INTERNATIONAL TELECOMMUNICATION UNION

ITU-T Q.816.1

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(08/2001)

SERIES Q: SWITCHING AND SIGNALLING

Q3 interface

CORBA-based TMN services: Extensions to support coarse-grained interfaces

ITU-T Recommendation Q.816.1

ITU-T Q-SERIES RECOMMENDATIONS

SWITCHING AND SIGNALLING

SIGNALLING IN THE INTERNATIONAL MANUAL SERVICE	Q.1–Q.3
INTERNATIONAL AUTOMATIC AND SEMI-AUTOMATIC WORKING	Q.4–Q.59
FUNCTIONS AND INFORMATION FLOWS FOR SERVICES IN THE ISDN	Q.60–Q.99
CLAUSES APPLICABLE TO ITU-T STANDARD SYSTEMS	Q.100–Q.119
SPECIFICATIONS OF SIGNALLING SYSTEMS No. 4 AND No. 5	Q.120–Q.249
SPECIFICATIONS OF SIGNALLING SYSTEM No. 6	Q.250–Q.309
SPECIFICATIONS OF SIGNALLING SYSTEM R1	Q.310–Q.399
SPECIFICATIONS OF SIGNALLING SYSTEM R2	Q.400–Q.499
DIGITAL EXCHANGES	Q.500–Q.599
INTERWORKING OF SIGNALLING SYSTEMS	Q.600–Q.699
SPECIFICATIONS OF SIGNALLING SYSTEM No. 7	Q.700–Q.799
General	Q.700
Message transfer part (MTP)	Q.701–Q.709
Signalling connection control part (SCCP)	Q.711–Q.719
Telephone user part (TUP)	Q.720–Q.729
ISDN supplementary services	Q.730–Q.739
Data user part	Q.740–Q.749
Signalling System No. 7 management	Q.750–Q.759
ISDN user part	Q.760–Q.769
Transaction capabilities application part	Q.770–Q.779
Test specification	Q.780–Q.799
Q3 INTERFACE	**Q.800–Q.849**
DIGITAL SUBSCRIBER SIGNALLING SYSTEM No. 1	Q.850–Q.999
PUBLIC LAND MOBILE NETWORK	Q.1000–Q.1099
INTERWORKING WITH SATELLITE MOBILE SYSTEMS	Q.1100–Q.1199
INTELLIGENT NETWORK	Q.1200–Q.1699
SIGNALLING REQUIREMENTS AND PROTOCOLS FOR IMT-2000	Q.1700–Q.1799
BROADBAND ISDN	Q.2000–Q.2999

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

ITU-T Recommendation Q.816.1

CORBA-based TMN services: Extensions to support coarse-grained interfaces

Summary

This Recommendation defines extensions to the set of TMN CORBA Services required to support coarse-grained interfaces. It specifies how CORBA Common Object Services are used to support coarse-grained interfaces, and defines extensions to the TMN-specific support services defined in ITU-T Q.816. A CORBA IDL module defining the interfaces to the new TMN-specific support services is provided.

Source

ITU-T Recommendation Q.816.1 was prepared by ITU-T Study Group 4 (2001-2004) and approved under the WTSA Resolution 1 procedure on 13 August 2001.

Keywords

Coarse-grained, Common Object Request Broker Architecture (CORBA), CORBA Services, Distributed Processing, Interface Definition Language (IDL), Managed Objects, TMN Interfaces.

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.

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, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

ITU-T Recommendation Q.816.1

CORBA-based TMN services: Extensions to support coarse-grained interfaces

1 Scope

The TMN architecture defined in ITU-T M.3010 (2000) introduces concepts from distributed processing and includes the use of multiple management protocols. ITU-T Q.816 and 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.

This Recommendation, along with ITU-T X.780.1, adds specifications to the framework to enable it to support a slightly different style of interaction between managing systems and managed systems than that specified in the original framework documents. This style of interaction has certain benefits, the main one being that it relieves a managing system from having to retrieve an object-oriented software address for each manageable resource it wishes to access. These software addresses could number in the millions on large systems. It also changes somewhat the way software is structured on the managed systems, which some managed system suppliers may prefer.

The scope of this Recommendation is the same as the original TMN CORBA framework. The framework and these extensions cover all interfaces in the TMN where CORBA may be used. 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) as well as between management systems and network elements.

1.1 Purpose

The purpose of this Recommendation is to extend the TMN CORBA framework to enable it to be used in a wider range of applications. The extensions enable a slightly different mode of interaction between the managing and managed systems which may be preferred in many situations. Thus, this Recommendation is intended for use by various groups specifying network management interfaces.

1.2 Application

The approach taken in the CORBA TMN framework Recommendations is to model manageable network resources as software objects accessible using CORBA. Information models written in the CORBA Interface Definition Language (IDL) describe the object interfaces.

CORBA provides location-transparency, enabling one software object to interact with another regardless of its location. A software object is accessed using what CORBA refers to as an Interoperable Object Reference (IOR).

The original CORBA TMN framework models each manageable resource as an independent CORBA object, each with its own unique IOR. This approach flexibly allows each object to reside anywhere. It does, however, require that managing systems have on hand an IOR for each object they wish to access. This is a burden that many companies and administrations in the telecommunications industry have sought to avoid. It also could require a managed system to support large numbers of IORs, which some managed system suppliers would like to avoid. This Recommendation, along with ITU-T X.780.1, defines how the TMN CORBA framework is to be extended to avoid the need for large numbers of IORs.

CORBA-based interfaces using the approach where each manageable resource is addressable with a unique IOR have become known as "fine-grained" interfaces. Alternatively, those where an IOR is not assigned to each manageable resource are known as "coarse-grained" interfaces.

Because X.780.1 defines a slightly different approach to modelling manageable resources on coarse-grained interfaces, interface model specifications will be slightly different for the fine-grained and coarse-grained approaches.

1.3 Recommendation roadmap

This Recommendation has the following structure:

Clause 1	Introduction, roadmap and updates.
Clause 2	References.
Clauses 3 and 4	Definitions of terms and abbreviations used throughout this Recommendation.
Clause 6	Design considerations that must be addressed as support for coarse-grained interfaces is added to the framework.
Clause 7	TMN CORBA framework and coarse-grained requirements overview.
Clause 8	Requirements on the use of OMG Common Object Services to support coarse-grained network management interfaces.
Clause 9	Requirements on the use of TMN-specific object services to support coarse-grained network management interfaces. The original framework defined some new services that will have to be extended to support coarse-grained interfaces. Also, a new service is required.
Clause 10	Compliance and conformance guidelines.
Annex A	TMN-specific coarse-grained support service IDL.

2 Normative references

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

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

[2] ITU-T X.780 (2001), *Guidelines for defining CORBA managed objects*.

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

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

[5] OMG Document formal/01-02-65, *Naming Service Specification*.

[6] OMG Document formal/00-06-20, *Notification Service Specification*, Version 1.0.

[7] OMG Document formal/00-01-04, *Telecom Log Service Specification*, Version 1.0.

[8] OMG Document formal/00-06-25, *Security Services Specification*, Version 1.5.

[9] OMG Document formal/00-06-28, *Transaction Service Specification*, Version 1.1.

[10] OMG TC Document orbos/98-05-05, *CORBA Messaging*.

[11] IETF RFC 2246 (1999), *The TLS Protocol Version 1.0*.

3 Definitions

3.1 Definitions from ITU-T X.701

The following terms used in this Recommendation are defined in ITU-T X.701:

– managed object class;

– manager;

– agent.

3.2 Definitions from ITU-T X.703

The following term used in this Recommendation is defined in ITU-T X.703:

– notification.

3.3 Definitions from ITU-T Q.816

The following term used in this Recommendation is defined in ITU-T Q.816:

– event channel.

4 Abbreviations

This Recommendation uses the following abbreviations:

AMI	Asynchronous Messaging Invocation
API	Application Programming Interface
CMIP	Common Management Information Protocol
CORBA	Common Object Request Broker Architecture
COS	Common Object Services
DN	Distinguished Name
EMS	Element Management System
GDMO	Guidelines for the Definition of Managed Objects
ID	Identifier
IDL	Interface Definition Language
IIOP	Internet Interoperability Protocol
IOR	Interoperable Object Reference
ITU-T	International Telecommunication Union – Telecommunication Standardization Sector
MO	Managed Object
MOO	Multiple Object Operation
NE	Network Element
NMS	Network Management System
OAM&P	Operations, Administration, Maintenance and Provisioning

OID	Object Identifier
OMG	Object Management Group
ORB	Object Request Broker
OSI	Open Systems Interconnection
PDU	Protocol Data Unit
PM	Performance Management
POA	Portable Object Adapter
QoS	Quality of Service
RDN	Relative Distinguished Name
SSL	Secure Socket Layer
TII	Time-Independent Invocation
TLS	Transport Layer Security
TMN	Telecommunications Management Network

5 Conventions

5.1 Recommendation 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) OPTION-1 An example optional requirement.

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

Many examples of CORBA IDL are included in this Recommendation, and IDL specifying the TMN specific services, and supporting data types, included in Annex A. The IDL is written in a 9-point courier typeface:

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

5.2 Compiling the IDL

An advantage of using IDL to specify network management interfaces is that IDL can be "compiled" into programming code by tools that accompany an ORB. This actually automates the development of some of the code necessary to enable network management applications to interoperate. This Recommendation has an annex that contains source code that implementers will want to extract and compile. Annex A is normative and should be used by developers implementing systems that conform to this Recommendation. The IDL in this Recommendation has been checked with two compilers to ensure its correctness. A compiler supporting the CORBA version specified in ITU-T Q.816 must be used.

Annex A has been formatted to make it simple to cut and paste into plain text files that may then be compiled. Below are tips on how to do this.

1) Cutting and pasting seems to work better from the Microsoft® Word® version of this Recommendation. Cutting and pasting from the Adobe® Acrobat® file format seems to include page headers and footers, which cannot be compiled.

2) All of Annex A, beginning with the line "/* This IDL code…" through the end should be stored in a file named "itut_q816_1.idl" in a directory where it will be found by the IDL compiler.

3) The headings embedded in the annex need not be removed. They have been encapsulated in IDL comments and will be ignored by the compiler.

4) Comments that begin with the special sequence "/**" are recognized by compilers that convert IDL to HTML. These comments often have special formatting instructions for these compilers. Those that will be working with the IDL may want to generate HTML as the resulting HTML files have links that make for quick navigation through the files.

5) The annex has been formatted with tab spaces at 8-space intervals and hard line feeds that should enable almost any text editor to work with the text.

6 Coarse-grained interface design considerations

This clause identifies several design considerations that must be addressed by the framework as support for coarse-grained interfaces is added.

6.1 Reduced number of IORs

As support for coarse-grained interfaces is added to the framework, the framework must enable the number of managed resources (like termination points) to grow without increasing the number of IORs exposed across the management interface.

6.2 Ability to derive IORs

A lot of the early debate over the merits of coarse-grained interfaces focused on the need to reduce the number of IORs supported by a managed system. This was because throughout most of the 1990s Object Request Brokers (ORBs) offered system developers no standard way to persistently store the state of an object between method invocations. Thus, all objects with outstanding IORs had to be kept in memory, which limited the number of IORs a system could support. The OMG remedied this with the CORBA 2.2 specification, however, and most ORBs now support the OMG's Portable Object Adapter (POA) standard, which enables systems to persistently store object state between method invocations. Thus, the number of objects a managed system can now support is basically limited only by the number of objects it can store in its disk space.

This, however, does not negate all the benefits of coarse-grained interfaces. The benefactor, it turns out, will be mainly the managing system. When a managing system uses a fine-grained CORBA interface in order to interact with each managed resource, it must at some time retrieve the IOR for each managed resource. In a system with millions of managed resources, this amounts to millions of IORs. Managing systems will probably implement a range of strategies for dealing with large numbers of IORs. The simplest will be to just resolve a managed resource's name to its IOR before each interaction, but this is slow and wasteful of data communications network resources. An alternative will be to retrieve all names and their paired IORs once, and store them on the managing system. Another strategy will be to cache names and IORs on the managing system, keeping the most recently used IORs on hand for quick reference while discarding those not used for some time to be retrieved from the managed system when needed again. Other strategies could also emerge.

Coarse-grained interfaces have the potential to relieve managing systems from having to implement such schemes. Coarse-grained interfaces could enable a managing system to initially retrieve and store just a small number of IORs from a managed system. To make this work, though, a managing system must subsequently be able to tell from just a managed resource's name which of the previously-retrieved IORs to use to interact with that resource. That is, given a name, a managing system must be able to derive the IOR of the interface for that specific managed resource. If instead the managing system has to query the managed system with the name in order to discover the corresponding IOR, then the managing system is back to implementing the schemes described above and the benefit of adding support for coarse-grained interfaces to the framework is largely lost.

6.3 Application of framework services

Support for coarse-grained interfaces must be added to the framework in a way that enables the existing framework services (such as the Multiple Object Operation service and Terminator service) to be applied. This does not preclude changes to the implementations of those services to support coarse-grained interfaces.

6.4 Distinguishing between the two types of objects

It must be possible for a managing system to distinguish between resources managed with fine-grained and coarse-grained interfaces.

6.5 Hierarchical naming

Hierarchical (containment-based) naming of resources managed with coarse-grained interfaces must be supported.

6.6 Migration of modelled entities across approaches

The framework must enable implementations that can migrate access to a particular type of managed resource from the coarse-grained approach to the fine-grained approach, or vice versa.

6.7 Migration of modelled entities across technologies

The framework must enable implementations to migrate to various software technologies, such as C++ or JavaBeans.

6.8 Equivalent distinguished names

The distinguished name of a managed resource must not depend on whether the resource is accessed with a fine-grained or coarse-grained interface. Also, it must be possible to retrieve containment relationship information from a single location regardless of whether the resources on the managed system are accessed with fine-grained or coarse-grained interfaces.

7 Framework and requirements overview

Clause 6 outlined the design considerations that must be resolved as support for coarse-grained interfaces is added the framework. This clause and the rest of this Recommendation provide the details on how the framework will be extended to address these issues. ITU-T X.780.1 describes how coarse-grained interfaces will be modelled. First, a brief overview of the current framework is presented, then an overview of the extensions.

7.1 Framework overview

The framework for CORBA-based TMN interfaces is a collection of capabilities. A central piece of the framework is a set of OMG Common Object Services. The framework defines their role in network management interfaces, and defines conventions for their use. The framework also defines support services that have not been standardized as OMG Common Object Services, but are expected to be standard on network management interfaces conforming to the framework.

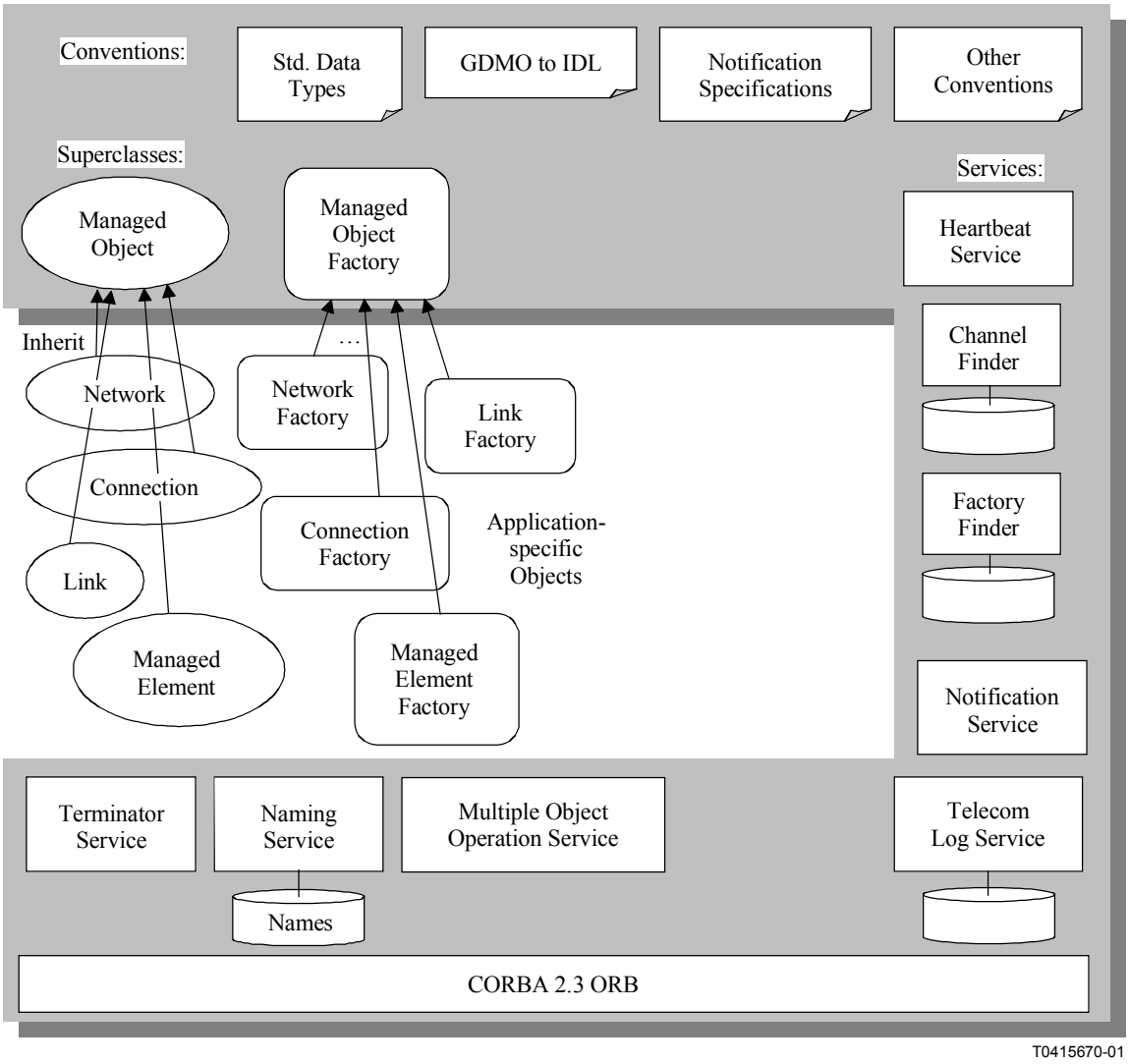

Figure 1/Q.816.1 – Overview of framework

The framework is depicted graphically in Figure 1 above. The figure shows the framework in grey. In the middle are the application-specific objects that are supported by the framework. Along the bottom is a box representing the CORBA ORB. Above that are a number of boxes with names in them representing the services that compose the framework. (Some also have icons depicting the databases they would have to maintain to perform their functions.) These services, along with ORB version requirements, are defined in ITU-T Q.816. Along the top of the figure are icons representing two superclasses, one for managed objects and one for managed object factories. Each of the managed objects and managed object factories supported by this framework must ultimately inherit from these superclasses, respectively. Also shown on the figure are icons of pages with up-turned corners representing standard object modelling conventions. These conventions and the superclasses are defined in ITU-T X.780.

7.2 Coarse-grained extensions overview

This clause provides an overview of the extensions to the framework required to support coarse-grained interfaces.

7.2.1 The facade design pattern

The most significant change to the framework required to support coarse-grained interfaces is the way managed objects are accessed. The number of managed objects on a managed system must be able to go up while the number of IORs supported by the system does not. It is still desirable, though, that access to the managed objects remain strongly-typed. This leads to the use of a design pattern referred to here as the "facade" pattern. A facade can be thought of as a false front, or as a portal. Using the facade design pattern, a managed system will support a small number of facade 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 invoke an operation on a managed object by actually invoking the operation on a facade for that type of managed object on that system. In the facade 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 facade approach does not need to implement the fine-grained managed object interfaces.

It is best to think of a facade not as a managed object, but as an intermediary object that enables a managing system to access managed objects. The facade object has a CORBA interface and is accessible using CORBA. The managed objects, however, may not have CORBA interfaces and might not be directly accessible using CORBA. The facade itself does not represent a manageable network resource; its purpose is to enable interaction with the objects that do represent manageable resources. All facade objects are created automatically by the managed system, and exist as long as the managed objects are accessible through the facade. Multiple facades for the same type of managed objects may exist on a coarse-grained interface, but a managed object shall be accessible through only 1 facade. See Figure 2 below.

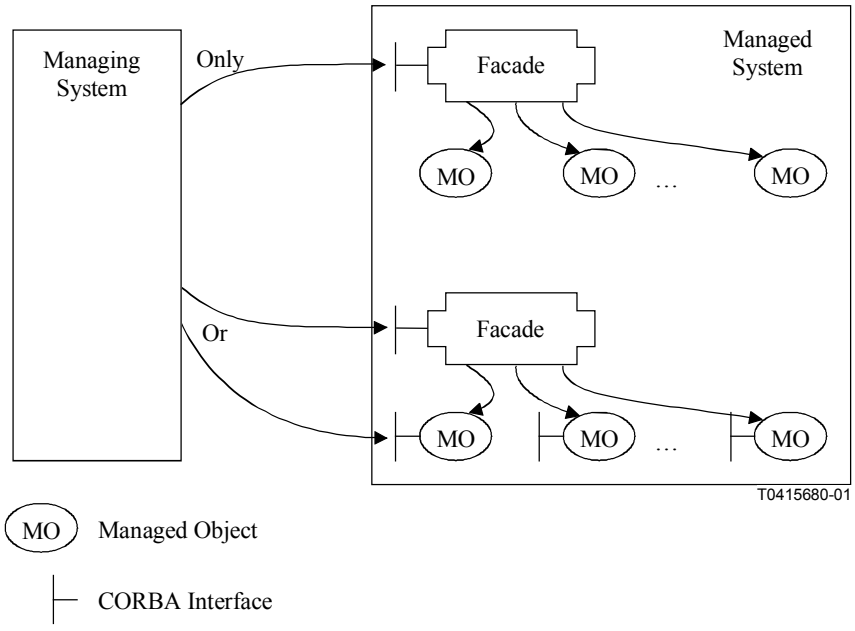

Figure 2/Q.816.1 – The facade role

The figure shows a managing system accessing a managed system that supports the coarse-grained approach. The managed system has two facade interfaces 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 facade. The managed objects at the bottom also support direct CORBA interfaces and can be accessed either through the facade or directly. Direct CORBA access is optional, but a managed system that supports the facade approach must provide facade interfaces for each of its managed object instances.

A facade may use a managed object's CORBA interface to invoke an operation on it, or some other implementation-specific means. A managed system, in fact, need not even implement managed objects as individual objects internally. By implementing an interface based on this framework, however, it will give the illusion that managed objects are internally implemented as objects.

When an operation is invoked on a managed object through a facade, the facade must then invoke the operation on the actual managed object or entity. Because many managed objects will be accessed through a single facade, the facade 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 facade operation directed at a managed object.

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.

7.2.2 Managed object name extension

As mentioned above, managed objects accessed through a facade will still have a name even though they may not have an individual CORBA interface. It is important that a managing system be able to determine which facade to use based on the managed object's name. If it cannot it will have to query the managed system or persistently associate a facade IOR with every managed object's name. To support the ability to determine a managed object's facade based on only its name, the names of managed objects accessible through a facade are extended slightly beyond the names of managed objects not accessible through a facade. In the final name component, which always has an *ID* string with the value "Object" (or, as extended by this Recommendation, <empty>), the *kind* string is set to the value of a facade identifier assigned to the facade through which the object may be accessed. For managed objects not accessible through a facade, this *kind* string is empty. Clause 8.1.2 provides additional details on how the facade ID is used to identify a facade.

7.2.3 Support services for facade-accessible managed objects

The framework support services provided on interfaces that use the facade approach will be largely the same as those defined in ITU-T Q.816. Some, such as the Factory Finder and Channel Finder services, require no change at all. Others, such as the Terminator and Multiple Object Operation (MOO) services, require no changes to their interfaces or the way they are used by managing systems, but may require slight changes to their implementations if they access managed objects using the managed objects' facade interfaces (rather than some implementation-specific method). Clause 9 provides details on the framework support service changes required to support coarse-grained interfaces.

The biggest change to the support services comes in the area of support for naming. The facade interfaces are bound to names in the naming service, much the same way the support service interfaces are. On interfaces that use facades, however, the managed objects' names are not required to be bound to IORs in the OMG Naming Service. Instead, a new service is introduced as a place to store managed object names and containment relationship information. This new service, the Containment Service, is defined in 9.6. The Containment Service must be supported on systems that use managed object facades, but will not be required on systems that do not use facades.

7.2.4 Facade modelling

To support the facade design pattern and the definition of facades usable with this framework, a new base interface is introduced. This interface will be known as the Managed Object Facade interface. It plays the same role in coarse-grained interfaces as the Managed Object interface does in fine-grained interfaces. That is, it is the base interface from which all managed object facade interfaces must either directly or indirectly inherit to work with the framework. The Managed Object Facade interface is quite similar to the Managed Object interface defined in ITU-T X.780. See ITU-T X.780.1 for the definition of the Managed Object Facade interface, along with other coarse-grained information modelling guidelines.

The changes to the framework are reflected in Figure 3 below. A new superclass, *ManagedObjectFacade*, is added to the figure. Also, the Containment Service is added. Note that it provides access to a database of managed object names. The managed object name database maintained by the Naming Service is shown with dotted lines, indicating that it need not store the names and IORs of managed objects. The Naming Service is still required, however, to enable managing systems to find facade interfaces and support service references. Finally, the managed objects are also shown drawn with dotted lines, to indicate that they need not be directly accessible.

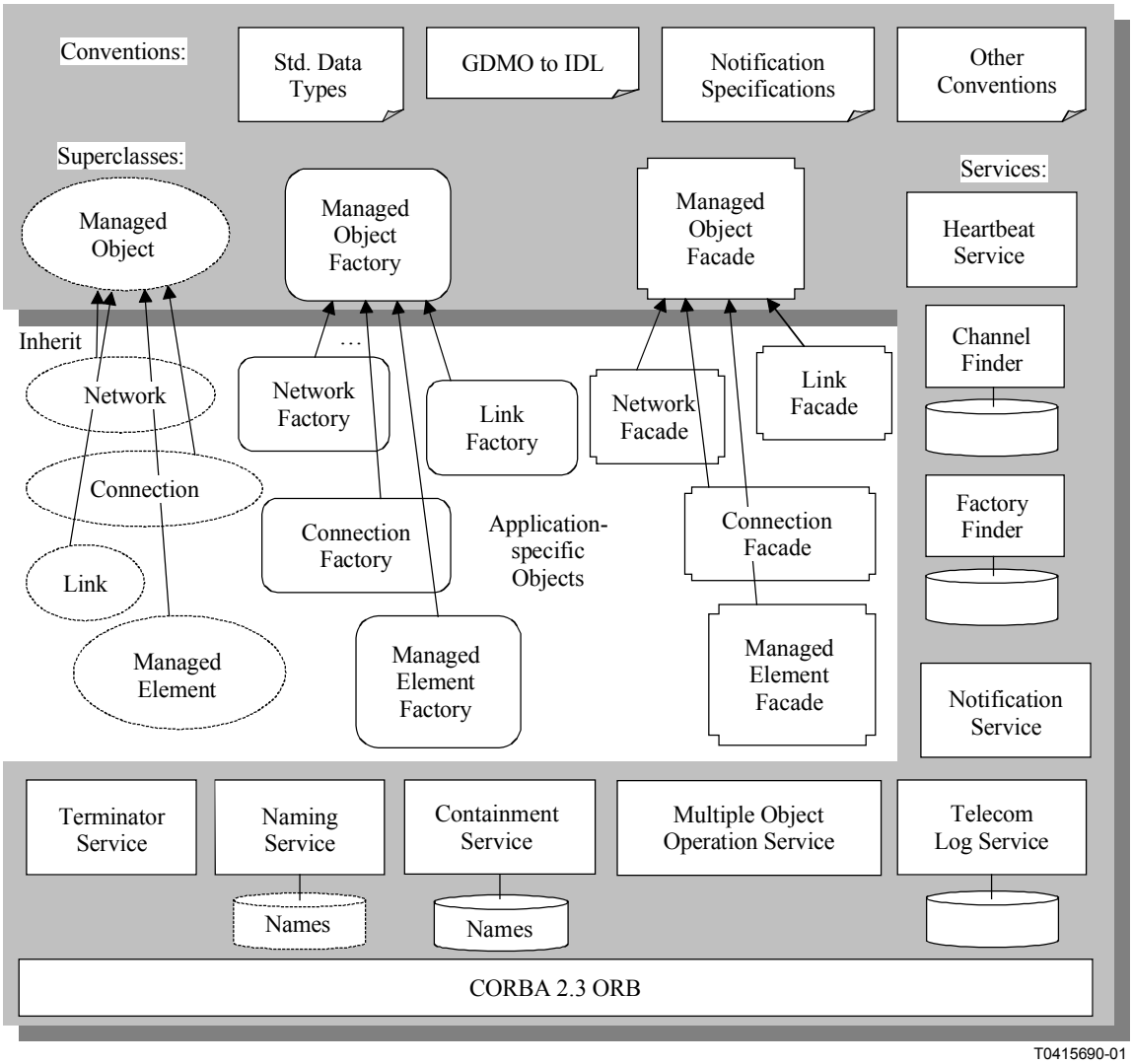

Figure 3/Q.816.1 – Framework with extensions to support coarse-grained interfaces

8 Framework common object services usage requirements for supporting coarse-grained interfaces

ITU-T Q.816 describes how the original framework includes several of the OMG's Common Object Services. These are services defined by the OMG for use in generally any CORBA application. The framework defines which of the OMG's Common Object Service must be supported by a managed system, and conventions on their use. This clause provides additional conventions and requirements needed to support coarse-grained interfaces for each of the Common Object Services in the framework.

8.1 The naming service

This clause describes both how the OMG Naming Service is used on coarse-grained interfaces, and the extension to managed object names required to support facades.

8.1.1 Naming service use on coarse-grained interfaces

The role of the Naming Service can be greatly reduced on coarse-grained interfaces. On fine-grained interfaces, managed object names and IORs are stored in the Naming Service. On coarse-grained interfaces, managed objects are not required to expose IORs, and name bindings for them are not required to be stored in the Naming Service.

The concept of local root naming contexts still applies. Recall that a local root naming context contains name bindings for one or more objects that each form the root of a tree of managed objects related by containment. A local root naming context itself has a unique name, and all the objects under it are named relative to it. A local root naming context also contains bindings for the framework services supporting the managed objects named relative to that local root.

Name bindings for the managed objects do not have to go in the Naming Service, but bindings for the facade interfaces do. Facade interfaces do have IORs, and there is a relatively small number of them, so name bindings for them are simply placed in the local root naming context along with the bindings for the support services. The format of their names is defined below.

The result of these rules is that, on a managed system with a coarse-grained interface, the Naming Service may contain only local root naming contexts. Since many systems will have only a single local root naming context, those systems will have only a single Naming Service naming context object. This context object will contain just name bindings for the support services and the managed object facade interfaces implemented by the system.

8.1.2 Facade interface names

As mentioned above, facade interfaces have IORs, and names will be bound to these IORs in the root naming context. This clause describes the format of the names bound to the facade interface IORs.

Since the names for all of the facades on a system are placed in the same naming context, they must be unique with respect to each other. Also, it is helpful if a managing system can recognize a name as one belonging to a facade, distinct from the support service names also bound in the root naming context. Finally, as described in 6.2, it must be possible for a managing system to determine which facade to use based on the name of a managed object. To meet this last requirement, some coordination between facade names and managed object names is required.

When an IOR is bound to a name in an OMG naming context, the name consists of two strings, an *ID* string and a *kind* string. All facade interfaces shall be bound with the *ID* string equal to "Facade" and the *kind* string set to a value unique among all the facade names on that system. This value will be referred to as a "facade ID." The facade ID goes in the *kind* field rather than the *ID* field, though, because later it will be seen that it must be matched with the value in the *kind* string in a managed object's name. Matching a *kind* value to another *kind* value was felt to be less confusing than putting an ID in a *kind* field.

Since the facade names will all have the same *ID* value, "Facade", then the *kind* strings must be unique to make the name binding unique, as required by the OMG Naming Service. The value of the *kind* string shall be chosen by the managed system implementer. The actual value is not important, but as described below, each managed object accessed through that facade will have the same *kind* string value in the final component of its name.

8.1.3 Managed object name extension

Managed objects accessible through a facade will have names identical to those that are not accessible through a facade except for the values of the *ID* and *kind* strings in the name's final component. ITU-T Q.816 states that managed object names shall have an "extra" name component appended to their fully-distinguished names. This final name component is required to represent containment relationships in the OMG Naming Service. Q.816 requires that the *ID* string in this final name component have a value of "Object" and that the *kind* string be null. Extending the framework to support coarse-grained interfaces changes these requirements for objects accessible through a facade.

Managed objects that are accessible through a facade shall have a final name component with the *kind* string set to the value of the *kind* string in the name binding of the facade that is used to access that object. That is, the *kind* string is set to the facade ID chosen for that facade. See 8.1.2 above for details on facade names. Managed objects that are accessible only through a facade shall have a null *ID* string value in the final component of their names. Managed objects that are accessible both directly and through a facade shall have an *ID* value of "Object" and a *kind* value equal to the facade id in their final name component.

To illustrate how managed object names can be resolved to a facade IOR, consider an EMS owned by the carrier XYZ Communications. This EMS might have a root naming context with the globally unique name "ems17.ems.xyz.com". Let us say the circuit pack facade on that system is bound to the name "Facade.cp" in the root naming context, where "Facade" is the value of the *ID* string and "cp" is the Facade ID placed in the *kind* string. Now, an instance of a circuit pack managed object might have the following name:

```
ems17\.ems\.xyz\.com/me1.ManagedElement/bay1.Equipment/shelf1.Equipment/
slot1.Equipment/cp1.Equipment/.cp
```

(When CORBA names, which consist of a sequence of data structures, each containing a string named "ID" and a string named "Kind", are represented in string form, the OMG Naming Service Specification [5] specifies that they be represented with the following format:

$ID_1.Kind_1/ID_2.Kind_2/ID_3.Kind_3\ldots$

Where ID_1 represents the value of the *ID* string in first component of the name, etc. Thus, a slash ('/') separates the values of one component from the values of the next, and a period ('.') separates the value of the *ID* string in each component from the *kind* string. The *ID* string comes before the *kind* string, and if the *kind* string is empty the period is eliminated. If either an *ID* string or a *kind* string value has a period, slash, or backslash ('\') in it, that character is escaped by preceding it with a backslash. This is the reason for the backslashes preceding the periods in the name above.)

Given this name, a managing system can determine that the object is accessible through a facade because the value of the *kind* string in the final component is not empty. Next, the managing system can find the IOR of the facade for that object by performing two steps. First, the managing system must compare the name with the root contexts that have been registered with it to find the best match. In this case, that will be the root context named "ems17\.ems\.xyz\.com". Once it has determined this, it knows the name of the facade for the object is "ems17\.ems\.xyz\.com/Facade.cp". It knows this because all of the facades on that system are bound to that root context, they all have an *ID* string with the value "Facade", and the *kind* string must match the *kind* string value in the final component of the managed object's name, "cp". Assuming the managing system caches the IORs for

the facades, it can go ahead and invoke an operation on the object using the facade IOR and the object's name. Notice that the managing system does not have to retrieve or cache a large number of IORs, or query the managed system for the IOR before each operation.

Since the name of an object depends to some degree upon how it is accessed, changing the way an object is accessed has the side effect of requiring changes to its name. So, for example, if an object that has previously been accessible only through a facade is made accessible directly, likely through a software upgrade, that software upgrade must include updating the object's name. Also, if an object that had been accessible through one facade is moved to another, its name must change. Changing an object's name might result in the need to delete it and re-create it. Because of this level of difficulty, changing the way an object is accessed is expected to occur infrequently.

8.1.4 Object name comparisons

Table 1 compares the how the values of names for different kinds of interfaces are constructed.

Table 1/Q.816.1 – Object name comparisons

Type of interface	Name bound in root naming context?	2nd to last name component		Last name component	
		ID	Kind	ID	Kind
Support Service	Always	Not applicable – only 1 component used.	Not applicable – only 1 component used.	Unique value picked by managed system, such as "ChannelFinder1"	The fully-scoped interface name of the service, such as "itut_q816:: ChannelFinder"
Facade	Always	Same as above.	Same as above.	"Facade"	A value unique among the facade name bindings in the root context.
Managed Object not Accessible through a Facade	The top-most objects will have a naming context bound to the root context.	Application-dependent value chosen to name the object relative to its parent – the "RDN."	The value of the *kind* constant string in the naming module referred to when the object is created.	"Object"	<empty>

Table 1/Q.816.1 – Object name comparisons

Type of interface	Name bound in root naming context?	2nd to last name component		Last name component	
		ID	Kind	ID	Kind
Managed Object Accessible through a Facade only	Not required to have a name in Naming Service, but top-most objects will be named directly subordinate to root naming context.	Same as above.	Same as above.	<empty>	The value chosen by the managed system for the *kind* string used in the name binding for the facade through which this object is accessed. See cell explaining *kind* string values for facades.
Managed Object Accessible through a Facade as well as directly	Same as above.	Same as above.	Same as above.	"Object"	Same as above.

8.1.5 Storing managed object names in the naming service

While not required, some systems that utilize facades may still choose to store some or all managed object names in the OMG Naming Service. This may be because some or all of the objects expose direct CORBA interfaces, and hence have IORs that can be bound to their names.

If a managed object has an IOR bound to its name in the Naming Service and it is also accessible through a facade, its name binding in the naming service will have an *ID* of "Object" and a *kind* string set to the facade ID for its facade.

A system that stores only some of its managed object names in the Naming Service may have to deal with the problem of binding an IOR to a managed object name in the naming tree below a managed object that does not have an IOR. Note that typically this will not happen, as the objects at the "leaves" of the naming tree tend to be the most numerous and are therefore the most likely to not have individual IORs. The objects interior to the tree tend to be less numerous and are therefore more likely to have IORs when a mixed fine and coarse-grained implementation is chosen. If this situation arises, however, it shall be handled by creating a naming context for the superior object without an IOR bound to the ID value "Object". Below that the managed system shall create a context for the new managed object, and bind its IOR to the ID value "Object" in that context. The *kind* string shall be set the value of the *kind* string bound to the facade used to access the object.

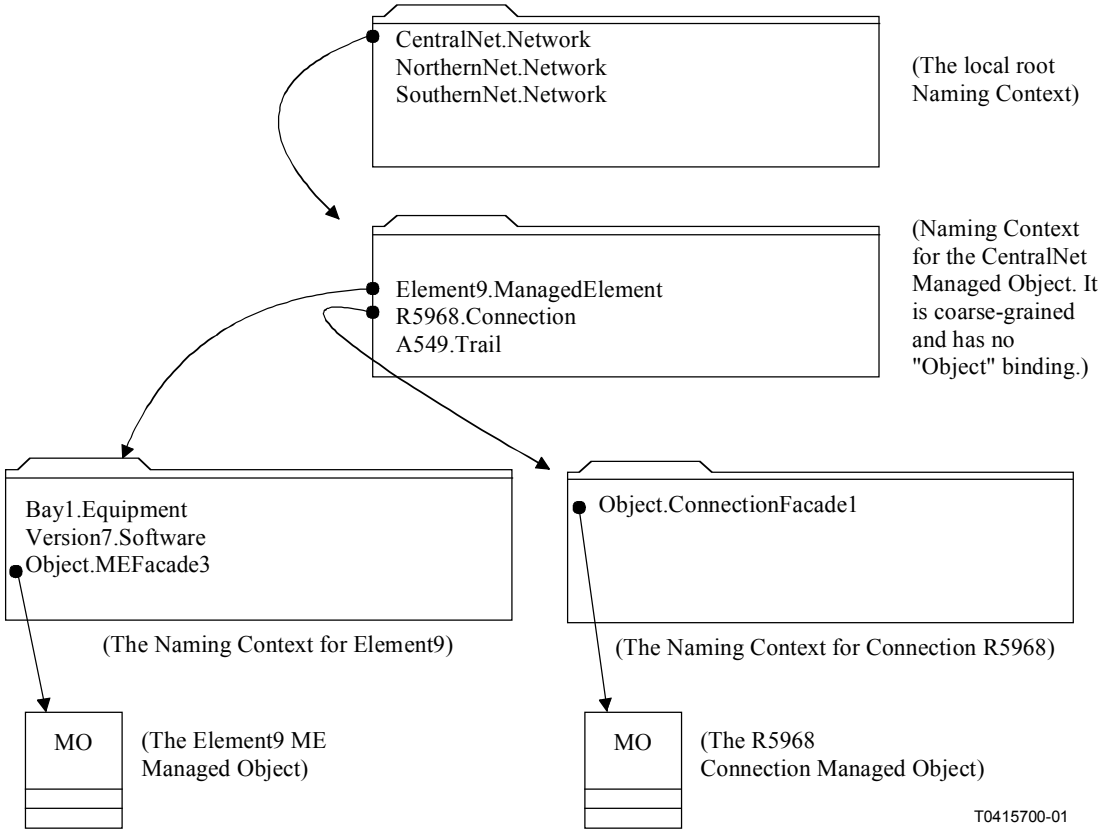

Figure 4/Q.816.1 – Naming graph combining fine and coarse-grained objects

In this example, the ManagedElement object name "Element9" and the Connection object named "R5968" are both contained by the Network object named "CentralNet". CentralNet is accessible through a facade, but not directly. Both Element9 and R5968 are accessible both directly and through facades. To represent this in the Naming Service, a naming context for CentralNet is created (it is drawn as the second folder from the top) but it has no binding for an IOR to CentralNet, since CentralNet does not have a direct IOR. If CentralNet did have a direct CORBA interface, the CentralNet naming context would contain a binding to it, with an ID value equal to "Object". Both Element9 and R5968 have bindings for their IORs, which are shown at the bottom of the figure.

Since systems that support facades are not required to expose IORs for managed objects or bind names to IORs for managed objects in the Naming Service, a managing system using a coarse-grained interface cannot count on the Naming Service for complete containment relationship information. A managed system may bind none or only some names to IORs in the naming service. So, on interfaces that support facades, managing systems must use the Containment Service to retrieve containment information. The Naming Service can only be used to resolve names to IORs for those objects that have individual IORs.

If the Naming Service is used to store managed object names, it is the responsibility of the managed system to keep the information in the Naming Service accurate and synchronized with the information in the Containment Service. One possible means of accomplishing this would be to implement the two services to access a common database. Other means could also be implemented.

8.1.6 Naming service requirements for coarse-grained interfaces

This clause defines naming service-related requirements managed systems must meet when using coarse-grained interfaces.

(R) NAME-1. A managed system shall bind names to the IORs of each facade used on the system. A name binding for each facade shall be contained in each root naming context where the tree beneath that root contains objects accessed through the facade.

(R) NAME-2. The *ID* string of each facade name shall have the value "Facade". The *kind* string shall have a value chosen by the managed system to be unique among the facade interfaces bound in a particular root naming context.

(R) NAME-3. If stored in the Naming Service, the last name component of the name of a managed object accessible through a facade shall have an *ID* string with the value "Object" and a *kind* string with a value identical to the *kind* string used in the name binding for the facade used to access the managed object.

(R) NAME-4. If the managed system stores some or all managed object names in the Naming Service, the managed system shall ensure the accuracy of those names and the integrity of the naming tree.

(R) NAME-5. If the managed system stores some or all managed object names in the Naming Service, it will use naming contexts with no IOR bound to the *ID* value "Object" for managed objects that do not have individual IORs. These only need to be used when a managed object with an IOR is bound to a name in the naming tree below a managed object without an IOR.

8.2 Notification service

No additional requirements are placed on the use of the Notification Service to support coarse-grained interfaces. The Notification Service can be used to convey notifications from managed objects that are accessible through a facade in the same manner as managed objects that are not.

8.3 Telecom log service

No additional requirements are placed on the use of the Telecom Log Service to support coarse-grained interfaces.

8.4 Messaging service

No additional requirements are placed on the use of the Messaging Service to support coarse-grained interfaces.

8.5 Security service

Using the facade approach, the OMG Security Service can protect only at the level of individual facade instances. If there are requirements to protect managed objects at the instance level using the OMG Security Service, then a fine-grained access approach should be used.

8.6 Transaction service

No additional requirements are placed on the use of the Transaction Service to support coarse-grained interfaces.

9 Framework support services requirements for supporting coarse-grained interfaces

In addition to rules for using OMG Common Object Services, ITU-T Q.816 defines some new support services for use on CORBA TMN interfaces. These services provide common functions specific to network management that are not provided by the general-purpose OMG Common Object Services.

The clauses below define the additional requirements these services must meet to support coarse-grained interfaces. Also, one new service is defined to maintain managed resource containment relationships on coarse-grained interfaces. The IDL describing the interface for this new service is provided in Annex A.

9.1 The factory finder service

The Factory Finder Service is a service defined in ITU-T Q.816 to enable managing systems to find "factories" on managed systems. A managed system creates a new object on a managed system by invoking an operation on a factory, which is itself an object. (This is a commonly-used CORBA design pattern.) A managing system finds a factory by querying this service, provided by the managed system, with the class name of a factory. The service responds with a reference to a factory of that type. The managing system may then invoke the appropriate operation on the factory and create the object. The Factory Finder Service is found by looking it up in the Naming Service. The managed system is required to place a reference to it in the root naming context.

This same approach is used for managed objects that must be accessed through a facade. The only difference is that if the object does not support a direct CORBA interface, the factory will return a nil reference. It will, however, return a suitable name so that the managing system can access the newly created object through its facade.

Because the approach for creating managed objects on coarse-grained interfaces is similar to that for managed objects on fine-grained interfaces, the Factory Finder Service does not need to be modified. Thus, no additional requirements are placed on the use of the Factory Finder Service to support coarse-grained interfaces.

9.2 The channel finder service

The Channel Finder Service is a service defined in ITU-T Q.816 to enable managing systems to discover the event channels present on a managed system, and which notifications each of them handles. A small managed system might have only a single channel, but a more complex system might have multiple channels, perhaps for different sets of managed resources or different types of notifications. The Channel Finder Service does not enable a managing system to create new event channels on the managed system, or change which events any of the channels handles. Individual event channels, however, do allow managing systems to add filters and destinations for the notifications handled by the channel. Thus, the managed system registers information about the configuration of the event channels with the Channel Finder Service, which managing systems can subsequently retrieve and use to decide to which channels to listen. The Channel Finder Service is found by looking it up in the Naming Service. The managed system is required to place a reference to it in the root naming context.

When the Channel Finder reports information about which objects it handles events for, it identifies the objects by name. Since managed objects on coarse-grained interfaces will have individual names based on containment just as managed objects on fine-grained interfaces, the Channel Finder can be used in the same way on coarse-grained interfaces. Therefore, no new requirements are placed on the use of the Channel Finder Service to support coarse-grained interfaces.

9.3 The terminator service

ITU-T Q.816 defines the Terminator Service as a common service to implement the functions needed to delete a managed object. Each managed object supported by the framework has a delete policy attribute, set when the object is created. The Terminator Service ensures that this delete policy is followed when the object is deleted. Also, managed objects are named based on containment relationships. It is therefore important that objects that contain other objects only be deleted if the contained objects are also deleted. The Terminator Service does this, and ensures the validity of the containment-based naming tree.

Managed objects on a coarse-grained interface will be treated the same way. So, the Terminator Service will still be needed to play its role on coarse-grained interfaces. There are, however, some differences on coarse-grained interfaces that will affect how the Terminator Service fulfils its role. First, if the Terminator Service uses the managed objects' CORBA interface to retrieve their delete policy (as opposed to some proprietary means), it will have to do so in the manner prescribed by ITU-T X.780.1 for interacting with managed objects through a facade. Second, since the managed objects' names may not be stored in the OMG Naming Service, the Terminator Service will have to retrieve containment information from the Containment Service (see 9.6), and maintain the correctness of the naming hierarchy there. If managed object names are stored in the Naming Service as well as the Containment Service, the Terminator Service will have to update the naming trees in both unless the services synchronize themselves.

These changes, though, will not require modifications to the IDL interface of the Terminator Service. ITU-T Q.816 defines this interface with two delete operations, one which takes a managed object name to identify the object, and one which takes a managed object reference (IOR). The operation that takes a managed object reference will not apply on coarse-grained interfaces since the managed objects may not have individually unique IORs. The operation is defined so that the reference must be of type *ManagedObject* or a subclass. Facade interfaces do not inherit from this interface, so it will be impossible for a managing system to mistakenly try to use this interface to delete a managed object that did not have an IOR.

A managing system will delete managed objects on coarse-grained interfaces by passing the name of the object to the Terminator Service's *deleteByName* operation, just as it could for a managed object on a fine-grained interface. The Terminator Service shall recognize that the name refers to a facade-accessible object based on information in the name itself. (See 8.1.3.) The Terminator Service shall then determine the facade interface that can be used to access this managed object, and retrieve the object's delete policy, passing along the object's name. If the Terminator Service ends up deleting the object, it will do so by invoking the *destroy* operation on the same facade, again passing along the object's name. This is very similar to managed objects on fine-grained interfaces, except that with these objects the Terminator Service uses the object's interface directly, rather than a facade interface.

(R) TERM-1. The Terminator Service shall recognize the differences in names for objects accessible through a facade and objects that are not, and access them accordingly.

(R) TERM-2. When deleting a facade-accessible managed object and accessing it using CORBA, the Terminator Service shall use the object's facade interface to retrieve its delete policy and destroy it (unless it uses an implementation-specific non-CORBA access method).

(R) TERM-3. When deleting a facade-accessible object the Terminator Service shall access containment information stored in the Containment Service. The Terminator Service must also ensure the names in the Containment Service are correct as objects are deleted.

9.4 The multiple-object operation service

ITU-T Q.816 defines the Multiple Object Operation (MOO) Service as a common service that can be used by a managing system to invoke operations on groups of managed objects with a single (or small number of) method invocations between the managing system and managed system. To use the service, the managing system invokes a single operation on the MOO service resident on the managed system. The operations supported are *get*, to retrieve one or more values from a group of managed objects, *set*, to modify one or more values on a group of managed objects, and *delete*, to delete a group of managed objects. The group of objects to be operated on is identified with a scope and a filter. A scope identifies a base object plus some set of objects contained by that object, based on their names. A filter is a logical statement testing the values of the attributes of the objects in the scope of the operation. If the statement evaluates to true for a particular object, the operation is applied to that object. If the operation is applied to many objects, the results may be too large to return in one batch. The MOO service uses the iterator design pattern to enable the managing system to retrieve the results in manageable batch sizes.

ITU-T X.780.1 also defines a common facade operation that will enable a managing system to retrieve attribute values from multiple objects accessed through that facade by supplying a list of the objects' names on a "bulk get" operation. This "bulk get" operation, however, does not supply the flexibility and power of the MOO Service operations. It is therefore desirable that the MOO service also be applicable to coarse-grained interfaces.

There are, however, some differences on coarse-grained interfaces that will affect how the MOO Service fulfils its role. First, since managed objects' names might not be stored in the OMG Naming Service, the MOO Service will have to retrieve containment information from the Containment Service (see 9.6). Second, if the MOO Service uses the managed objects' CORBA interface (as opposed to some proprietary means) to retrieve attribute values for filtering and to invoke the actual operation, it will have to use the managed object's facade. Third, while coarse-grained object interfaces do not have methods to return an object's name, the MOO service must always return each object's name as part of the results of a scoped get operation, just as it does for fine-grained interfaces. It must also allow managing systems to list the objects' name among the list of attributes it wants to retrieve.

These changes, though, will not require modifications to the IDL interface of the MOO Service or its iterators. ITU-T Q.816 defines the MOO Service interface with three operations, get, set, and delete. Each of these takes the name of the base object, a scope indicator, and a filter statement. If attributes are to be acted upon, they are identified by name. All of these are the same on coarse-grained interfaces as they are on fine-grained interfaces. What changes with coarse-grained interfaces is where containment information is located, and how managed object attributes are accessed.

Containment information is located in the Containment Service instead of the Naming Service, and to access a managed object's attributes the MOO service will have to determine the managed object's facade interface and invoke the operation there, supplying the managed object's name. Using the Containment Service to retrieve containment information may actually be simpler on coarse-grained interfaces since the Containment Service accepts a base object name and scope. The MOO service can simply pass to the Containment Service the base object name and scope supplied to it, then iterate through the contained objects to see if they pass the filter. It does not have to navigate the naming tree.

Accessing managed object attributes through a facade is not too different from accessing attributes directly. Given a name, the correct facade is determined by matching the kind of object (supplied in the *kind* field of the last name component) with a facade registered in the root naming context. The operation is then applied to the facade, with the name of the managed object supplied as the first parameter on the operation.

(R) MOO-1. The MOO Service shall recognize the differences in names for managed objects accessible through a facade and those that are not, and treat them accordingly.

(R) MOO-2. When accessing a facade-accessible managed object using CORBA, the MOO Service shall use the managed object's facade interface (unless it uses an implementation-specific non-CORBA access method).

(R) MOO-3. The MOO Service shall determine the objects within the scope of an operation by accessing containment information stored in the Containment Service.

9.5 Heartbeat service

Because CORBA provides location-transparency, network management applications lose visibility of the connections to specific systems. This makes the applications easier to develop, but could prevent a managing system from knowing when communications with a managed system is lost. To work around this, ITU-T Q.816 defines the Heartbeat Service. This service enables a managing system to configure a managed system to periodically emit a short notification. The notification goes through each of the event channels on a managed system, so in addition to testing the communications link it also tests the operation of the notification event channels. A managing system can be alerted of problems if it does not receive a heartbeat notification from a managed system when it is supposed to.

Coarse-grained interfaces will use event channels in the same capacity as fine-grained interfaces. The Heartbeat Service can continue to be used to verify the operation of these channels and the data communications network connecting the managed system with its managing systems. No additional requirements are placed on the use of the Heartbeat Service to support coarse-grained interfaces.

9.6 Containment service

Containment relationships between managed objects on fine-grained interfaces are represented by the names stored in the OMG Naming Service. A similar capability is required for managed objects on coarse-grained interfaces. That is, a function is needed to be able to report which objects are contained by a superior object, to verify that a superior object exists before a subordinate is created, to make sure two objects with the same name are not created, etc. The framework will be extended to support this function by adding a new service, the Containment Service.

(R) CONTAINMENT-1. A managed system shall instantiate at least one Containment Service object. Also, each local root naming context on a system shall have at least one name binding for a Containment Service object. The value of the *ID* string in this binding shall simply identify the server, perhaps with a value similar to "Containment1". The *kind* string in the binding shall identify the class of the object ("itut_q816_1::ContainmentService").

9.6.1 Containment service rationale

There are many different places where managed object names could be stored: the Naming Service, the facade objects, an existing service such as the Terminator Service or MOO service, or a new service.

Putting the names in the Naming Service does not really make sense because the true purpose of the Naming Service is to bind names to IORs, and managed objects on coarse-grained interfaces may not have their own IORs. Binding the name to the facade's IOR would be inefficient. All of the managed objects accessed through one facade would share the same IOR, so all of their names would be bound to the same IOR in the naming service. The result would likely be large numbers of copies of just a few IORs stored in the naming service.

Putting the names in the facade objects, or the managed objects themselves, is a logical choice, but one purpose of the framework is to provide places to implement common functions. Duplicating the management of containment information in every facade or managed object runs counter to this.

The Terminator Service and MOO Service both rely upon containment information to perform their tasks, so one possibility is to extend these services to manage containment information. These services, however, do not have the management of containment data as their primary duty. Placing the management of containment information elsewhere enables these services to continue to focus on their intended tasks.

While adding a new service is a significant change to the framework, it does seem to be the best choice. It provides a single repository for containment information, and provides an opportunity to introduce features for accessing containment information that are not supported by the Naming Service, such as using scope to query for contained objects.

9.6.2 Containment service description

The main function to be supported by the containment service is to enable a managing system to query a managed system with the name of an object, and receive back the names of the objects contained by that object. In addition, a means of getting names added to and removed from the service will be defined. These are not for use by managing systems, but internally by managed objects, factories, and other parts of a managed system. They are provided to promote the development of reusable components, possibly by third parties, and are defined on an interface separate from that used by managing systems. Finally, a potentially large number of names may be returned in response to a query, so the iterator design pattern is used. The iterator is described in 9.6.2.3.

9.6.2.1 The containment service interface

The Containment Service provides three operations to retrieve containment information. The IDL describing the Containment Service interface (without comments) is provided below.

```
interface Containment {

     boolean exists (in NameType name)
               raises (ApplicationError);

     NameSetType getContained (in NameType base,
               in ScopeType scope,
               in unsigned short howMany,
               out NameIterator iterator)
               raises (ApplicationError);

     NameSetType getContainedByKind (in NameType base,
               in ScopeType scope,
               in KindType kind,
               in unsigned short howMany,
               out NameIterator iterator)
               raises (ApplicationError);

};   // end of Containment interface
```

The *exists* operation takes a name and returns true if it is registered with the Containment Service. The other two operations return the names of objects contained by the object named in the *base* parameter. The *scope* parameter on both of these operations can be used to specify which part of the tree of objects contained below the base object is to be retrieved. The third operation, *getContainedByKind*, takes a *kind* parameter to instruct the Containment Service to return the names for objects of a certain kind.

One important note about the names exchanged across this interface is that the names supplied by the managing system in these queries need not contain the final component of the name, where the *ID* string is either null or "Object" and the *kind* string is either empty or set to the facade identifier. Names returned to the managing system, however, shall always contain this final name component. Thus, if a managing system has the name of an object but it does not contain the final component, it can get it by querying the Containment Service to find the contained objects but set the scope equal

to "base object only." A managing system might find itself with an object's name missing the final component by, for example, truncating an object's name to find its parent.

As will be seen below, before names can be added to the Containment Service, the names of the local root naming contexts must first be registered. The root naming contexts registered with the Containment Service may be retrieved by submitting a zero-length *base* name to the *getContained* operation along with a *scope* indicating the first level of contained objects. Submitting the name of a root naming context simply results in retrieving the names of the managed objects bound below it. The same is true for any managed object.

(R) CONTAINMENT-2. The interface supported by the Containment Service object(s) shall be the Containment interface described above and defined in the CORBA IDL in Annex A.

(R) CONTAINMENT-3. In response to an invocation of the *exists* operation, the Containment Service shall return true if the name is currently registered with the service, and false otherwise. If some error on the server prevents this determination, an appropriate application error exception shall be raised.

(R) CONTAINMENT-4. In response to an invocation of the *getContained* operation, the Containment Service shall return a list of the names of the objects contained by the object named in the *base* parameter. The list of contained objects shall be determined according to the *scope* parameter. (See ITU-T Q.816 for a description of scope information.) The maximum number of names to be returned in the result set shall be the value of the *howMany* parameter. If more than *howMany* names need to be returned, the Containment Service shall return a reference to a name iterator, and make the additional name available through this interface. If all the names can be returned in the result set, the name iterator reference shall be null. If a zero-length base name is submitted, the first level of contained names shall be the names of the registered root naming contexts. If some error prevents the list from being returned, an appropriate application error exception shall be raised. In particular, if the base name is not registered, an *InvalidParameter* application error exception shall be raised.

(R) CONTAINMENT-5. The Containment Service shall respond to the invocation of the *getContainedByKind* operation as described in requirement CONTAINMENT-4, except only those names that match the *kind* parameter are returned. Recall that the final component of a name has an *ID* of "Object". It is the second-to-last component that the *ID* contains the relative distinguished name for the object and the *kind* field contains the object's kind value. Thus, in response to an invocation of the *getContainedByKind* operation, the Containment Service shall return only those names where the *kind* field in the second-to-last name component matches the value of the *kind* parameter.

9.6.2.2 The containment service component interface

A few more operations are defined on a separate interface for use internally to the managed system. These are defined to promote the development of reusable Containment Service components, possibly by third parties. The IDL describing the Containment Service component interface (without comments) is provided below.

```
interface ContainmentComponent : Containment {

     void registerLocalRoot (in NameType name,
                             in NamingContext localRoot)
                    raises (ApplicationError);

     void unRegisterLocalRoot (in NameType name)
                    raises (ApplicationError, DeleteError);

     void addName (in NameType name)
                    raises (ApplicationError, CreateError);
```

```
        void removeName (in NameType name)
                        raises (ApplicationError, DeleteError);
    };
```

The first two operations are used to register and un-register local root naming contexts with the containment service. When a name is added to the Containment Service, the name of its superior object must already be registered, or the attempt to add the name will fail. At the top of the naming tree, however, will be objects with no superior object on the managed system. These objects will be named relative to a local root naming context. Without knowing what the names of the local root naming contexts are, the Containment Service cannot know if a new name, with an unrecognized superior object, is incorrect or an object at the top of a naming tree.

The final two operations are used to add and remove names to and from the Containment Service. The most likely clients of these operations will be, respectively, managed object factories and the Terminator Service. When a name is added, it is checked to make sure it is unique and that the superior object's name has been added (unless it is named directly relative to a root naming context). A name is only removed if there are no subordinate names still registered.

(O) CONTAINMENT-6. The interface supported by the Containment Service object(s) may be the Containment component interface described above and defined in the CORBA IDL in Annex A.

(O) CONTAINMENT-7. When a local root naming context is registered using the *registerLocalRoot* operation, the Containment Service shall add it to its list of registered local root naming contexts. If an error on the server prevents this, an appropriate application error exception shall be raised.

(O) CONTAINMENT-8. When a local root naming context is un-registered using the *unRegisterLocalRoot* operation, the Containment Service shall remove it from its list of registered local root naming contexts, but only if there are no subordinate object names registered. If an error on the server prevents this, an appropriate application error exception shall be raised.

(O) CONTAINMENT-9. When a managed object name is added using the *addName* operation, the Containment Service shall add the name to its list of registered object names, but only if the name is unique and the name is directly subordinate to either a registered local root or another registered name. If the name is not unique or if no registered superior name exists, the Containment Service shall not add the name and instead raise the appropriate create error exception. Note that if two names are the same except for differences in the final name component (where the *ID* string is "Object" and the *kind* string identifies a facade) these names are not unique. If an error on the server prevents the operation from completing, an appropriate application error exception shall be raised.

(O) CONTAINMENT-10. When a managed object name is removed using the *removeName* operation, the Containment Service shall remove the name from its list of registered object names, but only if there are no subordinate names registered. If one or more subordinate names exist, the Containment Service shall not remove the name and instead raise the appropriate delete error exception. If an error on the server prevents the operation from completing, an appropriate application error exception shall be raised.

9.6.2.3 The name iterator

When the Containment Service returns a list of contained object names, there may be too many to return at once. To enable the Containment Service to return a virtually unlimited number of names, the iterator design pattern is used. As described above, if the Containment Service cannot return a list of names in response to an operation, it returns the most it can along with a non-null reference to an iterator interface. The IDL description of the name iterator interface is shown below.

```
interface NameIterator {

     boolean getNext(in unsigned short howMany,
                     out NameSetType results)
          raises (ApplicationError);
```

```
        void destroy();

    }; // end of interface NameIterator
```

(R) CONTAINMENT-11. The Containment Service shall support the use of name iterators with interfaces matching the description above and the definition in the IDL in Annex A.

(R) CONTAINMENT-12. Each time a client invokes a *getNext* operation on a name iterator, the iterator shall return the next set of results. The iterator shall keep track of how many results have already been retrieved by the client, and return all of the results once. The results initially returned in response to an operation on the Containment Service interface shall not be returned again by the iterator. The iterator shall return in response to a *getNext* operation at most the number of names indicated by the value of the *howMany* parameter. 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 are available. If there are more results to return (in addition to those being returned), the return value of the *getNext* operation shall be true, otherwise false. The iterator shall not return an empty result set unless *howMany* was set to zero or there are no more results to return, as doing so would force the client to poll the iterator.

(R) CONTAINMENT-13. The managed system shall control the life cycle of the iterator. A destroy operation, however, is provided if the manager wants to stop retrieving results before reaching the last iteration. Upon invocation of the *destroy* operation, the iterator shall free any resources it is using and delete itself. Upon returning the last result, the iterator shall destroy itself. The iterator may also be destroyed by the managed system if it is unused for an unreasonably long period of time.

10 Compliance and conformance

This clause defines the criteria that must be met by other standards documents claiming compliance to this framework and the functions that must be implemented by systems claiming conformance to this Recommendation.

10.1 System conformance

10.1.1 Conformance points

This clause summarizes the individual functions described earlier in this Recommendation. These conformance points are then combined in profiles that must be supported by systems claiming conformance to this Recommendation.

1) Coarse-grained Naming: An implementation claiming conformance to the Coarse-grained Naming requirements must:
   - Support all of the Naming Service requirements specified in 8.1.
2) Coarse-grained Terminator Service: An implementation claiming conformance to the Coarse-grained Terminator Service requirements must:
   - Support all of the Terminator Service requirements specified in 9.3.
3) Coarse-grained MOO Service: An implementation claiming conformance to the Coarse-grained MOO Service requirements must:
   - Support all of the MOO Service requirements specified in 9.4.
4) Containment Service: An implementation claiming conformance to the Containment Service requirements must:
   - Support the mandatory Containment Service requirements specified in 9.6. The mandatory requirements are preceded by "**(R)**." Those that are preceded by "**(O)**" may be supported but are not required.

10.1.2 Basic conformance profile

A system claiming conformance to the Q.816.1 Basic Profile shall support:

1) The Q.816 Basic Profile **except**:
   - managed object names are not required to be bound to managed object IORs in the OMG Naming Service.
2) The Coarse-grained Naming requirements. (See conformance point 1 above.)
3) The Coarse-grained Terminator Service requirements. (See conformance point 2 above.)
4) The Coarse-grained MOO Service requirements. (See conformance point 3 above.)
5) The Containment Service requirements. (See conformance point 4 above.)

10.2 Conformance statement guidelines

The users of this framework 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 a module in a conformance statement, therefore, will not suffice to identify a set of IDL interfaces. The conformance statement must identify a document and year of publication to make sure the right version of IDL is identified

ANNEX A

Coarse-grained framework support services IDL

```
/* This IDL code is intended to be stored in a file named "itut_q816_1.idl"
located in the search path used by IDL compilers on your system. */

#ifndef ITUT_Q816_1_IDL
#define ITUT_Q816_1_IDL

#include <CosNaming.idl>
#include <itut_q816.idl>
#include <itut_x780.idl>

#pragma prefix "itu.int"


module itut_q816 {

// IMPORTED TYPES

     // Types imported from CosNaming
     typedef CosNaming::NamingContext NamingContext;
     typedef CosNaming::Istring KindType;

// INTERFACES

     /** The Name Iterator interface is used to retrieve the results from
     a Containment interface getContained or getContainedByKind operation
     using the iterator design pattern. */

     interface NameIterator {

          /** This method is used to retrieve the next "howMany" results
          in the result set.
          @param howMany  The maximum number of items to be returned in
                          the results. Fewer may be returned if that is
                          all that is left, or to balance delay
                          with efficiency.
          @param results  The next batch of results.
```

```
      @return         True if there are more results after those
                      being returned. If the return value is true
                      the results set should not be empty, as this
                      forces the client to poll for results.
                      Instead the call should block.
      */

      boolean getNext(in unsigned short howMany,
            out NameSetType results)
            raises (itut_x780::ApplicationError);

      /** This method is used to destroy the iterator and release its
      resources. The iterator, though, is automatically destroyed
      after the last results are returned, and may be destroyed if
      unused for an unreasonably long period. */

      void destroy();

};  // end of interface NameIterator


/** The Containment Service interface is used to retrieve from a
managed system information about the containment relationships between
the managed objects on the system. Containment relationships are
represented through names. An object that is contained by another
object is named relative to it. All managed object names are
registered with the Containment Service. */

interface Containment {

      /** This method is used to check to see if a name is registered
      with the Containment Service.
      @param name          The name to check to see if it is registered
      @return              True if the name is registered. All components
                           of the submitted name match a registered name.
      */

      boolean exists (in NameType name)
            raises (itut_x780::ApplicationError);

      /** This method is used to retrieve the names of the objects
      contained under a target object. An object is contained by the
      target object if its name begins with all but the last
      component of the target object's name. The iterator design
      pattern is used to support returning potentially large numbers
      of names. If the name does not exist, an application error
      exception is raised indicating an invalidParameter (See X.780
      for application error exception codes.)
      @param name          The name of the object for which the contained
                           objects are sought. Need not contain the final
                           name component (where ID=Object, kind = facade)
      @param scope         The scope is used to identify what part of
                           the tree of contained objects to return.
      @param howMany       The maximum number of names to return in the
                           results. If there are more than howMany names,
                           an iterator must be used to return the rest.
      @param iterator      A reference to an iterator to return additional
                           results. If all results are returned in
                           response to the call, this shall be null.
      @return              The names of the contained objects. Must
                           always contain the final name component.
      */

      NameSetType getContained (in NameType name,
            in ScopeType scope,
            in unsigned short howMany,
            out NameIterator iterator)
            raises (itut_x780::ApplicationError);
```

```
      /** This method is used to retrieve the names of the objects
      contained under a target object that match a specific kind.
      An object's kind is the value of the kind field in the
      next-to-last component of its name. This field must match
      the submitted kind value.
      @see getContained
      @param name          The name of the object for which the contained
                           objects are sought. Need not contain the final
                           name component (where ID=Object, kind = facade)
      @param scope         The scope is used to identify what part of
                           the tree of contained objects to return.
      @param kind          The value that must be matched in the kind
                           field of the next-to-last component in the
                           names returned.
      @param howMany       The maximum number of names to return in the
                           results. If there are more than howMany names,
                           an iterator must be used to return the rest.
      @param iterator      A reference to an iterator to return additional
                           results. If all results are returned in
                           response to the call, this shall be null.
      @return              The names of the contained objects. Must
                           always contain the final name component.
      */

      NameSetType getContainedByKind (in NameType name,
            in ScopeType scope,
            in KindType kind,
            in unsigned short howMany,
            out NameIterator iterator)
            raises (itut_x780::ApplicationError);

};    // end of interface Containment


/** The Containment Component interface extends the Containment
interface to add functions used internally to a managed system.
*/

interface ContainmentComponent : Containment {

      /** This method is used to register a local root naming context
      with the terminator service. Re-registering a name results in
      the newly supplied reference subsequently being used by the
      service, if it needs to access the root naming context. */

      void registerLocalRoot     (in NameType name,
                                  in NamingContext localRoot)
                                  raises (itut_x780::ApplicationError);

      /** This method is used to remove a local root naming context
      registration. */

      void unregisterLocalRoot (in NameType name)
            raises (itut_x780::ApplicationError);

      /** This method is used to add a name to the Containment
      Service. The name must be relative to an existing name or
      local root name. This means all but the last two name
      components of the submitted name must match all but the last
      name component of a registered name. Also, the name must
      be unique. This means that all but the last name component
      of the submitted name cannot match all but the last name
      component of any other registered name. If the name has
      no relative superior object registered, the name is not
      registered and a create error exception is raised with the
      cause set to badName. If the name is not unique, the name
      is not registered and a create error exception is raised
      with the cause set to duplicateName. (See X.780 for
      create error exeption codes.)
      @param name     The name to be added.
      */
```

```
            void addName (in NameType name)
                 raises (itut_x780::ApplicationError,
                      itut_x780::CreateError);

            /** This method is used to remove a name from the Containment
            Service. There must be no contained names registered with
            the Containment service when a name is removed. This means
            there must be no other names beginning with all but the last
            name component of the name to be removed. If there are, the
            name is not removed and a DeletError exception is raised with
            the cause set to containsObjects (see X.780). If the name
            does not exist, an application error is raised with the
            cause set to invalidParameter (See X.780).
            @param name     The name to be removed.
            */

            void removeName (in NameType name)
                 raises     (itut_x780::ApplicationError,
                             itut_x780::DeleteError);


      };    // end of interface ContainmentComponent

}; // end of module itut_q816

#endif // end of #ifndef ITUT_Q816_1_IDL
```

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
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	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
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 and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	Languages and general software aspects for telecommunication systems

Printed in Switzerland
Geneva, 2002