of
 * event acknowledgement or clearance such as note priority/time/text
 * and the user name of the attending operator).</p>
 **/
 typedef string AdditionalText_T;

/**
 * <p>Alarms have an optional record member <b>additionalInfo</b> of type
 * globaldefs::NVSList_T which allows, according to stringification
 * conventions, the communication from the EMS to the NMS of additional
 * information that is not explicitly modelled (e.g., any X.733-specified
 * details that are not yet covered otherwise) (see also 8.5/X.780.2).</p>
 **/
 typedef globaldefs::NVSList_T AdditionalInfo_T;

/**
 * <p>Alarms have a mandatory record member <b>rcaiIndicator</b> of
 * type boolean which shall indicate whether the alarm is a raw,
 * i.e., uncorrelated, alarm (default situation) or a root cause alarm
 * indication (RCAI).</p>
 **/
 typedef boolean RcaiIndicator_T;

/**
 * <p>ALM and TCA notifications have a mandatory record member
 * <b>isClearable</b> of type boolean which is an indication as to
 * whether a raise/trigger event will have an associated clear event
 * or the event is a clear itself. If an ALM or TCA clear event is
 * generated in case of clearance then the raise/trigger event
 * is defined to be clearable.</p>
 **/
 typedef boolean IsClearable_T;

/**
 * <p>AcknowledgeIndication_T describes the event acknowledgement
 * indication. Active and cleared ALM and TCA notifications have
 * an optional record member <b>acknowledgeIndication</b> of this type.
 * Refer to section 3.7 of the OMG Notification Service specification
 * regarding support of event acknowledgement based on sequence numbers
 * that allow (together with QoS properties) for reliable event delivery
 * and are conveyed in the variable event header. Refer to clause
 * 8.7.5.3 "Event acknowledgement" of Rec. X.780.2 for a description of
 * how to use these sequence numbers or the ALM and TCA identifiers
 * (see AlarmId_T and TCAId_T) for event acknowledgement.</p>
 *
 * <p>AI_EVENT_ACKNOWLEDGED: Used in case the event has been
 * acknowledged in the managed system (EMS), or in case the event
 * has been previously acknowledged and has now been requested to
 * be unacknowledged but the EMS is unable to do so. All event
 * fields other than emsTime and acknowledgeIndication shall
 * remain similar to the original ALM or TCA notification.
 * The emsTime is always provided as the time that the event
 * acknowledgement notification has been reported by the EMS.<br>
 *
```

```
 * <br>AI_EVENT_UNACKNOWLEDGED: Used in case the event has not (yet)
 * been acknowledged in the managed system but the EMS supports
 * acknowledgement for this event, or in case the event has been
 * previously acknowledged and has now been unacknowledged.
 * In this case all event fields other than emsTime shall
 * remain similar to the original ALM or TCA notification.
 * The emsTime is always provided as the time that the event
 * (un)acknowledgement notification has been reported by the EMS.<br>
 *
 * <br>AI_NA: Used in case the EMS does not support acknowledgement for
 * this event or does not support acknowledgement at all.</p>
 **/
 enum AcknowledgeIndication_T
 {
    AI_EVENT_ACKNOWLEDGED,
    AI_EVENT_UNACKNOWLEDGED,
    AI_NA
 };


/**
 * <p>The alarm definition of the service-oriented framework provides
 * means to allow for conveyance of fully X.733-compliant alarms resp.
 * of all X.733-specified details that a managed system (EMS) may
 * offer to managing systems (NMSes) (see 8.7.3.3/X.780.2
 * "Coding of X.733 alarm information" for a concise overview).</p>
 *
 * <p>The <b>Threshold information</b> parameter of the X.733/X.721
 * alarm record is not present in the alarm definition of this framework
 * since threshold crossings are not reported by alarm notifications
 * but by TCA notifications (see TCA_T). Refer to clause 8.7.3.3 "Coding
 * of X.733 alarm information" of Rec. X.780.2 for the mapping of X.733
 * Threshold information to TCA event attributes.</p>
 *
 * <p>The <b>x733_EventType</b> attribute of type string specifies the
 * type of ITU-T event being reported where the semantics is defined in
 * Recs. X.710 and X.733 and the ITU-T syntax in Rec. X.711. As part
 * of an alarm record it categorizes the alarm.</p>
 *
 * <p>Five basic ITU-T categories of an alarm are defined:<br>
 *
 * "communicationsAlarm" - an alarm of this type is principally
 * associated with the procedures and/or processes required to convey
 * information from one point to another;<br>
 *
 * "qualityofServiceAlarm" - an alarm of this type is principally
 * associated with a degradation in the quality of a service;<br>
 *
 * "processingErrorAlarm" - an alarm of this type is principally
 * associated with a software or processing fault;<br>
 *
 * "equipmentAlarm" - an alarm of this type is principally
 * associated with an equipment fault;<br>
 *
 * "environmentalAlarm" - an alarm of this type is principally
 * associated with a condition relating to an enclosure
 * in which the equipment resides.</p>
 *
 * <p>The X.730-series Recommendations define further notification
 * types whose GDMO definitions and ASN.1 productions are summarized
 * in Rec. X.721: objectCreation, objectDeletion, attributeValueChange,
 * stateChange, relationshipChange, and five security violation event
 * types. They correspond to the CORBA notification types defined
 * in Rec. X.780 which are encoded in the type_name field of the fixed
 * event header of OMG structured events as are the notification types
 * of this framework (see 8.5/Q.816.2 and 8.7/X.780.2).</p>
 *
 * <p>Rec. X.733 includes informative recommendations for the mapping
 * of each defined Probable cause to one of the five alarm event types.
 * For example, thresholdCrossed is mapped to qualityofService. This
 * alarm is dealt with as the separate event type TCA by this framework
```

```
 * (see TCA_T for details). Similarly Rec. M.3100 groups the defined
 * and reserved Probable causes according to the alarm event types.</p>
 **/
 typedef string X733_EventType_T;

/**
 * <p>The <b>x733_EventTime</b> attribute contains the time of generation
 * of the alarm. Its type is a generalized time (see globaldefs::Time_T).
 * This may be equal to one of the attributes emsTime or neTime.
 * The definition complies with the syntax and semantics of
 * Recs. X.710 and X.721.</p>
 **/
 typedef globaldefs::Time_T X733_EventTime_T;

/**
 * <p>Element of SpecificProblemList_T (see below), i.e., a clarification of
 * the Probable cause of an alarm. It is designed to be human-readable and
 * compatible with ITU-T usage and semantics as defined by Rec. X.733.</p>
 **/
 typedef string SpecificProblem_T;

/**
 * <p>List of SpecificProblem_T values. When present in an alarm, it
 * identifies further refinements to the Probable cause of the alarm.
 * The optional <b>x733_SpecificProblems</b> field attribute uses that type.
 * It fully qualifies the chosen Probable cause and may be used to specify,
 * in an X.733-consistent way, a set of Specific problems for use in
 * managed object classes that emit alarms with that Probable cause.</p>
 *
 * <p>When stringified (e.g., through separation of the elements
 * of the list by a slash character "/"), it corresponds to the
 * probableCauseQualifier.</p>
 **/
 typedef sequence<SpecificProblem_T> SpecificProblemList_T;

/**
 * <p>The <b>x733_BackupStatus</b> attribute of type string specifies whether
 * or not the managed object emitting the alarm has been backed-up, and
 * services provided to the user have, therefore, not been disrupted.
 * If the value is "BACKED_UP", it indicates that the object emitting
 * the alarm has been backed-up, if "NOT_BACKED_UP", the object has not
 * been backed-up. The use of this field in conjunction with the
 * perceivedSeverity field provides information in an independent form
 * to qualify the seriousness of the alarm and the ability of the managed
 * system (EMS) as a whole to continue to provide services.</p>
 **/
 typedef string X733_BackupStatus_T;

/**
 * <p>The <b>x733_BackupObject</b> attribute specifies, when x733_BackupStatus
 * is "BACKED_UP", the (name of the) managed object instance that is providing
 * back-up services for the managed object about which the alarm pertains.
 * This attribute is useful, for example, when the back-up object is from
 * a pool of managed objects any of which may be dynamically allocated
 * to replace a faulty managed object.</p>
 **/
 typedef globaldefs::NamingAttributesList_T X733_BackupObject_T;

/**
 * <p>The <b>x733_TrendIndication</b> attribute of type string specifies
 * the current Perceived severity trend of the managed object reporting
 * the alarm. If present it indicates that there are one or more alarms
 * ("outstanding alarms") which have not been cleared, and pertain to
 * the same managed object as that to which this alarm ("current alarm")
 * pertains. It has the three possible values "LESS_SEVERE", "NO_CHANGE",
 * and "MORE_SEVERE" with the semantics defined by Rec. X.733. In order
 * for x733_TrendIndication to be meaningful, the perceivedSeverity
 * attribute value of each alarm that may be emitted by the considered
 * managed object must be defined consistently over all of the defined
 * alarm types for that managed object class.</p>
 **/
```

```
 typedef string X733_TrendIndication_T;

/**
 * <p>List of notification IDs (i.e., values of the field attribute
 * <b>notificationId</b> in the notifications).</p>
 **/
 typedef sequence<string> NotifIDList_T;

/**
 * <p> Correlated notifications are identified by the object that emitted
 * the notification and the notification IDs. Both are included in case
 * the notification IDs are not unique across managed objects.</p>
 *
 * <p>globaldefs::NamingAttributes_T <b>source</b>:
 * Reference to the managed object that emitted the correlated
 * notifications. If empty, the correlated notifications are from
 * the same source as the alarm containing this data structure.<br>
 *
 * <br>NotifIDList_T <b>notifIDs</b>:
 * List of notification IDs of the correlated notifications.
 * To use this structure, notification IDs must be chosen to be
 * unique across all notifications from a particular managed object
 * throughout the time that correlations are significant.</p>
 **/
 struct CorrelatedNotifications_T
 {
    globaldefs::NamingAttributes_T source;
    NotifIDList_T notifIDs;
 };

/**
 * <p>List of CorrelatedNotifications_T values. The optional
 * <b>x733_CorrelatedNotifications</b> field attribute uses this type.
 * When present in an alarm, it contains, in an X.733-consistent way,
 * a set of notification identifiers and, if necessary, their associated
 * object names. This set is defined to be the set of all notifications
 * to which this alarm is considered to be correlated.</p>
 **/
 typedef sequence<CorrelatedNotifications_T> CorrelatedNotificationList_T;

/**
 * <p>The <b>x733_MonitoredAttributes</b> attribute of type NVList_T defines
 * one or more attributes of the managed object and their corresponding
 * values (of arbitrary type) at the time of the alarm (e.g., state or
 * status attributes). This allows, for example, the timely reporting of
 * changing conditions prevalent at the time of the alarm generation.</p>
 **/
 typedef NVList_T X733_MonitoredAttributes_T;

/**
 * <p>Element of ProposedRepairActionList_T (see below), i.e., a solution
 * proposed by the managed system (EMS) to a known cause of an alarm (such
 * as switch in standby equipment, retry, replace media). It is designed
 * to be human-readable and compatible with ITU-T usage and semantics as
 * defined by Rec. X.733. Two general values are specified by Rec. X.733:
 * "no repair action required" and "repair action required".</p>
 **/
 typedef string ProposedRepairAction_T;

/**
 * <p>List of ProposedRepairAction_T values. When present in an alarm, it
 * indicates that the cause of the alarm is known and the system being
 * managed (EMS) can suggest one or more solutions. The optional
 * <b>x733_ProposedRepairActions</b> field attribute uses this type.
 * It may be used to specify, in an X.733-consistent way, a set of
 * Proposed repair actions for use in managed object classes that emit
 * alarms with that known (Probable) cause.</p>
 **/
 typedef sequence<ProposedRepairAction_T> ProposedRepairActionList_T;

/**
```

```
 * <p>The <b>x733_StateChange</b> attribute of type NVList_T is an
 * X.733-compliant AVC definition, i.e., it contains a series of state
 * attribute identifiers and their (old and)new values that can be used
 * in an SCN notification. This attribute, when present, is used to
 * indicate an X.731 state transition associated with the alarm.
 * In this case, if the MOC definition of the managed object emitting
 * the alarm includes notifications this managed object should also
 * emit an SCN as specified above.</p>
 **/
 typedef NVList_T X733_StateChange_T;

/**
 * <p>The <b>x733_AdditionalInfo</b> attribute of type NVList_T is used
 * to supply additional information in alarms. It allows the inclusion
 * of a set of additional information in the alarm whose syntax and
 * semantics are specified by Recs X.721 and X.733 based on the concept
 * of management extensions. The representation of the X.721/X.733
 * information in the NVList_T structure is implementation-defined
 * (free format).</p>
 **/
 typedef NVList_T X733_AdditionalInfo_T;


/**
 * <p>AlarmId_T is used as a unique identifier of an alarm. It includes:</p>
 *
 * <p>globaldefs::NamingAttributes_T <b>objectName</b>:<br>
 * The name represents the name of the managed entity that gave rise
 * to the alarm. Refer to clause 10.3.2 "Structure of naming attribute
 * of managed objects" of Rec. X.780.2 for further details.<br>
 *
 * <br>transmissionParameters::LayerRate_T <b>layerRate</b>:<br>
 * Identifies the layerRate of the managed object raising the alarm.
 * For objects where the layerRate is not applicable, such as EMS, the
 * value is set to LR_Not_Applicable. LayerRate is applicable in alarms
 * raised by objects that include transmission parameters such as TPs.<br>
 *
 * <br>string <b>probableCause</b>:<br>
 * Identifies the type of alarm raised against the object.
 * Refer to clause 8.7.3.1 of Rec. X.780.2 and the standardized interface
 * specifications that extend this framework for a normative
 * specification of the Probable cause strings that may occur.<br>
 *
 * <br>string <b>probableCauseQualifier</b>:<br>
 * Is used as the final component of the unique
 * identification of the alarm and is left empty where
 * objectName, layerRate (if applicable) and probableCause alone
 * provide a unique identification of the alarm.</p>
 **/
 struct AlarmId_T
 {
    globaldefs::NamingAttributes_T objectName;
    transmissionParameters::LayerRate_T layerRate;
    string probableCause;
    string probableCauseQualifier; // OPTIONAL
 };

/**
 * <p>Defines the ALM notification type. Holds details of an alarm.</p>
 *
 * <p>An alarm must be reported against the correct managed object if it
 * is modelled by the EMS (see ObjectType_T and ObjectTypeQualifier_T).
 * Alarms against entities which are not modelled at the interface should
 * still be reported using the OT_AID objectType. The EMS should ensure
 * that all such virtual entities have a unique value for objectName.</p>
 *
 * <p>The combination of the objectName, layerRate, probableCause and
 * probableCauseQualifier subfields of the alarmId field must uniquely
 * identify an alarm. It is acceptable that the probableCauseQualifier
 * be an empty string, provided that uniqueness (using the remaining
 * three fields) is still guaranteed. Together, these four fields contain
```

```
 * information such that any clear of an alarm would correlate to the
 * correct alarm. This means that if an alarm is raised, cleared,
 * raised again, and cleared again, and if the original clear and second
 * alarm were missed, the second clear would clear the first alarm.</p>
 *
 * <p>The comment OPTIONAL means that the record member need not be
 * present in the corresponding OMG Structured Event.</p>
 *
 * <p>When mapping the attribute alarmId to the filterable event body of
 * the OMG Structured Event (see Event_T), each record member of AlarmId_T
 * shall be mapped to a separate filterable data field. When mapping the
 * x733-attributes to the filterable event body of the Structured Event,
 * their name prefix shall be changed from "x733_" to "X.733::".</p>
 *
 * <p>For some type declarations (e.g., X.733-related ones) their unnamed
 * equivalents are used. This is justified for the sake of readability
 * and since ALM_T is mapped to structured events anyway.</p>
 **/
 struct ALM_T
 {
    EventType_T eventType;
    string eventName;
    string notificationId;
    AlarmId_T alarmId;
    string nativeProbableCause; // OPTIONAL
    ObjectType_T objectType;
    ObjectTypeQualifier_T objectTypeQualifier;
    globaldefs::Time_T emsTime;
    globaldefs::Time_T neTime;
    boolean edgePointRelated;
    string nativeEMSName; // OPTIONAL
    PerceivedSeverity_T perceivedSeverity;
    ServiceAffecting_T serviceAffecting;
    globaldefs::NamingAttributesList_T affectedTPList; // OPTIONAL
    string additionalText; // OPTIONAL
    globaldefs::NVSList_T additionalInfo; // OPTIONAL
    boolean rcaiIndicator;
    boolean isClearable;
    AcknowledgeIndication_T acknowledgeIndication; // OPTIONAL
    string x733_EventType; // OPTIONAL
    globaldefs::Time_T x733_EventTime; // OPTIONAL
    SpecificProblemList_T x733_SpecificProblems; // OPTIONAL
    string x733_BackedUpStatus; // OPTIONAL
    globaldefs::NamingAttributesList_T x733_BackUpObject; // OPTIONAL
    string x733_TrendIndication; // OPTIONAL
    CorrelatedNotificationList_T x733_CorrelatedNotifications; // OPTIONAL
    NVList_T x733_MonitoredAttributes; // OPTIONAL
    ProposedRepairActionList_T x733_ProposedRepairActions; // OPTIONAL
    NVList_T x733_StateChange; // OPTIONAL
    NVList_T x733_AdditionalInfo; // OPTIONAL
 };


/**
 * <p>The framework does not encode specific performance parameters
 * but any interface specification that extends this framework shall
 * specify all performance parameters that may occur in the normative
 * form <code>"PMP_&lt;performance measure&gt;"</code> according to
 * the rules laid down in clause 8.7.4.1 "PM parameters" of Rec.
 * X.780.2 (e.g., <code>"PMP_AISS"</code> for Alarm Indication Signal
 * Seconds, <code>"PMP_BER"</code> for Bit Error Rate).</p>
 *
 * <p>PMParameterName_T holds the name of a performance measurement
 * (i.e., a performance parameter). It has been defined as a string
 * to accommodate backward compatibility and proprietary extension. TCAs
 * have a mandatory attribute <b>pmParameterName</b> of this type.</p>
 **/
 typedef string PMParameterName_T;

/**
 * <p>PMLocation_T specifies the location and orientation/direction
```

```
 * for the measurement of the associated PM parameter thereby qualifying
 * the traffic to be monitored, supervised and measured. It has been
 * defined as a string to accommodate extensibility. TCAs have
 * a mandatory attribute <b>pmLocation</b> of this type.</p>
 *
 * <p>PM parameters may relate to measurements taken on receive (Rx)
 * or transmit (Tx) traffic either at the named TP (PML_NEAR_END_Rx/Tx)
 * or at the TP at the far end of the trail/connection connected to
 * the named TP (PML_FAR_END_Rx/Tx). Alternatively, the PM parameters
 * may be bidirectional (PML_BIDIRECTIONAL) (e.g., resulting from a
 * second by second summation and evaluation of both far and near TPs).
 * Measurements may also be carried out contra-directional to the
 * supervised signal (PML_CONTRA_NEAR/FAR_END_Tx).</p>
 *
 * <p>Valid standard values are:<br>
 * "PML_NEAR_END_Rx",<br>
 * "PML_FAR_END_Rx",<br>
 * "PML_NEAR_END_Tx",<br>
 * "PML_FAR_END_Tx",<br>
 * "PML_BIDIRECTIONAL",<br>
 * "PML_CONTRA_NEAR_END_Tx",<br>
 * "PML_CONTRA_FAR_END_Tx".</p>
 *
 * <p>Refer to clause 8.7.4.2 "Location identification"
 * of Rec. X.780.2 for further details.</p>
 **/
 typedef string PMLocation_T;

/**
 * <p>Granularity_T specifies the count period resp. measurement
 * period (interval or duration) for which PM data may be
 * collected (and subsequently retrieved). TCAs have
 * a mandatory attribute <b>granularity</b> of this type.</p>
 *
 * <p>The format is one of:
 * <ul><li>"<i>n</i>min" (representing <i>n</i> minute granularity)
 * for values of <i>n</i> that are not multiples of 60;</li>
 * <li>"<i>n</i>h" (representing <i>n</i> hour granularity);</li>
 * <li>"NA" (not applicable, representing instantaneous
 * measurements).</li></ul></p>
 *
 * <p>Standard values are:<br>
 * "15min",<br>
 * "24h",<br>
 * "NA" (for current instantaneous measurements).<br></p>
 **/
 typedef string Granularity_T;

/**
 * <p>PMThresholdType_T describes threshold watermark (TWM) levels,
 * or threshold types for short, of TCA notifications. TCAs have
 * a mandatory attribute <b>thresholdType</b> of this type.</p>
 *
 * <p>The TWM_HIGH and TWM_HIGHEST types are used for TCAs that are
 * raised when the measured value goes above the threshold. The
 * TWM_LOW and TWM_LOWEST types are used for TCAs that are raised
 * when the measured value goes below the threshold; they only apply
 * to gauges since counters do not count down (but may wrap around).</p>
 *
 * <p>When there is one level of TCA notification triggers, only
 * TWM_HIGH and/or TWM_LOW are used, and when there are two levels of
 * TCA triggers, TWM_HIGHEST and/or TWM_LOWEST are used in addition.</p>
 *
 * <p>Refer to clause 8.7.4.4 "PM thresholds" of Rec. X.780.2 for
 * information on how watermark levels are used for raise/trigger
 * and clear TCAs. Note that the perceivedSeverity attribute (see
 * PerceivedSeverity_T) serves (mainly, in case of TCAs) as the
 * trigger flag (i.e., it indicates whether a TCA is a clear or
 * is a trigger/raise with or without severity).</p>
 **/
 enum PMThresholdType_T
```

```
 {
    TWM_HIGHEST,
    TWM_HIGH,
    TWM_LOW,
    TWM_LOWEST
 };

/**
 * <p>TCAId_T is used as a unique identifier of a threshold crossing
 * alert (TCA). It includes:</p>
 *
 * <p>globaldefs::NamingAttributes_T <b>objectName</b>:<br>
 * The name represents the name of the managed entity that gave rise
 * to the TCA (i.e., the measurement point). Refer to clause 10.3.2
 * "Structure of naming attribute of managed objects" of Rec. X.780.2
 * for further detail on the name structure.<br>
 *
 * <br>transmissionParameters::LayerRate_T <b>layerRate</b>:<br>
 * Identifies the layerRate of the managed object raising the TCA.
 * For objects where the layerRate is not applicable, such as EMS, the
 * value is set to LR_Not_Applicable. LayerRate is applicable in TCAs
 * raised by objects that include transmission parameters such as TPs.<br>
 *
 * <br>PMParameterName_T <b>pmParameterName</b>:<br>
 * Holds the name of the performance measure for which the TCA was
 * raised (i.e., the subject of performance measurement).<br>
 *
 * <br>PMLocation_T <b>pmLocation</b>:<br>
 * Holds the location and orientation of the performance measure.<br>
 *
 * <br>Granularity_T <b>granularity</b>:<br>
 * Holds the count/measurement period of the performance measure.</p>
 **/
 struct TCAId_T
 {
    globaldefs::NamingAttributes_T objectName;
    transmissionParameters::LayerRate_T layerRate;
    PMParameterName_T pmParameterName;
    PMLocation_T pmLocation;
    Granularity_T granularity;
    PMThresholdType_T thresholdType;
 };

/**
 * <p>TCA notifications have an optional attribute <b>value</b> of type
 * float, the threshold value, which holds the value of the threshold
 * (counter or gauge) that has been crossed if known.</p>
 **/
 typedef float Value_T;

/**
 * <p>TCA notifications have an optional attribute <b>unit</b> of type
 * string, the threshold unit, which holds the unit of measurement for
 * the threshold value in a free format if value is present.</p>
 **/
 typedef string Unit_T;

/**
 * <p>Defines the TCA notification type. Holds details of a Threshold
 * Crossing Alert notification (TCA), which is an event used across
 * the OS-OS interface to indicate that a Performance Monitoring
 * (PM) parameter threshold has been crossed.</p>
 *
 * <p>This framework calls the X.733 alarm notification with Probable
 * cause thresholdCrossed (and event type qualityofService) a Threshold
 * Crossing Alert (TCA) and defines its structure independently of the
 * ALM notification structure. The TCA structure and behaviour depend
 * on the concepts of service-oriented Performance Management (PM)
 * outlined in 8.7.4/X.780.2. TCA notifications have specific PM-related
 * attributes not found in other alarms and do not need a Probable cause
 * and related attributes, and ALM notifications do not provide any
```

```
 * threshold information.</p>
 *
 * <p>The targets of PM are managed objects such as TPs which are
 * called measurement points and where access points to PM activities
 * are identified by certain transmission Layer rates, locations, and
 * granularities. PM parameters are the subjects of PM activities on
 * measurement points (such as performance monitoring, threshold
 * supervision, and PM data collection) and may have associated one or
 * more PM thresholds. When such a threshold is being crossed (and
 * threshold supervision is activated), the measurement point emits
 * a TCA notification with the characteristics of the PM parameter
 * and PM threshold. Such a complete set of threshold crossing
 * characteristics is called a TCA parameter (PM parameter name,
 * PM location, granularity, PM threshold) and is applicable to
 * one or more (or all) transmission Layer rates.</p>
 *
 * <p>TCA notifications are the results of threshold supervision but
 * the real results of PM activities on measurement points are the PM data
 * which are stored for a certain holding time as current PM data in
 * collection bins according to the required granularity (e.g., 15min,
 * 24h, instantaneous) and made persistent as history PM data records
 * at the end of the granularity period. Currently the service-oriented
 * framework only specifies the TCA-related part of PM. The definition
 * of the PM data record structure and PM operations (i.e., a
 * service-oriented façade for PM) is for further study.</p>
 *
 * <p>The combination of the objectName, layerRate, granularity,
 * pmParameterName, pmLocation, and thresholdType fields must uniquely
 * identify a TCA. Together, these six fields contain information
 * such that any clear of a TCA would correlate to the correct TCA.
 * This means that if a TCA is raised, cleared, raised again, and
 * cleared again, and if the original clear and second TCA were missed,
 * the second clear would clear the first TCA.</p>
 *
 * <p>The comment OPTIONAL means that the record member need not be
 * present in the corresponding OMG Structured Event.</p>
 *
 * <p>When mapping the attribute TCAId to the filterable event body of
 * the OMG Structured Event (see Event_T), each record member of TCAId_T
 * shall be mapped to a separate filterable data field.</p>
 *
 * <p>For some type declarations their unnamed equivalents are used.
 * This is justified for the sake of readability and since TCA_T is
 * mapped to structured events anyway.</p>
 **/
 struct TCA_T
 {
    EventType_T eventType;
    string eventName;
    string notificationId;
    TCAId_T TCAId;
    ObjectType_T objectType;
    ObjectTypeQualifier_T objectTypeQualifier;
    globaldefs::Time_T emsTime;
    globaldefs::Time_T neTime;
    boolean edgePointRelated;
    string nativeEMSName; // OPTIONAL
    PerceivedSeverity_T perceivedSeverity;
    float value; // OPTIONAL
    string unit; // OPTIONAL
    string additionalText; // OPTIONAL
    boolean isClearable;
    AcknowledgeIndication_T acknowledgeIndication; // OPTIONAL
 };


/**
 * <p>Used to distinguish TCA from alarm.</p>
 **/
 enum AlarmTypeQualifier_T
 {
```

```
      ALARM,
      TCA
   };

  /**
   * <p>Structure used for components of a mixed list of alarm and TCA
   * identifiers. The structure is switched on AlarmTypeQualifier_T.
   * The contents is either an alarm ID or a TCA ID.</p>
   **/
   union AlarmOrTCAIdentifier_T switch (AlarmTypeQualifier_T)
   {
      case ALARM: AlarmId_T alarmId;
      case TCA: TCAId_T tcaId;
   };

  /**
   * <p>Sequence of identifiers for alarms and TCAs. This allows to
   * uniformly treat mixed lists of alarm and TCA identifiers, for example
   * as operation parameters for acknowledge and unacknowledge purposes.</p>
   **/
  typedef sequence<AlarmOrTCAIdentifier_T> AlarmAndTCAIDList_T;


/**
```

A.4.3 Event iterator interface

```
*/

  /**
   * <p>In order to allow the NMS to deal with retrieval of a large number
   * of events, iterators are used for bulk retrieval of notifications.</p>
   *
   * <p>Refer to clause 9.4 "Iterator interfaces" of Rec. X.780.2
   * for information on how iterators are used in this interface.</p>
   **/
   interface EventIterator_I
   {
   boolean next_n(
            in unsigned long how_many,
            out EventList_T eventList)
           raises(globaldefs::ProcessingFailureException);

   unsigned long getLength()
                raises(globaldefs::ProcessingFailureException);

   void destroy()
        raises(globaldefs::ProcessingFailureException);

   };  // end of interface

};  // end of module


#endif  // end of #ifndef ITUT_X780_2_IDL
```

Bibliography

The following references contain information that was used in the development of the framework for CORBA-based service-oriented TMN interfaces.

[b-ITU-T X.731] ITU-T Recommendation X.731 (1992), *Information technology – Open Systems Interconnection – Systems management: State management*.

[b-af-nm-0020.000] ATM Forum af-nm-0020.000 (1994), *M4 Interface Requirements and Logical MIB*.

[b-ISO/IEC 646] ISO/IEC 646-1991, *Information Technology – ISO 7-bit Coded Character Set for Information Interchange*.

[b-ISO/IEC 8859] ISO/IEC 8859-series (1999-2003), *Information Technology – 8-bit Single-Byte Coded Graphic Character Sets*.

[b-OMG 98-06-03] OMG Document ab/98-06-03, *OMG IDL Style Guide*.

[b-Siemens] Siemens AG (2003), *Using CORBA for MTNM*, TM Forum MTNM contribution, http://www.tmforum.org/browse.asp?catID=2014.

[b-TMF layers] TM Forum TMF814 Version 3.0 (2004), *Multi-Technology Network Management (MTNM) NML-EML Interface: CORBA IDL Solution Set, Supporting Document "Functional modelling concepts"*, file "layers.pdf".

[b-GAMMA] GAMMA (E.), HELM (R.), JOHNSON (R.), VLISSIDES (J.): Design Patterns: Elements of Reusable Object-Oriented Software, *Addison-Wesley*, August 1994, ISBN 0-201-63361-2.

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	**Data networks, open system communications and security**
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2008