CORBA/TC Interworking and SCCP Inter-ORB Protocol Specification

First Edition
January 2001

PATENT

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

NOTICE

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

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIEDWARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR PARTICULAR PURPOSE OR USE. In no event shall The Object Management Group or any of the companies listed above be liable for errors contained herein or for indirect, incidental, special, consequential, reliance or cover damages, including loss of profits, revenue, data or use, incurred by any user or any third party. The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

ISSUE REPORTING

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

Contents

Contents

Preface

About the Object Management Group

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

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

What is CORBA?

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

OMG Documents

In addition to the CORBA Core Specification, OMG's document set includes the following publications.

OMG Modeling

The Unified Modeling Language (UML) Specification defines a graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems. The specification includes the formal definition of a common Object Analysis and Design (OA&D) metamodel, a graphic notation, and a CORBA IDL facility that supports model interchange between OA&D tools and metadata repositories. The UML provides the foundation for specifying and sharing CORBA-based distributed object models.

The Meta-Object Facility (MOF) Specification defines a set of CORBA IDL interfaces that can be used to define and manipulate a set of interoperable metamodels and their corresponding models. The MOF provides the infrastructure for implementing CORBA-based design and reuse repositories. The MOF specifies precise mapping rules that enable the CORBA interfaces for metamodels to be automatically generated, thus encouraging consistency in manipulating metadata in all phases of the distributed application development cycle.

The OMG XML Metadata Interchange (XMI) Specification supports the interchange of any kind of metadata that can be expressed using the MOF specification, including both model and metamodel information. The specification supports the encoding of metadata consisting of both complete models and model fragments, as well as tool-specific extension metadata. XMI has optional support for interchange of metadata in differential form, and for metadata interchange with tools that have incomplete understanding of the metadata.

Object Management Architecture Guide

This document defines the OMG's technical objectives and terminology and describes the conceptual models upon which OMG standards are based. It defines the umbrella architecture for the OMG standards. It also provides information about the policies and procedures of OMG, such as how standards are proposed, evaluated, and accepted.

OMG Interface Definition Language (IDL) Mapping Specifications

These documents provide a standardized way to define the interfaces to CORBA objects. The IDL definition is the contract between the implementor of an object and the client. IDL is a strongly typed declarative language that is programming language-independent. Language mappings enable objects to be implemented and sent requests in the developer's programming language of choice in a style that is natural to that language. The OMG has an expanding set of language mappings, including Ada, C, C++, COBOL, IDL to Java, Java to IDL, Lisp, and Smalltalk.

CORBAservices

Object Services are general purpose services that are either fundamental for developing useful CORBA-based applications composed of distributed objects, or that provide a universal-application domain-independent basis for application interoperability.

These services are the basic building blocks for distributed object applications. Compliant objects can be combined in many different ways and put to many different uses in applications. They can be used to construct higher level facilities and object frameworks that can interoperate across multiple platform environments.

Adopted OMG Object Services are collectively called CORBAservices and include Collection, Concurrency, Event, Externalization, Interoperable Naming, Licensing, Life Cycle, Notification, Persistent Object, Property, Query, Relationship, Security, Time, Trader, and Transaction.

CORBAfacilities

Common Facilities are interfaces for horizontal end-user-oriented facilities applicable to most domains. Adopted OMG Common Facilities are collectively called CORBAfacilities and include Internationalization and Time, and Mobile Agent Facility.

Object Frameworks and Domain Interfaces

Unlike the interfaces to individual parts of the OMA “plumbing” infrastructure, Object Frameworks are complete higher level components that provide functionality of direct interest to end-users in particular application or technology domains.

Domain Task Forces concentrate on Object Framework specifications that include Domain Interfaces for application domains such as Finance, Healthcare, Manufacturing, Telecoms, E-Commerce, and Transportation.

Definition of CORBA Compliance

The minimum required for a CORBA-compliant system is adherence to the specifications in CORBA Core and one mapping. Each additional language mapping is a separate, optional compliance point. Optional means users aren’t required to implement these points if they are unnecessary at their site, but if implemented, they must adhere to the *CORBA* specifications to be called CORBA-compliant. For instance, if a vendor supports C++, their ORB must comply with the OMG IDL to C++ binding.

Interoperability and Interworking are separate compliance points. For detailed information about Interworking compliance, refer to the *Common Object Request Broker: Architecture and Specification*, *Interworking Architecture* chapter.

Obtaining OMG Documents

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

OMG formal (published) specifications are available from the OMG website *http://www.omg.org/technology/documents/formal/index.htm*. To obtain print-on-demand books in the documentation set or other OMG publications, contact the Object Management Group, Inc. at:

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

Acknowledgments

The following companies submitted parts of this specification:

- AT&T
- GMD FOKUS
- IONA Technologies, Plc
- NORTEL
- Teltec DCU

Introduction 1

Contents

This chapter contains the following sections.

Source Documents

This specification is based on the following OMG document(s):

- telecom/98-10-03 - submission document
- dtc/99-12-02 - FTF draft adopted specification
- dtc/00-02-02 - FTF final adopted specification
- dtc/00-02-08 - FTF final report

1.1 Interworking Overview

This specification addresses the interworking between CORBA-based Intelligent Network (IN) applications and the same applications implemented using the existing IN infrastructure. Two scenarios for the use of CORBA in IN signaling are:

1. The interworking of CORBA-based IN Application Entities (e.g., a Service Control Point (SCP)) with legacy IN Application Entities (e.g., a Service Switching Point (SSP)) through a gateway mechanism that provides a CORBA view of a legacy target and a legacy view of a CORBA target.

2. The interworking of CORBA-based IN Application Entities while using the existing Signalling System No. 7 (SS7) infrastructure as a transport network for GIOP messages.

The first type of interworking may be identified as a gateway that allows CORBA-based IN applications to interwork with those that communicate using the Intelligent Network Application Part (INAP) or another protocol that uses TC/ROS services.

However, while it is anticipated that the principal and immediate application of the solution provided in this specification will be the development of an IN/CORBA gateway, and most of the examples that illustrate the use of such gateways will be based on IN scenarios, it must be kept in mind that the solution described here is of general applicability to any TC User. An important family of TC-based protocols that need to be supported by the gateway includes the Mobile Application Part (MAP) protocol. The various MAP specifications [12] (e.g., MAP-GSM, IS41, etc.) define signalling traffic for mobile telephony networks. The approach taken in this specification supports any TC-User protocol and is not limited to INAP or any set of INAP variants.

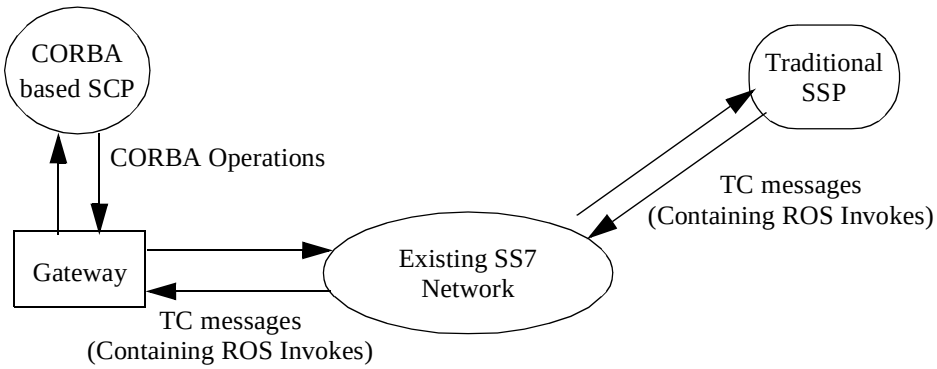


Figure 1-1 Interworking between CORBA-based IN applications and Traditional IN Applications (IN/CORBA Gateway)

In Figure 1-1, the CORBA-based SCP (or any TC-User Application Entity) has IDL interfaces created through Specification Translation of the ASN.1 specifications of INAP (or any TC-User protocol). This IDL-based specification provides a uniform interface that may be used when implementing either native CORBA-based IN Applications or Proxy CORBA objects at the gateway. This uniformity is essential to ensure location transparency and to eliminate the need for proxy objects in native CORBA to CORBA interactions. The translation algorithm is based on the NMF/The Open Group Joint Inter-Domain Management Task Force (JIDM) work on ASN.1 to IDL specification translation [4].

To support IN/TC-User interaction semantics (naming, dialogs, etc.) in the CORBA domain, a small set of functionality has been defined that re-uses some of the CORBA Object Services (Interaction Translation). This allows maximum re-use of the CORBA infrastructure when using it as an environment for developing IN or other TC-User

applications. It also means that building an IN/TC-User application is simplified as most of the TC-specific functionality has been encapsulated in these TC-specific uses of some of the CORBA Object Services.

Additionally an IDL-based TC interface is provided to standardize the communication between proxy objects associated with the gateway and the TC protocol stack.

The second type of interworking may be identified as what the EURESCOM P508 project [13] calls a Kernel Transport Network. This allows islands of CORBA objects to communicate over the existing SS7 infrastructure, thereby taking advantage of the investment of public network operators in this efficient and robust packet network.

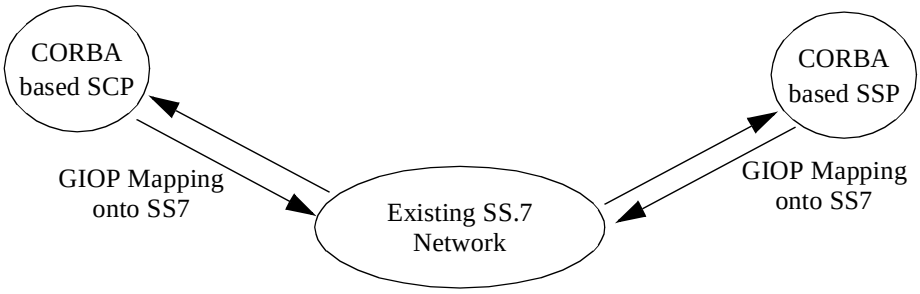


Figure 1-2 Interworking between islands of CORBA based IN Applications using the SS7 Network (SS7 as Kernel Transport Network)

In Figure 1-2, the ORB hides the use of SS7 as a transport mechanism from the interacting CORBA objects. This specification defines a GIOP mapping onto the connectionless Signalling Connection Control Part (SCCP) protocol of the SS7 protocol suite.

1.2 Architectural Overview

1.2.1 Introduction

There are two complementary but orthogonal architectures in this specification:

1. *TC/CORBA Application Interworking*: a set of static Specification Translation algorithms for converting TC/ROS-User Application definitions in ASN.1 to OMG IDL and a set of TC/ROS-User services. The IDL interfaces for the TC/ROS-User specific services provide additional functionality on top of the IDL interfaces defined for a subset of the standard CORBA object services. The Interworking mechanisms described in this specification are location transparent (i.e., when interacting through a gateway with entities in another domain no special interaction sematics are required).

2. *GIOP Mapping onto connectionless SCCP*: a well-defined new GIOP mapping, onto connectionless SCCP in the SS7 stack, is defined.

An overview of each of these architectures is provided in the following sub-sections.

1.2.2 TC/CORBA Application Interworking

1.2.2.1 Background on TC/SS7 and IN

Signalling in telecommunications networks consists of communication between customer premise equipment, switches, network-based servers, and databases to complete a call between two end-subscribers (either fixed or mobile). The signaling protocol used by telecom network operators and equipment vendors for communications between the various network elements is the one standardized by the ITU-T, the Signaling System No. 7 (SS7) protocol suite [9].

A brief overview of the SS7 protocol architecture of relevance to this specification is provided below. As shown in Figure 1-3 on page 1-5, the SS7 protocols consist of:

- The stack comprising of the Message Transfer Part (MTP), which provides a connectionless, highly reliable datagram capability augmented by some additional addressing capabilities provided by the connectionless Signalling Connection Control Part (SCCP).
- On top of this is the Transaction Capabilities (TC) protocol, which consists of the Transaction sub-layer (TSL) that provides a very “skinny” end-to-end connection for the transfer of remote operations (RO) using the OSI Remote Operations Service (ROS), which is essentially an RPC-like capability.

The specifics of the remote operations and their returns are described as ROS Application Service Elements (ASE) using the Abstract Syntax Notation One (ASN.1) and encoded using the Basic Encoding Rules (BER). One group of ASEs is of particular interest to this specification - the remote operations that define the interactions between switches and network servers (containing service-specific intelligence) are defined by a set of ASEs called the Intelligent Network Application Part (INAP). While it is anticipated that the principal and immediate application of the solutions provided in this specification will be the development of a gateway that maps INAP-based remote operations to CORBA calls, and all the examples that illustrate the use of such gateways will be based on the Intelligent Network architecture, it must be kept in mind that the solutions offered here are of general applicability to any TC-User ASE.

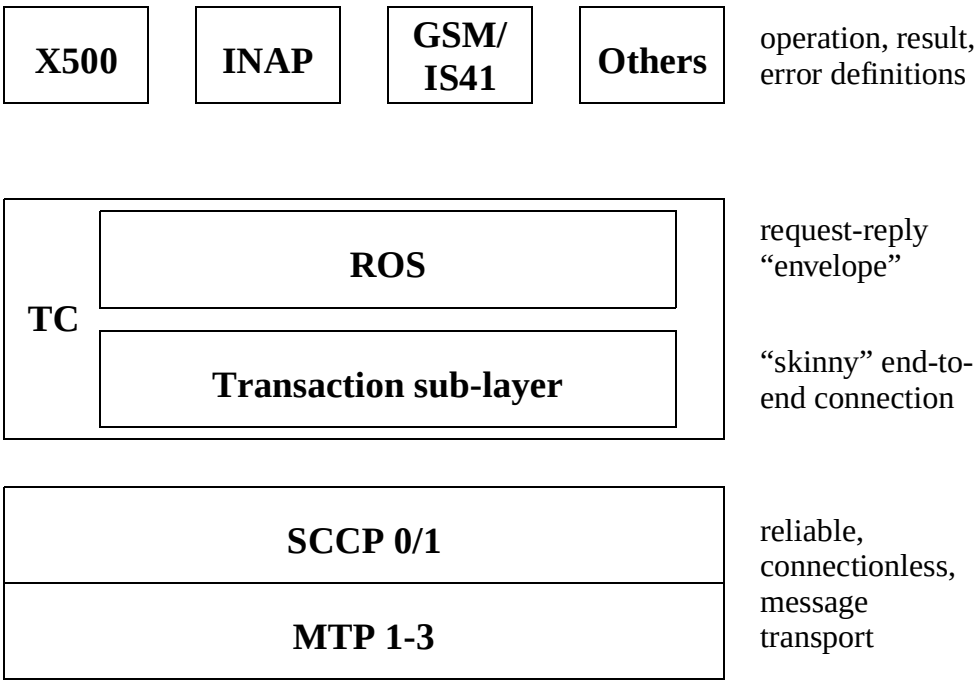


Figure 1-3 The SS7 Protocol Stack

Several scenarios for the use of CORBA in signalling networks are envisaged. The principal application appears to be in support of Intelligent Network (IN) services. Briefly, therefore, an IN-based architecture refers to all extensions of switched networks that include request/reply based communications between switching equipment and remote network servers to enhance the development and real-time delivery of services.

In an IN, the service application programs are no longer totally switch-based, but are distributed among the switches and supporting computers (remote servers). This separation of service control from traditional switching functions, if provided through logical separation of the respective control software, has the advantage of considerably simplifying the software development and maintenance in a switch.

If the network implementation of this logical separation places the service control functions remote from the switch, such as in a remote server, the switch interrupts its processing at certain pre-determined points to query the remote server to provide service-specific instructions. The switches in a network do not have to be loaded with service-specific information. That is located at remote servers, where the service-specific logic and service data can be added and changed by Operations Support Systems (OSS). This permits quicker service implementations, and allows the network operator to more easily customize the service logic and data for individual customers. It is also possible to allow customers with access to their data and service logic, something that would be regarded with concern if these functions were located at a switch. This separation is fundamental to Intelligent Network (IN) architectures.

Figure 1-4 shows a typical IN architecture. (The interfaces of interest for this specification are shown in bold).

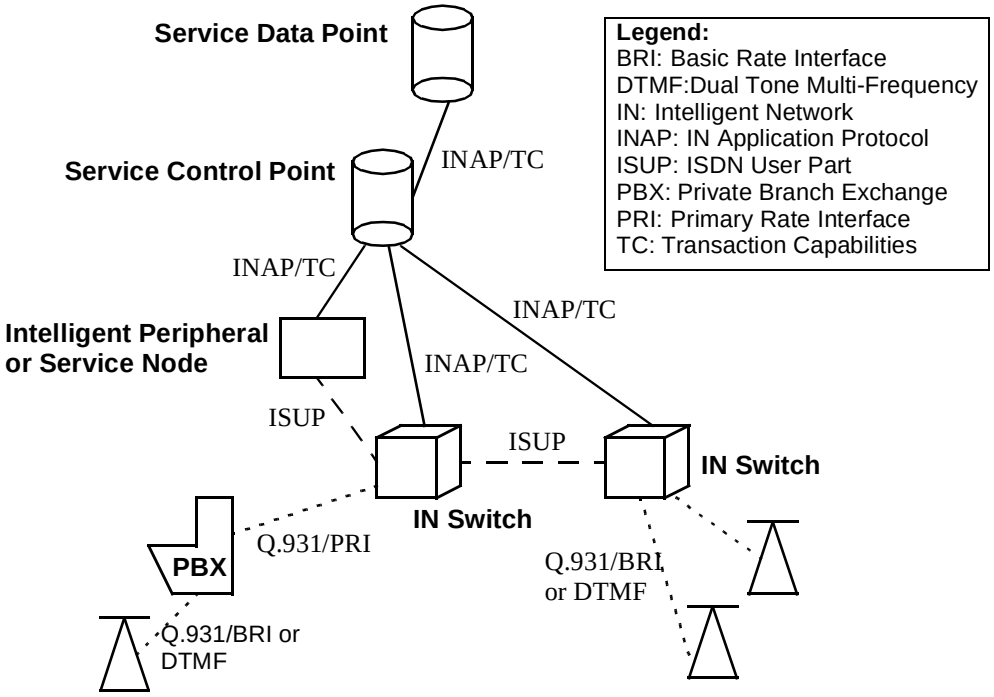


Figure 1-4 Typical IN Architecture and Signalling Interfaces

IN-capable switches called Service Switching Points (SSP) are connected to remote servers called Service Control Points (SCP) via the SS7 network. SCPs may have associated databases, called Service Data Points (SDP). The standardized application protocol on the interfaces of interest (shown in bold in Figure 1-4) is the Intelligent Network Application Part (INAP), which in turn uses the request-reply mechanism of the Transaction Capabilities Part (TC) protocol.

A likely application of CORBA in IN signalling is the interworking of CORBA-based implementations of SCPs with legacy SSPs through a bridging mechanism where the CORBA-based implementations offer an external message-based interface, which is identical to the standardized SS7 protocol suite. (A more long-term, though perfectly technically feasible scenario, is the reverse situation, where CORBA-based SSPs communicate with “legacy” SCPs).

1.2.2.2 Interworking Architectures

The interworking mappings provided in this specification will enable IN-based systems that define their interfaces based on INAP and use the SS7 protocol to interwork with CORBA-based IN systems. This requires a mapping between existing IN-based system

interfaces and CORBA object models. Several algorithms need to be provided to effect this interworking. Application interworking is split into Specification Translation (ST) and Interaction Translation (IT).

The ST is an extension of the JIDM specification translation standard [**4**] already adopted by The Open Group/NMF and also adopted by the *OMG TMN/CORBA Interworking* specification. The JIDM work uses GDMO templates to define IDL interfaces for managed objects. In the TC-User domain there are no GDMO templates defined, instead TC-User protocols are defined by several ASN.1 constructs, either using the ASN.1 macro notation (pre-1994) or ASN.1 information object classes (post-1994). The JIDM approach is expanded here to define mappings for these constructs to IDL interfaces. The JIDM translation algorithms of basic ASN.1 types to IDL are retained. As the JIDM ST does not provide a mapping of the ROS constructs based on the 1994 ASN.1 notation, there will be no discrepancy between the two ST algorithms.

Specifically, the ST will map a generic TC-User protocol (defined using Abstract Syntax Notation One, and using the ROS Information Object Classes OPERATION, ERROR, and EXTENSION to specify remote operations) to the corresponding CORBA IDL constructs. While INAP operations will be used to illustrate the mappings, it is once again worth reiterating that the mappings will permit the conversion of any TC-User protocol to its CORBA equivalent. This is particularly important in the IN domain due to the proliferation of variants of the basic IN protocol (INAP) and the pressures for convergence between INAP and other IN-like protocols such as Mobile Application Part (MAP).

The IT defines how to locate, name, and interact with TC/ROS-User implementations in the CORBA domain. It includes defining the mapping of the TC dialog handling facilities onto appropriate behavior at the gateway such that a TC-User implemented as a CORBA object has access to the TC facilities, while a TC-User implemented in the non-CORBA domain is offered the same standardized message-based interface as at present. This also includes the conversion between CORBA calls marshalled using IIOP/CDR onto TC APDUs encoded in BER. The interworking implementation also has to construct the SCCP/MTP “envelope” for communication with the network element in the SS7 domain.

Figure 1-5 on page 1-8 illustrates the resulting architecture.

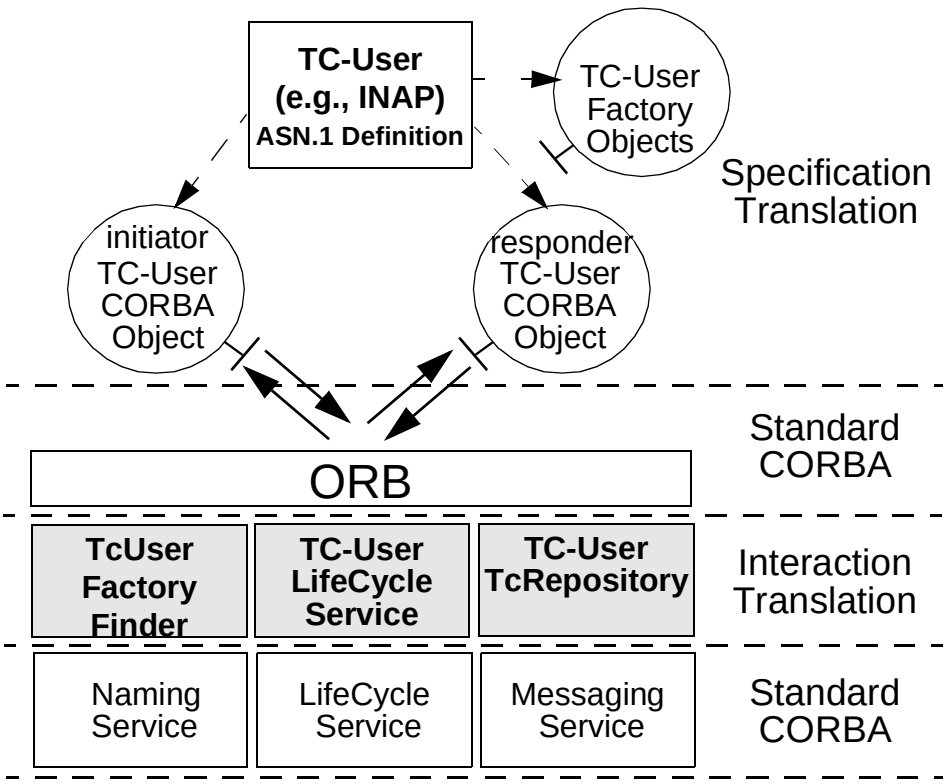


Figure 1-5 TC/CORBA Application Interworking Architecture

It is also possible to define a TC/CORBA gateway based on this architecture. By using proxy objects at a gateway, all TC-User interactions of CORBA objects can be mapped into TC PDUs in the SS7 domain. This is because the semantics of TC-User interactions have been retained in the CORBA domain. All of the interfaces defined during ST allow the transfer of sufficient context information to allow an individual CORBA object (or proxy object at a gateway) to maintain multiple TC dialogs simultaneously. This greatly enhances scalability of the solution. Additionally CORBA objects may support explicit TC flow control primitives (TC-Begin, TC-Continue, TC-End, etc.) to aid further in optimal utilization of the SS.7 infrastructure when interacting with TC-User AEs in the traditional SS7 domain.

TC-User CORBA objects and TC-User proxy objects (at a gateway) may optionally support TC flow control primitives through some IDL-defined parameters.

One further (optional) facility is provided in this specification, namely IDL interfaces to the TC protocol stack (hereafter called the TC PDU-oriented interfaces), which allow proxy objects standardized, efficient access to the TC/SS7 protocol stack. Of course individual proxy object implementations may instead access a particular TC implementation through a proprietary API.

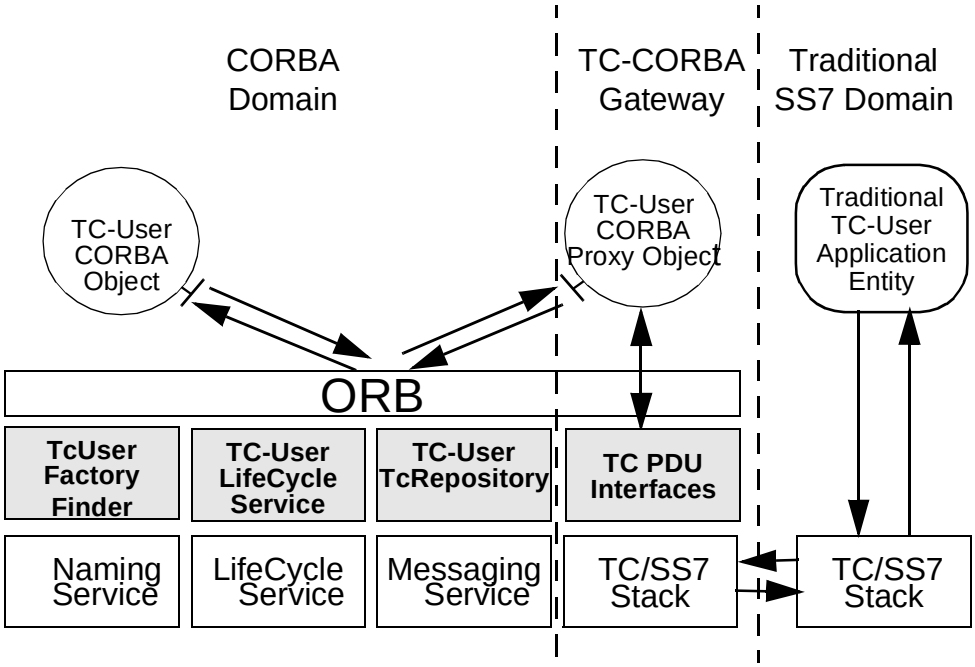


Figure 1-6 TC/CORBA Gateway based on Interworking Architecture

1.2.2.3 Specification Translation

Specification Translation defines a set of algorithms for mapping between ASN.1 descriptions of TC-User Protocols and CORBA-based TC-User objects. TC-User objects are characterized by sets of operations defined using the APPLICATION-SERVICE-ELEMENT (ASE) macro or the OPERATION-PACKAGE information object class and, optionally, the APPLICATION-CONTEXT (AC) macro or CONTRACT information object class to define the specific ASEs and the rules of interactions for a TC dialog.

Each AC/Contract is mapped to two CORBA interfaces, one for the initiator of a dialog and one for the responder. Each interface is populated with IDL operations and exceptions based on the ASN.1 operations and errors defined within the ASN.1 definition of the AC/Contract. A **Factory** interface is defined that has convenient create operations for each interface defined during Specification Translation. For TC-User modules that don't have ACs/Contracts defined, a symmetrical default interface is created into which all the defined operations are placed, together with a **Factory** interface with create operations to create the default single interface.

When translating ASN.1 operations and errors to IDL operations and exceptions, an additional parameter is added to hold TC-specific context information (such as invoke IDs, etc.). The mapping of the basic ASN.1 types uses the JIDM Specification Translation algorithms.

An additional repository is generated during Specification translation to hold ASN.1 operation and error identifier to IDL scoped name mappings and operation timer information. IDL operations have been defined to access information in this repository.

1.2.2.4 *Interaction Translation*

Figure 1-7 shows the basic steps invoked at a gateway between a legacy IN system, a Service Switching Point (SSP), and an IN system - a Service Control Point (SCP) implemented using CORBA to explain at a high-level the issues addressed by Interaction Translation.

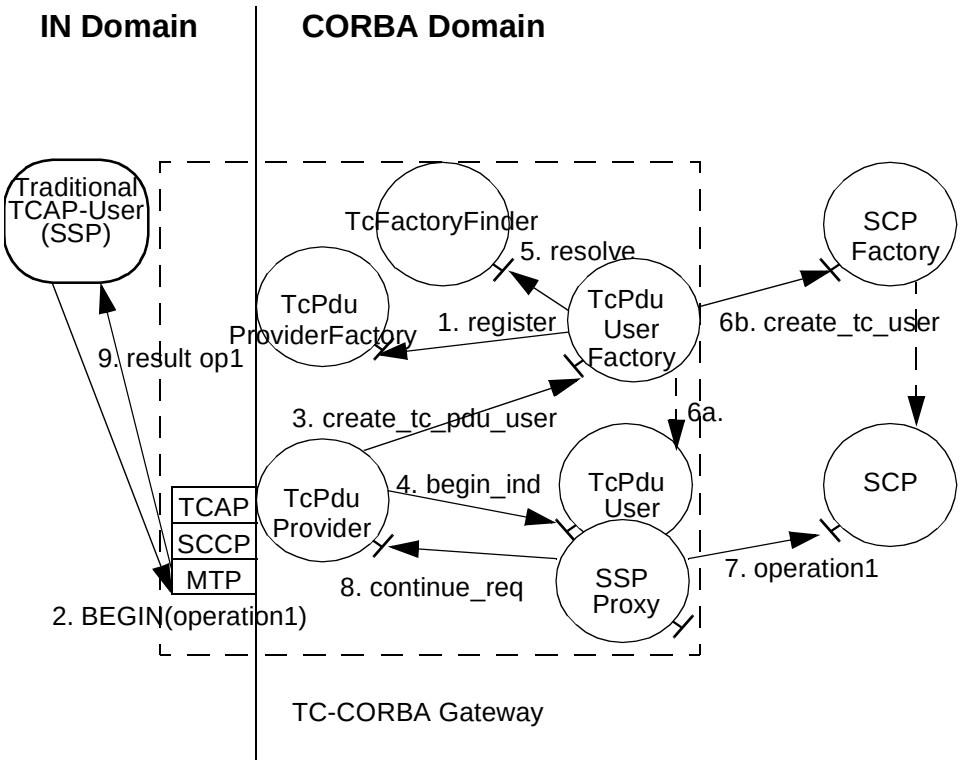


Figure 1-7 Interaction of TC-Users at an IN/CORBA Gateway

As shown in Figure 1-7, a **TcPduUserFactory** interface registers (step 1) with a **TcPduProviderFactory** object to request notification of PDUs received from the non-CORBA domain destined for a particular Global Title (GT) and, optionally, for a particular Application Context (AC). It is assumed that on the reception of a BEGIN PDU for a registered GT and AC, the **TcPduProviderFactory** creates (if an appropriate one does not exist) a suitable **TcPduProvider** object to interact with the TC/SS7 protocol stack.

Step 2 shows the reception of a TC operation, **operation1**, in a BEGIN PDU from a remote TC-User (the SSP). The **TcPduProvider** invokes a **create_tc_pdu_user** operation on the previously registered **TcPduUserFactory** (see step 3) which creates an appropriate **TcPduUser** object. The **TcPduProvider** invokes a **begin_ind**

operation on the **TcPduUser** object (see step 4), which contains within its input parameters both the TC dialog and component portions of the message as well as a call-back reference on itself.

The CORBA Naming Service is used to store name bindings using SS7 Global Titles to enable location of factories that can create TC-User/ROS-User Application Entities. A naming context is created for each Global Title, below which are naming contexts for each application context. The scoped names for the dialog initiator and responder interfaces for a particular AC are stored as naming contexts below the naming context for the application contexts. Each named object is either a default **TcUserFactory** (for cases where no Application Context is present[1]) or a specific factory associated with a particular Application Context (for those TC applications that explicitly signal the Application Context). Rather than deal directly with the CORBA Naming Service, a TC-friendly interface, **TcFactoryFinder**, provides operations to obtain and add information on the name tree. Step 5 in Figure 1-7 shows the **TcPduUserFactory** using the **TcFactoryFinder** interface to resolve the Global Title and obtain the factory references for the CORBA objects that will represent the TC-User interactions in the CORBA domain, namely the factories to create the SSP proxy object at the gateway and the SCP object.

Once a reference to the appropriate responder and initiator interface creation factories has been found in the naming service, a request can be made (to an initiator interface creation factory, not shown in the figure) to create an appropriate proxy object representing the legacy SSP at the gateway, as shown in step 6a in Figure 1-7. To initiate a TC-User interaction (with the appropriate responder interface), the responder interface creation factory (called SCP factory in the figure) must be supplied with an object reference to the corresponding initiator interface in the **create_tc_user** operation for the CORBA object corresponding to the SCP (step 6b).

Various creation parameters may be passed in the create request to constrain the created object. This initiates a new association and allows TC-like two-way communication (dialog). Operations may now be invoked (step 7) on the created TC-User object (using the CORBA Messaging Service to allow asynchronous requests and replies). Each operation carries with it TC context information that includes an Association ID. This allows a single TC-User object, such as the SCP in Figure 1-7, to simultaneously engage in multiple associations (TC dialogs). This is important for scalability and preserving response time.

Additional TC context information is carried in each operation, which may be used to regulate the use of TC flow control primitives by proxy objects at an IN/CORBA gateway. Pure CORBA to CORBA interactions may ignore this information.

Note – The support of TC flow control by TC-User/CORBA objects and TC-User proxy objects (at a gateway) is optional.

Each TC-User object has common functionality for ending and aborting associations defined in the **TcUser** interface. This allows an object to be informed of these important events.

1.Although an Application Context is implied.

All of these interfaces are common to both TC-User CORBA objects and TC-User proxy objects residing at an IN/CORBA gateway. This preserves location transparency for interactions in the CORBA domain.

1.2.3 Interfaces

The interfaces defined in this specification are as follows.

TcUser

This interface inherits from the **COS LifeCycleObject** and defines operations that are supported by all TC-User objects to create a new association, and to end or abort an existing association. It also defines a readonly attribute that identifies if TC context setting information (e.g., to indicate TC dialog handling primitives) should be included in associations with this object.

TcFactoryFinder

This interface provides "wrapper" operations for TC-User objects to register with the **CORBA Naming Service**, an operation to find such objects, and a method to explicitly replace information in the Naming Service rather than just add new entries.

TcUserGenericFactory

This interface defines generic create operations for TC-User objects. It provides a common type for all factories generated during Specification Translation.

TcServiceFinder

This is a helper interface to find references to all the TC-User CORBA services.

TcRepository

This interface provides standardized access to the information generated during specification translation providing the mappings from TC/ROS operation, extension and error codes to IDL scoped names.

GwAdmin

This interface provides a single point of contact for accessing all gateway-related functions. In particular, it defines readonly attributes for the **TcFactoryFinder** and **TcPduProvider** interfaces.

TcPduProvider

This is one of a set of TC PDU-oriented interfaces and defines a standard interface for encapsulating the TC protocol stack in the CORBA domain. Operations are modeled on the TC dialog handling request primitives defined in ITU-T Rec. Q.771 [**9**] (TC-BEGIN, TC-CONTINUE, TC-END, TC-CANCEL, TC-ABORT, etc.) as well as support operations.

TcPduUser

This is the complementary interface to the **TcPduProvider** and provides the CORBA equivalent of the TC-User implementation, which is the recipient of TC messages from, or originator of messages to, the SS7 stack.

TcPduProviderFactory

This interface allows a **TcPduUserFactory** object to register as the factory for creating the **TcPduUser** object that is to be associated with a particular Global Title and Application Context. The **TcPduProviderFactory** also permits the creation of **TcPduProvider** objects.

TcPduUserFactory

A **TcPduUserFactory** object may register with a **TcPduProviderFactory** object in order to be made available for creation of **TcPduUser** objects associated with a particular Global Title and Application Context.

The relationship of the various interfaces are illustrated using the following class diagrams in Figure 1-8 and Figure 1-9 on page 1-14.

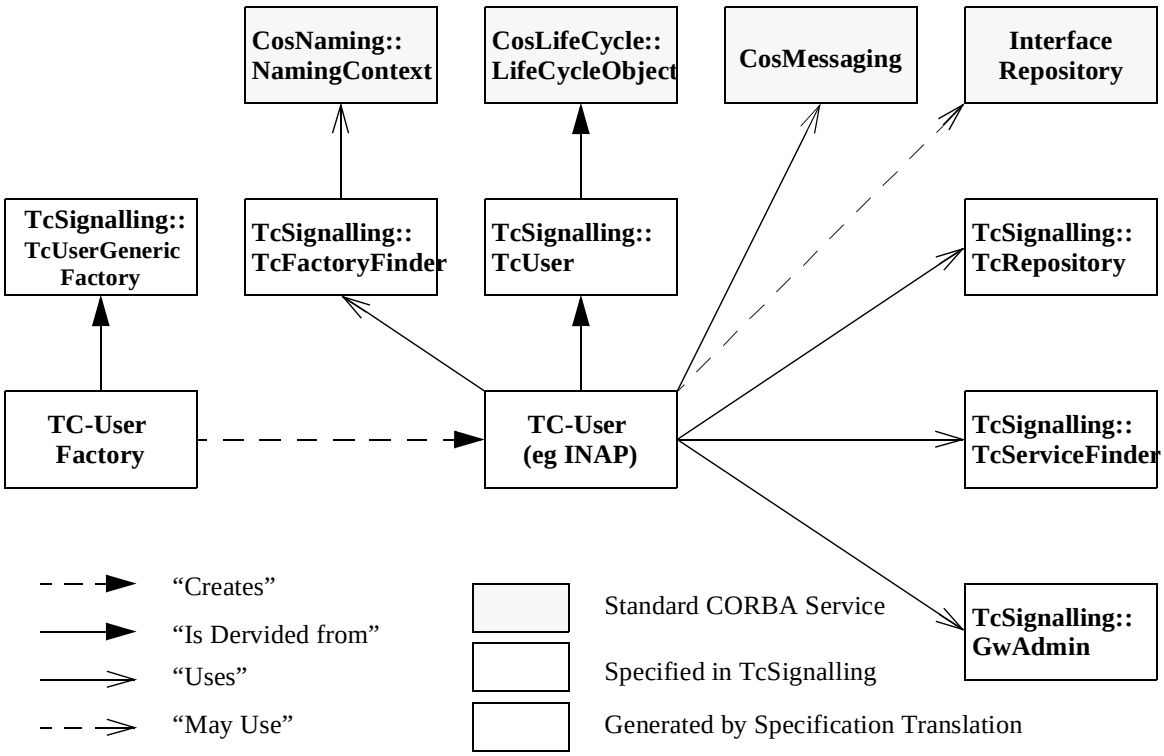


Figure 1-8 Relationship of the Various Interfaces

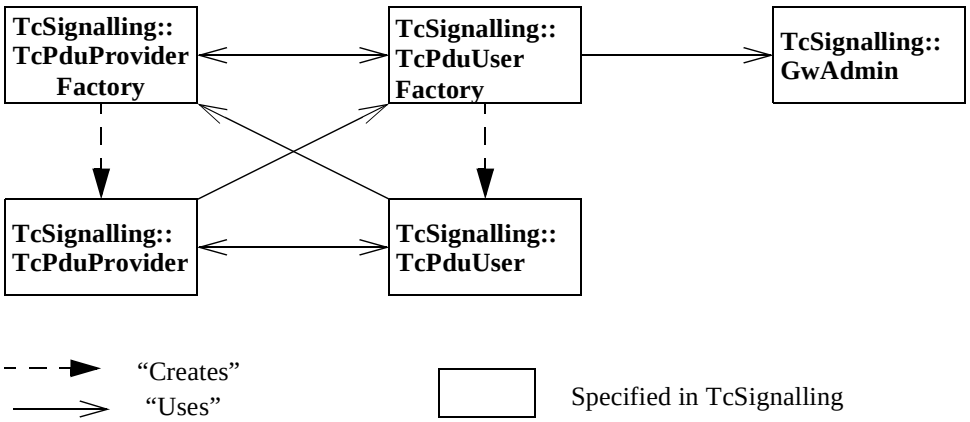


Figure 1-9 Relationship of the TC PDU Oriented Interfaces

1.2.4 GIOP Mapping onto Connectionless SCCP

To deploy CORBA technology as a central building block of a Distributed Processing Environment in specific domains, the General Inter-ORB Protocol (GIOP) and the Internet Inter-ORB Protocol (IIOP) are in many cases not sufficient. For example, in the telecommunications domain the use of GIOP/IIOP instead of existing telecommunication protocols would result in a loss of efficiency and reliability. It is important not to ignore proven networking technologies, like the reliable and efficient Signaling System No. 7 (SS.7).

SS7 consists of a set of protocols used for communication between signalling points as described in Section 1.2.2.1, "Background on TC/SS7 and IN," on page 1-4. In this set, the Signalling Connection Control Part (SCCP) provides services that are usable for object communication in a CORBA environment. It allows the reliable transport of data preserving the order of the data. Furthermore, it allows flexibility in addressing a signalling point. A signalling point may be addressed either with a signalling point code (an integer) or with a global title (a sequence of digits like a telephone number). The global title addressing mechanism is used together with global title translation. This translation mechanism transforms the global title transparent to the user into the signalling point code of the destination.

In currently deployed SS7 networks, only class 0 and class 1 SCCP are available. Both classes offer connectionless services. The class 0 service does not guarantee that the order of messages sent between a source and destination is maintained. Class 1 SCCP provides guaranteed message sequencing. As this is an essential requirement for GIOP, only class 1 SCCP can be used for mapping GIOP over SCCP. SCCP also provides means for failure detection. It informs the sender of a message in case of a failure in the message delivery.

The features available in SCCP class 1 make it suitable as a transport mechanism for GIOP messages. The definition of a GIOP mapping onto the SCCP protocol allow us to preserve large parts of current CORBA products as the generation and processing of GIOP messages is already available in most products.

The SS7 infrastructure is highly available, widely implemented, and can be used for fast communication. It is envisioned that more and more CORBA-based signalling applications will reside in CORBA-based “islands” connected through this signalling infrastructure. For instance, the information generated in a CORBA-based logging procedure could be made available to other entities in the network using Inter-ORB communication over SCCP. The introduction of such a service would not interfere with existing SCCP users. The SS7 infrastructure only needs to be changed at the originating and the terminating signalling point where the contents of the SCCP messages are processed.

For these reasons a mapping of GIOP onto the SCCP protocol is defined in the “SCCP Inter-ORB Protocol (SIOP)” chapter. The mapping provides:

- reliable and message-order-preserving transport of GIOP messages over SCCP class 1
- flexible addressing mechanism through a new IOR profile and a new IOR component
- version control
- grouping of CORBA objects through endpoint identifiers
- independence from byte ordering
- error procedures in case of misformatted data

TC/CORBA Application Interworking 2

Contents

This chapter contains the following sections.

2.1 Specification Translation

2.1.1 Introduction

As the translation of ASN.1 basic types to IDL has already been standardized by the Joint Inter-Domain Management (JIDM) Task Force of The Open Group and NMF, this specification will re-use that standardization [4] here. This specification fully supports the above mentioned JIDM Specification Translation specification, amended with the current list of errata and corrigenda to the document, as expressed in the “JIDM Specification Translation Issues List” (available from The OpenGroup and NMF web sites and from the OMG as document telecom/98-05-05). The justification for these changes is also available through the mentioned amending documentation.

Only the parts of the JIDM document dealing with the translation of ASN.1 modules and types to IDL are relevant to this specification. The relevant parts consist of Chapter 2 and supporting IDL modules. As this specification provides mappings of ROS constructs not covered by the JIDM document, there will be no discrepancy between the two.

The JIDM Translation algorithms are extended here by providing IDL mappings for five ASN.1 constructs used in TC and TC-User specifications. The JIDM translation of an ASN.1 module is also extended by specifying the generation of a repository containing all TC-User protocol operation codes, error codes, and object identifiers. This identifier mapping file is similar to the approach for identifier mapping in the SNMP portion of the *OMG TMN/CORBA Interworking* specification.

2.1.2 A Note on the "versions" of ASN.1, ROS, and TC used in this Specification

ASN.1 was first defined in 1988. Subsequently, a second "version" was provided in 1994 [10] that corrected many omissions and errors in the previous version. One major revision was the deprecation (and complete replacement) of the user-defined syntax called the MACRO notation. This was replaced with Information Object Classes, as well as notation to describe constraints, extensions, etc.

Several OSI standards that used the 1994 ASN.1 notation were completely revised to make use of the new notation. Among these were the revised ROS standards of 1994, which provided a complete replacement of the MACROs that described operations, errors, and concepts such as ASE and Application Context with Information Object Classes called OPERATION, ERROR, OPERATION-PACKAGE, and CONTRACT to define the same concepts.

It should be noted that the new ASN.1 notation allowed the definition of a user-defined syntax for Information Object Classes. This permitted the definition of a notation for operations, errors, operation packages, and contracts that closely mimicked the syntax of the earlier notation (for operations, errors, application service elements, and application contexts) without inheriting many of the defects of the earlier notation that had led to their deprecation. Thus, users of the earlier notation would find only very minor differences (e.g., position of brackets, etc.) between the earlier and the new notation.

A ROS-User protocol that took advantage of the new ASN.1 notation, as well as the new ROS information object classes, was the OSI Directory X.500 standard. All new OSI standards using ASN.1 and/or ROS are expected to use the new ASN.1 notation as well as the new ROS constructs.

Transaction Capabilities (TC) was first defined in 1988. A subsequent version was provided in 1993 [9]. The 1993 version corrected several minor errors in the 1988 version, provided a thorough revision of the state transition diagrams, and added one new feature - the optional support of protocol to convey Application Context information. There was no change to existing external interfaces, provided the additional (optional) DialogControl APDUs were not used. This would provide backward compatibility for TC users who had no need for the transfer of application context information, while offering the feature to new applications that would benefit from it.

There are several ITU-T standardized TC-User ASEs such as Mobile Application Part (MAP), Global Virtual Network Services (GVNS), International Calling Card Verification (ICCV) etc., which do not need the additional services provided by TC

1993. The ITU-T INAP, while partitioning the operations by interface (SCP-SSP, SCP-SDP etc.), each interface being assigned to several ASEs, have as yet not explicitly employed the additional functions provided by TC 1993. In addition, there are several versions of INAP based on different standards bodies (ETSI, ANSI, etc.).

By offering in this specification a generic mapping applicable to any TC-User protocol, with or without the additional functionality provided by TC 1993, we permit the building of gateways that can be used in a variety of realistic service scenarios. There is no need for service-specific gateways. Obviously all “dialects” of INAP (whether one of the standardized ITU-T IN Capability Sets, the ETSI version or the ANSI version, or indeed any proprietary version) will be able to make use of the mechanisms provided in this specification.

2.1.3 TC-User ASN.1 specification to OMG IDL Translation Algorithm

The interfaces of TC-User applications are described in ASN.1 modules. To convert these to descriptions of CORBA entities, these modules must be translated into IDL specifications. There are three main forms of description:

1. One using ASN.1 and natural language (for example ITU-T INAP CS-1)[8]
2. One using just ASN.1 (for example ETSI INAP CS-1)[11]. In this case additional ASN.1 macros are used to formally define the parts of the specification; otherwise, defined using natural language.
3. Newer specifications (for example ITU-T CS-2) may be identified as a third type as they use the revised ASN.1 notation (information objects instead of macros).

All three forms of specifications are considered in the mappings provided below.

2.1.3.1 Type I ASN.1 Description of TC-User

An older ASN.1 and natural language description (henceforth called a Type I TC-User description) has the general form:

```
MODULE ::= <module_name>
BEGIN
   IMPORTS <import_list>
   EXPORTS <export_list>
      OPERATION MACRO_1
      ...
      OPERATION MACRO_n
      ERROR MACRO_1
      ...
      ERROR MACRO_m
END
```

2.1.3.2 Type II ASN.1 description of TC-User

An older ASN.1 only description (henceforth called a Type II TC-User description) has the general form:

```
MODULE ::= <module_name>
BEGIN
   IMPORTS <import_list>
   EXPORTS <export_list>
      OPERATION MACRO_1
      ...
      OPERATION MACRO_n
      ERROR MACRO_1
      ...
      ERROR MACRO_m
      APPLICATION-SERVICE-ELEMENT MACRO_1
      ...
      APPLICATION-SERVICE-ELEMENT MACRO_x
      APPLICATION-CONTEXT MACRO_1
      ...
      APPLICATION-CONTEXT MACRO_y
END
```

2.1.3.3 Type III ASN.1 description of TC-User

A new ASN.1 only description (henceforth called a Type III TC-User description) has the general form:

```
MODULE ::= <module_name>
BEGIN
   IMPORTS <import_list>
   EXPORTS <export_list>
      OPERATION information object_1
      ...
      OPERATION information object_n
      ERROR information object_1
      ...
      ERROR information object_m
      OPERATION-PACKAGE information object_1
      ...
      OPERATION-PACKAGE information object_x
      CONTRACT information object_1
      ...
      CONTRACT information object_y
END
```

2.1.3.4 Mapping Algorithm

The JIDM Specification Translation document does not define algorithms for converting any of the ASN.1 macros or equivalent information objects used in the above specifications. This is the primary extension to the JIDM specification presented in this document. A mapping is also defined for the ASN.1 EXTENSION macro/information object as it is extensively used in several TC-user OPERATION definitions.

In general:

- An instance of an OPERATION provides zero or one RESULT definition and refers to zero, one, or more instances of an ERROR information object class/macro.
- An instance of an APPLICATION-SERVICE-ELEMENT/OPERATION-PACKAGE refers to one or more OPERATIONs.
- An instance of an APPLICATION-CONTEXT/CONTRACT refers to one or more instances of an APPLICATION-SERVICE-ELEMENT/OPERATION-PACKAGE.

The basic scheme for the static translation of a TC/ROS-User specification to a CORBA IDL specification is as follows:

1. Map a TC-User application definition contained within an ASN.1 module into an IDL module called **<module_name>** in accordance with the standard JIDM ASN.1 module to IDL module name mapping rules and all interfaces, types, and constants generated from an ASN.1 module will
   - be within the scope of the corresponding IDL module,
   - declare the imported types in the ASN.1 module as **typedef**s of imported IDL types, and
   - declare an IDL interface, called **TcUserFactory** that inherits from the interface **TcSignaling::TcUserGenericFactory**, if the current TC-User description is Type I and there is at least one instance of a ROS/TC OPERATION defined, or if the current TC-User description is Type II/III and there is at least one instance of an APPLICATION-CONTEXT/CONTRACT defined.
2. Map each of the ASN.1 types into a corresponding IDL type using the translation scheme defined in [4]. A complex ASN.1 data type (used to describe TC-User PDUs) may generate more than one IDL data type.
3. For a Type I description module that has a **TcUserFactory** interface defined:
   - Create an interface called **DefAc**.
   - Create an IDL exception definition (as described in Section 2.1.5.2, “Mapping for ERRORs,” on page 2-10) for every ERROR instance declared or imported into the module.
   - Create an IDL **return** type in the operation signature (as described in Section 2.1.5.1, “Mapping for OPERATIONs,” on page 2-8) for every OPERATION instance with a RESULT keyword declared or imported into the module.

- Create an IDL operation definition within **DefAc** (as described in Section 2.1.5.1, “Mapping for OPERATIONs,” on page 2-8) for every OPERATION instance declared or imported into the module.
- Define an operation (**create_def_ac_initiator**) within the scope of the **TcUserFactory** interface. This operation will return an object of type **<module_name>::DefAc**. This operation has no parameters.
- Define an operation (**create_def_ac_responder**) within the scope of the **TcUserFactory** interface. This operation will return an object of type **<module_name>::DefAc**. This operation has the following parameters:
  - **in** parameter of type **<module_name>::DefAc** called **initiator**
  - **in** parameter of type **TcSignaling::AssociationId** called **a_id**
  - **in** parameter of type **TcSignaling::TcContextSetting** called **tc_context_setting**
  - **out** parameter of type **TcSignaling::AssociationId** called **a_id_rtn**
  - **raises** clause containing the exceptions: TcSignaling::NoMoreAssociations, TcSignaling::UnsupportedTcContext
- [1]Define an operation (**create_def_ac_responder_with_dialog_data**) within the scope of the **TcUserFactory** interface. This operation will return an object of type **<module_name>::DefAc**. This operation has the following parameters:
  - **in** parameter of type **<module_name>::DefAc** called **initiator**
  - **in** parameter of type **TcSignaling::AssociationId** called **a_id**
  - **in** parameter of type **TcSignaling::TcContextSetting** called **tc_context_setting**
  - **in** parameter of type **string** called **protocol_version**
  - **in** parameter of type **TcSignaling::DialogUserData** called **d_u_d**
  - **out** parameter of type **TcSignaling::AssociationID** called **a_id_rtn**
  - **raises** clause containing the exceptions TcSignaling::NoMoreAssociations, TcSignaling::InvalidParameter, and TcSignaling::UnsupportedTcContext

4. For a Type II/III description module that has a **TcUserFactory** interface defined:
   - Create an IDL exception definition (as described in Section 2.1.5.2, “Mapping for ERRORs,” on page 2-10) for every ERROR instance declared or imported into the module.
   - Create an IDL return type for the operation signature (as described in Section 2.1.5.1, “Mapping for OPERATIONs,” on page 2-8) for every OPERATION instance with a RESULT keyword declared or imported into the module.
   - Create two interfaces: **<contract_name>Initiator** and **<contract_name>Responder** (as described in Section 2.1.5.4, “CONTRACT (or APPLICATION-CONTEXT) Mapping,” on page 2-12) for every APPLICATION-CONTEXT/CONTRACT instance.

1. It is unlikely that aType 1 TC-User description uses the TC Dialog portion. This create operation was put in for completeness to cover this unlikely case. The authors have found no example of such use.

- Define an operation (**create_<contract_name>_initiator**) within the scope of the **TcUserFactory** interface. This operation will return an object of type **<module_name>::<contract_name>Initiator**. This operation has no parameters.
- Define an operation (**create_<contract_name>_responder**) within the scope of the **TcUserFactory** interface. This operation will return an object of type **<module_name>::<contract_name>Responder**. This operation has the following parameters:
  - **in** parameter of type **<module_name>::<contract_name>Initiator** called **initiator**
  - **in** parameter of type **TcSignaling::AssociationId** called **a_id**
  - **in** parameter of type **TcSignaling::TcContextSetting** called **tc_context_setting**
  - **out** parameter of type **TcSignaling::AssociationID** called **a_id_rtn**
  - **raises** clause containing the exceptions: TcSignaling::NoMoreAssociations, TcSignaling::UnsupportedTcContext;
- Define an operation (**create_<contract_name>_responder_with_dialog_data**) within the scope of the **TcUserFactory** interface. This operation will return an object of type **<module_name>::<contract_name>Responder**. This operation has the following parameters:
  - **in** parameter of type **<module_name>::<contract_name>Initiator** called **initiator**
  - **in** parameter of type **TcSignaling::AssociationId** called **a_id**
  - **in** parameter of type **TcSignaling::TcContextSetting** called **tc_context_setting**
  - **in** parameter of type **string** called **protocol_version**
  - **in** parameter of type **TcSignaling::DialogUserData** called **d_u_d**
  - **out** parameter of type **TcSignaling::AssociationID** called **a_id_rtn**
  - **raises** clause with the exceptions TcSignaling::NoMoreAssociations, TcSignaling::UnsupportedTcContext, and TcSignaling::InvalidParameter

5. The compiler will generate data for an operation code repository that will provide access via the interface **TcRepository** to the following elements: IDL Scoped name for the ASN.1 operation or error, IdType (i.e., local (integer) or global (ASN.1 object identifier)), ID (i.e., the actual value defined in the ASN.1 specifications), Type (i.e., operation or error), Timer (associated with the operation).

2.1.4 Generation of TC Repository to hold ScopedName to ID Mapping

In addition to the IDL file produced during specification translation, a TC Repository is created that contains the information about mapping the IDL scoped name for interfaces and identifier constants to a corresponding TC operation, error and, if present, extension data type identifiers. This information is used to match ROS/TC operation codes, extensions, and error codes to specific IDL constructs.

The **TcRepository** interface provides access to the following information:

- *IDL scoped name of the IDL construct*: this represents the unique name in the Interface Repository of the corresponding ASN.1 module element (operations, errors and, if present, extension data types).
- *the type of identifier*: this allows constructs (usually operations or errors) to be named by either a local value or an object identifier.
- *the identifier value*: the operation or error code (or other identifier) is placed here.
- *the timer value*: this is the maximum length of an operation timer in seconds. If no timer value is formally defined for an operation then a value of 0 is used.

An example of the sort of information captured in such a repository would be:

# Scoped-Name	IdType	ID Type	Timer
Q1218_3::DefAc::initialDP	local	0 operation	0
Q1218_3::DefAc::originationAttemptAuthorized	local	1 operation	0
Q1218_3::DefAc::canceled	local	0 error	None

where Q1218_3 is the module name of an IN-specific ASN.1 module converted to IDL using specification translation.

2.1.5 Mapping of TC-User/ROS Constructs

This section defines detailed mapping rules for the individual ASN.1 constructs required for TC/ROS Specification Translation. Examples of all mappings are also provided.

2.1.5.1 Mapping for OPERATIONs

The OPERATION information object class (or MACRO in the case of Type I and II descriptions) is used to define ROS/TC-User operations within the scope of a ROS/TC-User protocol. Each instance of a ROS/TC-User OPERATION is mapped to an IDL operation signature. The mapping is defined below. Appendix E provides the ASN.1 definition for the OPERATION information object class as well as the macro.

Operation Instance Mapping

1. Map the ROS operation name to an IDL operation name according to the JIDM type name mapping rules.
2. Map the keyword ARGUMENT (which is a single ASN.1 type) to a single in argument type of the IDL operation. The ASN.1 ARGUMENT type is mapped to an IDL type using the JIDM specification translation rules.
3. Create an **inout** argument of type **TcContext** called **ctext**. This will be used to carry TC dialog handling information, invoke ids, association ids, and linked ids for operations that may have linked replies.

4. Map the ROS RESULT argument (which is a single ASN.1 type) to the IDL result type in the operation signature. The ASN.1 RESULT type is mapped to an IDL return type using JIDM specification translation rules. If there is no ASN.1 data type following the RESULT keyword or the RESULT keyword is absent, the IDL result type is a **void**.

5. If there are ERRORS defined for an operation, then add an IDL raises expression to the operation signature. For each ASN.1 ERROR type following the ERRORS keyword add its translated (see Section 2.1.5.2, “Mapping for ERRORs,” on page 2-10 item 1) IDL exception name to the list in the **raises** clause.

The **TcContext** construct is explained in the **TcUser** interface section of the Interaction Translation specification, section 4.2.1.

Operation Identifier Assignment Mapping

The operation code assignment is used to create the TC Repository as described in Section 2.1.4, “Generation of TC Repository to hold ScopedName to ID Mapping,” on page 2-7.

OPERATION Mapping Example

The example below shows how an **OPERATION** instance may be defined for a TC-User, in this case an operation defined by IN Capability Set 1 (CS-1). The argument of the operation is of type **InitialDPArg**. The operation only reports unsuccessful completion by one set of errors.

```
InitialDP ::= OPERATION
   ARGUMENT InitialDPArg
   ERRORS { MissingCustomerRecord, MissingParameter,
   SystemFailure, TaskRefused, UnexpectedComponentSequence,
   UnexpectedDataValue, UnexpectedParameter }
```

The mapping of the ASN.1 operation to an IDL operation is shown below:

```
void[2] InitialDP (in InitialDPArgType InitialDPArg, inout TcContext ctext)
    raises (MissingCustomerRecord, MissingParameter, SystemFailure,
    TaskRefused,
    UnexpectComponentSequence, UnexpectedDataValue, UnexpectedParameter);
```

The ASN.1 operation argument type **InitialDPArg** is mapped to the IDL type **InitialDPArgType** and an **in** parameter of this type is placed in the IDL operation parameter list. The IDL operation also takes an **inout** parameter of type **TcContext**, which is used to carry dialog flow control information, the operation’s invokeID and the AssociationId as context information when the operation is invoked. The ASN.1 errors are mapped to parameters of the **raises** clause in the IDL operation signature.

2. As the operation does not have a result returned, it is necessary to insert the IDL result type **void**.

2.1.5.2 Mapping for ERRORs

Each instance of a ROS ERROR information object class (or macro, for Type I and II descriptions) is mapped to an IDL user defined exception. The mapping is defined below. Appendix E provides the ASN.1 definition for the ERROR information object class as well as the macro.

Error Instance Mapping

1. Map the ROS ERROR instance's name to an IDL exception name.
2. If there is a PARAMETER defined as an ASN.1 type for the error instance, map it to a single parameter of an IDL exception using the JIDM translation rules.
3. If there is a PARAMETER defined as an unnamed ASN.1 construct, map the parameter of the error to a single IDL type with name **<ErrorName>Param** and use the JIDM translation rules for the contents of this type. This type is defined as a single parameter of the IDL exception created.
4. Create an additional parameter in the IDL exception of type **DialogFlowCtr** with name **ctr**.

The IDL for the type **DialogFlowCtr** is described in Section 2.2.2.1, "TC Context Information," on page 2-16.

Error Identifier Assignment Mapping

This form of the macro is used to create an entry in the TC Repository as described in Section 2.1.4, "Generation of TC Repository to hold ScopedName to ID Mapping," on page 2-7.

ERROR mapping example

A mapping for an instance of an error taken from IN CS-1 is shown below. The ASN.1 for an error instance named **`SystemFailure`** which conveys a single parameter **`UnavailableNetworkResource`** is as follows:

```
SystemFailure ::= ERROR
      PARAMETER  UnavailableNetworkResource
```

It maps to the following IDL:

```
exception SystemFailure (UnavailableNetworkResourceType
unavailableNetworkResource; DialogFlowCtr ctr;);
```

2.1.5.3 Mapping for the EXTENSION MACRO

The ASN.1 EXTENSION MACRO is defined in ITU-T Recommendation Q.1400 for the purpose of allowing extensions to be made to standardized TC application protocols. The use of the EXTENSION MACRO is intended only for minor extensions to an abstract syntax, for example to allow the addition of information elements that may enhance an activity but are not essential to performing the basic activity or to

allow the addition of a capability that is not essential to the base capability. All data types defined in Interface Recommendation for IN CS-1 use this extension mechanism and so an ASN.1-to-IDL mapping of the EXTENSION MACRO is required to fully support standard application protocols (e.g., INAP) at a gateway. The EXTENSION MACRO is defined below.

```
EXTENSION MACRO::=
TYPE NOTATION::= ExtensionType Criticality
VALUE NOTATION::= value (VALUE CHOICE {
                private-extension INTEGER,
                standard-extension OBJECT IDENTIFIER })
ExtensionType::= ìEXTENSION-SYNTAXî type | empty
Criticality::= ìCRITICALITYî value (Criticality Type)
CriticalityType::= ENUMERATED { ignore(0), abort(1) }
```

EXTENSION instance mapping

Every EXTENSION Macro instance will result in the creation of two IDL types, an extension criticality identifier and an extension type.

1. The instance of the EXTENSION macro EXTENSION SYNTAX field is mapped to an IDL type according to the standard JIDM ASN.1 type mapping rules. If the ASN.1 Type is named, the standard JIDM mapping to an IDL type is performed, if unnamed it is given the same name as the extension (barring the need for JIDM disambiguation).

2. The CRITICALITY field of the macro is mapped to an IDL constant declaration of the **ASN1_ExtensionCriticality** type with an identifier equal to the extension name appended with the string “ExtensionCriticality” and an appropriate value. If there is no CRITICALITY declaration, an IDL type with criticality **ASN1_EXTENSION_ABORT** is declared as this is the default.

3. The macro VALUE declaration and the **ScopedName** of the IDL Extension Type is placed in the TC Repository.

EXTENSION mapping example

The following example is taken from ETSI INAP CS-1 (where it is used as an example of the EXTENSION macro).

```
SomeNetworkSpecificIndicator ::= EXTENSION
EXTENSION SYNTAX BOOLEAN
CRITICALITY abort
someNetworkSpecificIndicator SomeNetworkSpecificIndicator ::= 1
```

The IDL this maps to is given below:

```
typedef ASN1_Boolean SomeNetworkSpecificIndicator;
const ASN1_ExtensionCriticality
SomeNetworkSpecificIndicatorExtensionCriticality =
ASN1_EXTENSION_ABORT;
```

2.1.5.4 CONTRACT (or APPLICATION-CONTEXT) Mapping

The CONTRACT information object class (or APPLICATION-CONTEXT macro in the case of Type II descriptions) is used to define the rules of engagement for a ROS/TC-User protocol. Each instance of a ROS CONTRACT is mapped to an IDL interface. The mapping is defined below. Appendix E provides the ASN.1 definition for the CONTRACT information object class (as well as the APPLICATION-CONTEXT macro).

Contract (or Application Context) Instance Mapping

When mapping a Type III (respectively Type II description), an instance of a CONTRACT (respectively APPLICATION-CONTEXT macro) is used to create IDL interface definitions as follows:

1. Create one interface named **<contract_name>Responder**, and within this interface create IDL operation signatures (as described in "Operation Instance Mapping" on page 2-8) for each ROS operation identified by the keyword:
   - CONSUMER INVOKES for each OPERATION-PACKAGE (or ASE) identified by the keyword INITIATOR CONSUMER OF for the CONTRACT/APPLICATION-CONTEXT instance definition.
   - SUPPLIER INVOKES for each OPERATION-PACKAGE (or ASE) identified by the keyword RESPONDER CONSUMER OF for the CONTRACT/APPLICATION-CONTEXT instance definition.
   - OPERATIONS for each OPERATION-PACKAGE (or ASE) identified by either of the keywords INITIATOR CONSUMER OF or RESPONDER CONSUMER OF for the CONTRACT / APPLICATION-CONTEXT instance definition.
   - OPERATIONS, CONSUMER INVOKES, and SUPPLIER INVOKES for each OPERATION-PACKAGE (or ASE) identified by the keyword OPERATIONS OF for the CONTRACT/APPLICATION-CONTEXT instance definition.
2. Create one interface named **<contract_name>Initiator**, and within this interface create IDL operation signatures (as described in "Operation Instance Mapping" on page 2-8) for each ROS operation identified by the keyword:
   - CONSUMER INVOKES for each OPERATION-PACKAGE (or ASE) identified by the keyword RESPONDER CONSUMER OF for the CONTRACT/APPLICATION-CONTEXT instance definition.
   - SUPPLIER INVOKES for each OPERATION-PACKAGE (or ASE) identified by the keyword INITIATOR CONSUMER OF for the CONTRACT/APPLICATION-CONTEXT instance definition.
   - OPERATIONS for each OPERATION-PACKAGE (or ASE) identified by either of the keywords INITIATOR CONSUMER OF or RESPONDER CONSUMER OF for the CONTRACT / APPLICATION-CONTEXT instance definition.
   - OPERATIONS, CONSUMER INVOKES, and SUPPLIER INVOKES for each OPERATION-PACKAGE (or ASE) identified by the keyword OPERATIONS OF for the CONTRACT/APPLICATION-CONTEXT instance definition.

3. Define an operation (**create_<interface_name>_responder**) within the scope of the **TcUserFactory** interface for each of the above-mentioned responder interfaces.*

4. Define an operation (**create_<interface_name>_responder_with_dialog_data**) within the scope of the **TcUserFactory** interface for each of the above-mentioned responder interfaces.*

5. Define an operation (**create_<interface_name>_initiator**) within the scope of the **TcUserFactory** interface for each of the above-mentioned initiator interfaces.*

* Refer to Section 2.1.3.4, “Mapping Algorithm,” on page 2-5 (item 4) for more information.

Contract (or Application Context) Mapping Example

The example below shows how a ROS/TC Contract is defined in ASN.1. The Application Context instance **`exampleContext1`** includes the OPERATION-PACKAGE instance **`examplePackage1`**, which is also defined below. The dialog initiator may invoke operations **`exampleOp1`** and **`exampleOp2`**. The dialog responder may not invoke any operation.

```
exampleContext1   CONTRACT ::=
   {
   INITIATOR CONSUMER OF {examplePackage1}
   ID objectIdentifierOfexampleContext1
   }
```

where:

```
examplePackage1 OPERATION-PACKAGE ::=
{
   CONSUMER INVOKES {exampleOp1 | exampleOp2}
   ID objectIdentifierOfexamplePackage1
}
```

The mapping of the Contract to an IDL interface is as follows:

```
interface exampleContext1Initiator {
    <resultType> exampleOp1(<params>);
    <resultType> exampleOp2(<params>);
};
```

Only one interface, **exampleContext1Initiator**, is generated in this case. Within this interface two operation signatures are generated. These operations correspond to the two ROS operations listed in the CONSUMER INVOKES clause of the **`examplePackage1`** operation package, which is then listed in the INITIATOR CONSUMER OF clause of the **`exampleContract1`** contract definition. There is no responder interface generated for this example as only the initiator is a consumer of the **`examplePackage1`** operation package and the operation package definition specifies that only the CONSUMER INVOKES.

Example

A real example taken from the ETSI CORE INAP CS-1 [11] is as follows. The example shows an Application Context definition taken from ETSI INAP CS-1. The **core-INAP-CS1-IP-to-SCP-AC** defines the rules of engagement between an SCP and an Intelligent Peripheral (IP). The initiator of this association plays the role of the consumer of the operation packages **SCF-SRF-activation-of-assist-ASE**, **Timer-ASE**, **Specialized-resource-control-ASE**, **Cancel-ASE**, **Activity-test-ASE**. Note that each of these ASEs are defined showing which operations the initiator (playing the role of the consumer) invokes and those that it performs.

```
core-INAP-CS1-IP-to-SCP-AC APPLICATION-CONTEXT
- - dialog initiated by IP with AssistRequestInstructions
INITIATOR CONSUMER OF { SCF-SRF-activation-of-assist-ASE,
Timer-ASE, Specialized-resource-control-ASE, Cancel-ASE,  Activity-test-ASE }
::= {ccitt(0) identified-organization(4) etsi(0) inDomain(1) in-network(1)
ac(1) cs1-ip-to-scp(2) version1(0)};

SCF-SRF-activation-of-assist-ASE ::=
APPLICATION-SERVICE-ELEMENT
                          - - consumer is SSF/SRF
                          CONSUMER INVOKES { assistRequestInstructions }

Timer-ASE ::= APPLICATION-SERVICE-ELEMENT
                          - - supplier is SCF
                          SUPPLIER INVOKES { resetTimer }

Specialized-resource-control-ASE  ::=
APPLICATION-SERVICE-ELEMENT
- - consumer is SSF/SRF
                          CONSUMER INVOKES {specializedResourceReport }
                          SUPPLIER INVOKES {playAnnouncement,
                          promptAndCollectUserInformation}
Cancel-ASE  ::= APPLICATION-SERVICE-ELEMENT
                          - - supplier is SCF
                       SUPPLIER INVOKES {Cancel}

Activity-test-ASE  ::= APPLICATION-SERVICE-ELEMENT
- - supplier is SCF
                          SUPPLIER INVOKES {activityTest}
```

The corresponding IDL that is generated is as follows:

```
interface Core_INAP_CS1_IP_to_SCP_ACInitiator:TcSignaling::TcUser {
... resetTimer( ... ) raises( ... );
... playAnnouncement( ... ) raises( ... );
... promptAndCollectUserInformation( ... ) raises( ... );
... Cancel( ... ) raises( ... );
... activityTest( ... ) raises( ... );
}; // end Core_INAP_CS1_IP_to_SCP_ACInitiator

interface Core_INAP_CS1_IP_to_SCP_ACResponder:TcSignaling::TcUser {
... assistRequestInstructions ( ... ) raises( ... );
```

```
... specializedResourceReport( ... ) raises( ... );
}; // end Core_INAP_CS1_IP_to_SCP_ACResponder

interface TcUserFactory:TcSignaling::TcUserGenericFactory {


Core_INAP_CS1_IP_to_SCP_ACResponder
create_Core_INAP_CS1_IP_to_SCP_ACResponder(
    in Core_INAP_CS1_IP_to_SCP_ACInitiator initiator,
    in TcSignaling::AssociationId a_id,
    in TcSignaling::TcContextSetting tc_context_setting,
    out TcSignaling::AssociationId a_id_rtn)
raises (NoMoreAssociations, UnsupportedTcContext);

Core_INAP_CS1_IP_to_SCP_ACResponder
create_Core_INAP_CS1_IP_to_SCP_A_Responder_with_dialogdata(
    in Core_INAP_CS1_IP_to_SCP_ACInitiator initiator,
    in TcSignaling::AssociationId a_id,
    in TcContextSetting tc_context_setting,
    in string protocol_version,
    in TcSignaling::DialogUserData d_u_d,
    out TcSignaling::AssociationId a_id_rtn)
raises (NoMoreAssociations, InvalidParameter, UnsupportedTcContext);

    Core_INAP_CS1_IP_to_SCP_ACInitiator
create_Core_INAP_CS1_IP_to_SCP_ACInitiator();

}; // end TcUserFactory
```

2.2 Interaction Translation

2.2.1 Introduction

A set of CORBA interfaces are described to support TC-User to TC-User interactions between either CORBA-based implementations or a mixture of CORBA-based implementations and Proxy CORBA objects that communicate with traditional TC-User implementations. This includes the preservation of location transparency and maximum re-use of standard CORBA Services for CORBA-based implementations.

The main TC concept modeled is the dialog which has no parallel in the CORBA domain. A TcSignaling Association is defined which may be mapped onto a TC dialog. In order to insulate CORBA objects from the details of TC dialogs:

- starting an association is constrained by the use of the CORBA Naming Service to find a suitable **TcUserFactory** interface and then invoking an appropriate **create** operation with suitable parameters. This models the operation of the TC-BEGIN dialog primitive and associated semantics.

- each association takes place between a pair of interacting CORBA objects. This does not preclude the use of a specific CORBA object in multiple, simultaneous associations as each operation within an association carries with it an **AssociationId** referring to a specific association.

As a way of showing the interactions between TC-user implementations, a number of message sequence charts are included in Section 2.2.3, “Application Location and Association Initiation,” on page 2-28. These graphically illustrate the interactions between objects and show some of the timing constraints involved.

To get location-independent interaction of TC-user applications, four major interaction features are supported:

1. application location (finding)
2. association initiation
3. association maintenance
4. operation invocation

In general, application location and association initiation are provided by the CORBA Naming Service and the Life Cycle Service, association maintenance is provided by the base interfaces of all TC-user CORBA objects and operation invocation is provided by specification translation, the ORB, the Messaging Service, and optionally the Interface Repository and TC Repository.

2.2.2 The Base TC-User Interfaces

There are two interfaces defined in the **TcSignaling** IDL module that serve as parent interfaces for TC-User objects and TC-User object Factories. Their functionality is described in this section. Each TC dialog maps to one association between two TC-User objects.

2.2.2.1 TC Context Information

In the CORBA domain, all TC-User interactions take place within the context of a TcSignaling Association. During Specification Translation all operations have an **inout** parameter **TcContext** added to their operation signature in addition to any types carried in the ROS/TC operation definition. This parameter contains the association ID, invoke ID, TC dialog flow control information, and the linked ID (where needed). Exceptions generated during Specification Translation have a TC dialog flow control parameter in addition to any parameter in the ROS/TC error definition. Each of these constructs is defined below.

```
typedef string ScopedName;
typedef CosNaming::Istring Istring;
typedef long AssociationId;
typedef long InvokeId;
// Range -128 to +127 for Q.773
// InvokeId values
const InvokeId NO_ID = 2000000000;
```

```
typedef short DialogFlowCtr;
// DialogFlowCtr values
const DialogFlowCtr BEGIN = 0;
const DialogFlowCtr CONTINUE = 1;
const DialogFlowCtr END = 2;
const DialogFlowCtr QUEUE_COMPONENT = 3;
const DialogFlowCtr UNIDIRECTIONAL = 4;
const DialogFlowCtr NOT_SPECIFIED = 5;

typedef short TcContextSetting;
// TcContextSetting values
const TcContextSetting TC_CONTEXT_BASE = 0;
const TcContextSetting TC_CONTEXT_NO_FLOW = 1;
const TcContextSetting TC_CONTEXT_ALL = 2;

struct TcContext {
    DialogFlowCtr ctr;
    InvokeId ivk_id;
    InvokeId lnk_id;
    AssociationId a_id;
};
```

typedef string ScopedName;;

ScopedName is used to hold the Scoped Name in the Interface Repository of an operation, extension, or error generated during ST.

typedef long AssociationId;

AssociationId is used to specify a unique identifier for a TC dialog between two TC-User objects.

typedef Istring TcAddress;

TcAddress is used to specify a SS7 Global Title

typedef long InvokeId;

InvokeId is used to specify a TC operation invoke ID. A TC-User object may place the value of NO_ID in this field to indicate that no invoke ID is specified. (Note that this is incompatible with support for TC linked operations.) Management of Invoke IDs follows that of Q.774 (93). The relevant text (3.2.1.1.2) is quoted here for completeness;

“Invoke IDs are assigned by the invoking end at operation invocation time. A TC-user need not wait for one operation to complete before invoking another. At any point in time, a TC-user may have any number of operations in progress at a remote end (although the latter may reject an invoke component for lack of resources).”

Each invoke ID value is associated with an operation invocation and its corresponding invoke state machine. Management of this invoke ID state machine takes place only at the end which invokes the operation invocation, and does not manage a state machine

for this invoke ID. Note that both ends may invoke operations in a full-duplex manner: each end manages state machines for the operations it has invoked, and is free to allocate invoke IDs independently of the other.

An invoke ID value may be reallocated when the corresponding state machine returns to idle. However immediate reallocation could result in difficulties when certain abnormal situations arise. A released ID value (when the state machine returns to idle) should therefore not be reallocated immediately; the way this is done is implementation-dependent, and thus is not described in this Recommendation."

typedef short DialogFlowCtr;

DialogFlowCtr is used to convey explicit TC flow control information for interactions between TC-Users. Normally TC-User objects interact without considering possible PDU flows from proxy objects, but some applications (e.g., INAP) may wish to explicitly trigger PDU creation at a gateway. TC allows this via dialog flow control primitives.

Normally **DialogFlowCtr** values are ignored by non-Proxy object implementations, although they may additionally be used to indicate the end of associations with a TC-User object, which supports multiple associations. A **DialogFlowCtr** type accompanies every IDL operation and exception defined in Specification Translation. All non-negative values are reserved for use in OMG specifications. Any negative value of **DialogFlowCtr** is considered a vendor extension.

The meaning of the values for TC-User objects are as follows:

- **BEGIN**: treat this operation, result, or exception as both the appropriate TC component type and a TC dialog handling begin primitive.
- **CONTINUE**: treat this operation, result, or exception as both the appropriate TC component type and a TC dialog handling continue primitive.
- **END**: treat this operation, result, or exception as both the appropriate TC component type and a TC dialog handling end primitive. This may also be used to signal the end of the current dialog to a TC-User CORBA object that supports multiple associations when included as a parameter to an operation, result, or exception.
- **QUEUE_COMPONENT**: queue the current operation, result, or exception until a subsequent operation or exception with a suitable value for this parameter triggers the sending of a PDU.
- **UNIDIRECTIONAL**: treat this operation as both a TC invoke component and a TC dialog handling unidirectional primitive. This is treated like an END parameter (i.e., it produces a short association that only lasts for the current operation). This value is only valid for operations and not operation results or exceptions.
- **NOT_SPECIFIED**: perform whatever default PDU sequencing behavior is configured at a proxy object in a gateway. This is the default setting for CORBA object to CORBA object interactions when **TcContextSetting** is set to **TC_CONTEXT_BASE** or **TC_CONTEXT_NO_FLOW**.

```
typedef short TcContextSetting;
```

The structure **TcContext** is used to carry TC-specific context information in the CORBA domain. This structure and **DialogFlowCtr** are used as parameters in generated IDL operations, results, and exceptions. Many CORBA-based TC applications may not directly use this information. When a TC-User object is created, a parameter of type **TcContextSetting** shall be supplied. There are three levels of TC context parameter support identified:

1. **TC_CONTEXT_BASE**: no support for **AssociationId** or **DialogFlowCtr**. **InvokeId** for both invoke IDs (**ivk_id**) and linked IDs (**lnk_id**) is supported. The parameter **a_id** shall be zero in all operations and operation results. The parameter **ctr** has the value **NOT_SPECIFIED** in all operations, operation results, and exceptions. This setting does not allow the support of multiple associations with one TC-User object. All TC-User objects must support this value.
2. **TC_CONTEXT_NO_FLOW**: no support for **DialogFlowCtr**. The parameter **ctr** has the value **NOT_SPECIFIED** in all operations, operation results, and exceptions. Invoke IDs, Linked IDs, and **AssociationId**s are supported. TC-User objects may support this value.
3. **TC_CONTEXT_ALL**: All **TcContext** information is supported. TC-User objects may support this value.

All non-negative values are reserved for use in OMG specifications. Any negative value of **TcContextSetting** is considered a vendor extension. When a proxy object at a gateway receives an operation, result, or exception with default values, the proxy object implementation adds any additional information required for generation of TC/SS7 PDUs.

```
struct TcContext {
    DialogFlowCtr ctr;
    InvokeId ivk_id;
    InvokeId lnk_id;
    AssociationId a_id;
};
```

TcLinkedContext is used to to hold the **AssociationId**, **InvokeId**, Linked ID (**lnk_id**), and **DialogFlowCtr** information for a TC-User operation. A **lnk_id** is used to signal a linked operation invocation; therefore, it must be equal to the **InvokeId** of the linked-to operation. Normally it has the value NO_ID. The use of context information by a TC-User depends upon the **TcContextSetting** creation parameter of that object.

2.2.2.2 The TcUser Interface

Every TC-User object derives from the **TcUser** interface. It provides functionality common to all TC-User objects. The IDL for this interface is shown below.

```
exception UnknownAssociation{};
exception NoMoreAssociations{};
```

```
exception InvalidParameter{};

interface TcUser:CosLifeCycle::LifeCycleObject {

    void abort_association(in AssociationId a_id)
        raises (UnknownAssociation);

    void abort_association_with_data(in AbortValue abort_value,
                    in AssociationId a_id)
        raises (UnknownAssociation, InvalidParameter);

    void end_association (in AssociationId a_id)
        raises (UnknownAssociation);

    AssociationId new_association(
        in TcUser initiator,
        in AssociationId a_id)
    raises(NoMoreAssociations);

    AssociationId new_association_with_dialog_data(
        in TcUser initiator,
        in AssociationId a_id,
        in string protocol_version,
        in DialogUserData d_u_d)
    raises(NoMoreAssociations, InvalidParameter);

readonly attribute TcContextSetting tc_context_setting;

}; //end TcUser
```

TcUser is derived from **CosLifeCycle::LifeCycleObject** although conformant implementations need only implement the remove operation from that interface. A CORBA No_Implement exception is thrown when move or copy are invoked.

By invoking operations on the **TcUser** interface, objects are able to:

- Abort a current association to the TC-User object.
- End gracefully a current association to the TC-User object[3].
- Allocate another association to an existing TC-User object.
- Check to see which **TcContext** parameters the object supports.

It is anticipated that many TC-User objects will support more than one simultaneous association to enhance scalability.

```
void abort_association (in AssociationId a_id)
    raises (UnknownAssociation);
```

3. This operation would be used to end an association without sending the result of an invocation.

The **abort_association** operation is used to inform the TC-user object that the association referenced by the parameter **AssociationId** has been aborted. All associated operations are also aborted. The UnknownAssociation exception is raised if the TC-user object has no current open association with an **AssociationId** equal to the one provided.

```
void abort_association_with_data (in AbortValue abort_value,
                                  in AssociationId a_id)
    raises (UnknownAssociation, InvalidParameter);
```

The **abort_association_with_data** operation is used to inform the TC-user object that the association referenced by the parameter **AssociationId** has been aborted. All associated operations are also aborted. Additional TC abort information may be supplied in the **AbortValue** parameter. The UnknownAssociation exception is raised if the TC-user object has no current open dialog with an **AssociationId** equal to the one provided. The InvalidParameter exception is raised if the TC-user object cannot process the **abort_value** parameter.

```
void end_association (in AssociationId a_id)
    raises (UnknownAssociation);
```

The **end_association** operation is used to inform the TC-user object that the association referenced by the parameter **a_id** has been ended normally. The UnknownAssociation exception is raised if the TC-user object has no current open association with an **AssociationId** equal to the one provided.

```
AssociationId new_association(
        in TcUser initiator,
        in AssociationId a_id)
    raises(NoMoreAssociations);
```

The **new_association** operation is used to inform the TC-user object that a new association with it has been initiated. A reference to the corresponding interface of the initiator is supplied by the **in** parameter **initiator**. The **AssociationId** to be used for all operations, exceptions, and results returned to the initiator is included in the **a_id** parameter. The **AssociationId** to be used by the initiator for all operations, exceptions, and results sent to the responder is supplied as a return parameter. When a new association is created in this way, it has the same **TcContextSetting** originally supplied when creating the object. The NoMoreAssociations exception is raised if the TC-user object cannot handle any more associations at the current time.

```
AssociationId new_association_with_dialog_data(
        in TcUser initiator,
        in AssociationId a_id,
        in string protocol_version,
        in DialogUserData d_u_d)
       raises(NoMoreAssociations, InvalidParameter);
```

The **new_association_with_dialog_data** operation is used to inform the TC-user object that a new association including **TC DialogPortion** data has been initiated by the calling object. A reference to the corresponding interface of the initiator is supplied

by the **in** parameter **initiator**. The **AssociationId** to be used for all operations, exceptions, and results returned to the initiator is included in the **a_id in** parameter. The **AssociationId** to be used by the initiator for all operations, exceptions, and results sent to the responder is supplied as a return parameter. When a new association is created in this way, it has the same **TcContectSetting** originally supplied when creating the object. The NoMoreAssociations exception is raised if the TC-user object cannot handle any more associations at the current time. The InvalidParameter exception is raised if the TC-user object is incapable of interpreting either the **protocol_version** or **d_u_d** parameters.

readonly attribute TcContextSetting tc_context_setting;

The **tc_context_setting** attribute has the value of **TcContextSetting** supplied to the object at creation time.

2.2.2.3 The TcUserGenericFactory Interface

The **TcUserGenericFactory** interface is provided as the basis for all TC-User object creation factories. The IDL definition for this interface is provided below.

As a **TcUserFactory** controls the creation of TC-User objects, many strategies for optimum scalability and minimum response time may be adopted. For example, a factory could create a pool of objects for which there is consistent demand then rather than creating a new object when a create request is received, it informs one of the objects in the pool of the new association and passes back the cached object reference of that object.

A **TcUserFactory** could also administer CORBA Security permissions to ensure that objects are only invoked upon by bona fide association initiators.

```
interface TcUserGenericFactory{

    TcUser create_tc_user_responder(
            in ScopedName resp_iface,
            in TcUser initiator,
            in AssociationId a_id,
            in TcContextSetting tc_context_setting,
            out AssociationId a_id_rtn)
        raises(CosLifeCycle::NoFactory,
            NoMoreAssociations, UnsupportedTcContext);

    TcUser create_tc_user_responder_with_dialog_data(
            in ScopedName resp_iface,
            in TcUser initiator,
            in AssociationId a_id,
            in string protocol_version,
            in DialogUserData d_u_d,
            in TcContextSetting tc_context_setting,
            out AssociationId a_id_rtn)
        raises(CosLifeCycle::NoFactory, NoMoreAssociations,
```

```
            InvalidParameter, UnsupportedTcContext);

    TcUser create_tc_user_initiator(
            in ScopedName init_iface)
        raises(CosLifeCycle::NoFactory);

}; // end TcUserGenericFactory
```

By invoking operations on the **TcUserGenericFactory** interface, objects are able to:

- Create TC-User responder objects and initiate associations with them.
- Create TC-User responder objects and initiate associations with them using dialog data to support TC93 applications.
- Create TC-User initiator objects.

All specific **TcUserFactory** interfaces inherit from this interface.

```
TcUser create_tc_user_responder(
            in ScopedName resp_iface,
            in TcUser initiator,
            in AssociationId a_id,
            in TcContextSetting tc_context_setting,
            out AssociationId a_id_rtn)
        raises(CosLifeCycle::NoFactory,
            NoMoreAssociations, UnsupportedTcContext);
```

The **create_tc_user_responder** operation is used to create a TC-User responder object and begin an association with it. A **ScopedName** describing the object to be created is supplied as the **in** parameter **resp_iface**. A reference to the initiator object is supplied as the **in** parameter **initiator**. The **in** parameter **a_id** holds the initiator's association ID for the association started. The responder's **AssociationId** for the association is supplied in the **out** parameter **a_id_rtn**. The **in** parameter **tc_context_setting** is used to define the TC context information to be used in the association.

- The NoFactory exception is raised if the generic factory cannot create the object requested.
- The NoMoreAssociations exception is raised if the factory cannot create any more requestor objects at the current time.
- The UnsupportedTcContext exception is raised if the value of **TcContextSettings** requested is not supported by the responder object created by the factory.

```
TcUser create_tc_user_Responder_with_dialog_data(
            in ScopedName resp_iface,
            in TcUser initiator,
            in AssociationId a_id,
            in string protocol_version,
```

```
            in DialogUserData d_u_d,
            in TcContextSetting tc_context_setting,
            out AssociationId a_id_rtn)
        raises(CosLifeCycle::NoFactory, NoMoreAssociations,
            InvalidParameter, UnsupportedTcContext);
```

The **create_tc_user_responder_with_dialog_data** operation is used to create a TC-User responder object and begin an association with it using TC dialog data. A **ScopedName** describing the object to be created is supplied as the **in** parameter **resp_iface**. A reference to the initiator object is supplied as the **in** parameter **initiator**. The **in** parameter **a_id** holds the initiator's association ID for the association started. The responder's **AssociationId** for the association is supplied in the **out** parameter **a_id_rtn**. The **protocol_version** parameter carries the TC-User protocol version information. The **d_u_d** BEGIN user data. The **in** parameter **tc_context_setting** is used to define the TC context information to be used in the association.

- The NoFactory exception is raised if the generic factory cannot create the object requested.
- The NoMoreAssociations exception is raised if the factory cannot create any more requestor objects at the current time.
- The InvalidParameter exception is raised if the factory cannot interpret either the **protocol_version** or **d_u_d** parameters.
- The UnsupportedTcContext exception is raised if the value of **TcContextSettings** requested is not supported by the responder object created by the factory.

```
TcUser create_tc_user_initiator(
            in ScopedName init_iface)
        raises(CosLifeCycle::NoFactory);
```

The **create_tc_user_initiator** operation is used to create a TC-User initiator object. A **ScopedName** describing the object to be created is supplied as the **in** parameter **init_iface**. The NoFactory exception is raised if the generic factory cannot create the object requested.

2.2.2.4 The TcFactoryFinder Interface

Every TC/CORBA gateway will provide a **TcFactoryFinder** interface that provides various helper operations which in effect provide a "wrapper" to the CORBA Naming Service. The **TcFactoryFinder** interface is available through the **GwAdmin** interface. The IDL for this interface is shown below.

```
interface TcFactoryFinder{

    void bind(in TcAddress addr,
            in ApplicationContext a_c,
```

```
            in ScopedName resp_iface,
            in ScopedName init_iface,
            in TcUserGenericFactory resp_tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName,
            CosNaming::NamingContext::AlreadyBound);

    void unbind(in TcAddress addr,
            in ApplicationContext a_c,
            in ScopedName resp_iface,
            in ScopedName init_iface,
            in TcUserGenericFactory resp_tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);

    void rebind(in TcAddress addr,
            in ApplicationContext a_c,
            in ScopedName resp_iface,
            in ScopedName init_iface,
            in TcUserGenericFactory resp_tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);

    TcUserGenericFactory resolve(in TcAddress addr,
            in ApplicationContext a_c,
            out ScopedName resp_iface,
            out ScopedName init_iface)
        raises (CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);

};  // end TcFactoryFinder
```

This interface allows TC-User objects to easily:

- register with the CORBA Naming Service,
- resolve the name of a TC-User object, and
- explicitly replace information in the Naming Service rather than just add new entries.

```
void bind(in TcAddress addr,
            in ApplicationContext a_c,
            in ScopedName resp_iface,
            in ScopedName init_iface,
            in TcUserGenericFactory tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
```

CosNaming::NamingContext::InvalidName,
CosNaming::NamingContext::AlreadyBound);

The **bind** operation is used to publish a new TC-user object factory address in the CORBA Naming Service. This binds a specific **TcUserFactory**'s object reference to a TC/SS.7 address consisting of a Global Title and optionally an Application Context. See Section 2.2.3, "Application Location and Association Initiation," on page 2-28 for details of the stucture of TC/SS.7 address information in the naming service.

The **addr** parameter holds the Global Title.

The **a_c** parameter holds the Application Context name or the value "**DefAc**" if no Application Context is to be specified.

The **resp_iface** parameter holds the TC-User responder interface's IDL scoped name (corresponding to the interface type created by the registering factory).

The **init_iface** parameter holds the TC-User initiator interface's IDL scoped name (corresponding to the interface type created by the registering factory). In the case of a Type I definition (symmetric initiator and responder interfaces) the values of both **resp_iface** and **init_iface** are the same.

The **tc_user_factory** parameter holds the reference to the registering factory. The exceptions thrown are the same as the **CosNaming::NamingContext::bind** operation and have the same meaning. For IN implementations, this operation may be seen as deploying a service (the TC-User object created by the registering factory).

```
void unbind(in TcAddress addr,
            in ApplicationContext a_c,
            in ScopedName resp_iface,
            in ScopedName init_iface,
            in TcUserGenericFactory tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
```

The **unbind** operation is used to remove a binding between a **TcUserFactory**'s object reference and a TC/SS.7 address. Only the singleton Cos Naming nodes referenced in the operation parameters are removed. The parameters have the same meaning as in the **bind** operation above. The exceptions have the same meaning as when thrown by the **CosNaming::NamingContext::unbind** operation. For IN implementations, this operation may be seen as withdrawing an already deployed service (the TC-User object created by the registering factory).

```
void rebind(in TcAddress addr,
            in ApplicationContext a_c,
            in ScopedName resp_iface,
            in ScopedName init_iface,
            in TcUserGenericFactory tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
            CosNaming::NamingContext::CannotProceed,
            CosNaming::NamingContext::InvalidName);
```

The **rebind** operation is used to change the object reference and optionally the responder and initiator interfaces already bound to a TC/SS.7 address. The **addr** and **a_c** parameters form the already existing TC/SS.7 address. The other parameters have the same meanings as in the **bind** operation. The exceptions are identical to those raised by the **CosNaming::NamingContext::rebind** operation and have the same meaning.

```
TcUserGenericFactory resolve(in TcAddress addr,
        in ApplicationContext a_c,
        out ScopedName resp_iface,
        out ScopedName init_iface)
    raises (CosNaming::NamingContext::NotFound,
        CosNaming::NamingContext::CannotProceed,
        CosNaming::NamingContext::InvalidName);
```

The **resolve** operation takes as input a TC/SS.7 address, a Global Title, and optionally an Application Context and returns a reference to a **TcUserGenericFactory** and the names of the responder and initiator interfaces associated with the objects created by that factory. The **in** parameter **addr** holds the Global Title to be resolved. The **in** parameter **a_c** holds the Application Context to be resolved or the value "**DefAc**" for the default Application Context. The returned **TcUserGenericFactory** is a reference to the factory associated with the TC/SS.7 address. The **out** parameter **resp_iface** is the IDL scoped name of the responder interface created by the factory. The **out** parameter **init_iface** is the IDL scoped name of the initiator interface corresponding to the interface created by the factory. In the case of a Type I definition (symmetric initiator and responder interfaces) the values of both **resp_iface** and **init_iface** are the same.

2.2.2.5 The GwAdmin Interface

Every TC/CORBA gateway will have a singleton **GwAdmin** object that provides various helper operations for CORBA based TC-Users. A **GwAdmin** object will always be identified by a well-publicized name in the CORBA Naming Service. The interface is defined below.

```
interface GwAdmin{

readonly attribute TcFactoryFinder tc_user_factory_naming_if;

readonly attribute TcPduProviderFactory tc_pdu_provider_factory_if;

}; // end GwAdmin
```

This interface allows TC-User objects to obtain the object reference to the **TcFactoryFinder** object or the **TcPduProvider** interface for use by protocol-aware CORBA objects in interacting with the SS7 stack. This will usually only be used by Proxy objects. It is not necessary to support this attribute in a conformant implementation as the proxy objects may use directly a proprietary API for accessing

the SS7/TC protocol stack. The **GwAdmin** will return a null reference for the **tc_pdu_provider_factory_if** attribute if the gateway does not support the TC PDU-oriented interfaces.

2.2.2.6 The TcServiceFinder Interface

Every TC/CORBA domain will have at least one **TcServiceFinder** object that provides various helper operations to find all the TC-User CORBA services. A **TcServiceFinder** object will always be identified by a well-publicized name in the CORBA Naming Service. The interface is shown below.

```
interface TcServiceFinder{
    readonly attribute CosNaming::NamingContext gt_root;
    readonly attribute GwAdmin gw_admin_if;
    readonly attribute TcRepository tc_repository_if;
}; //end TcServiceFinder
```

readonly attribute CosNaming::NamingContext gt_root;

The attribute **gt_root** returns a reference to the root TC/SS.7 Cos Naming Service naming context for the local domain.

readonly attribute GwAdmin gw_admin_if;

The attribute **gw_admin_if** returns a reference to the local TC/CORBA gateway's **GwAdmin** interface.

readonly attribute TcRepository tc_repository_if;

The attribute **gw_admin_if** returns a reference to the local TC/CORBA **TcRepository**.

2.2.3 Application Location and Association Initiation

Due to the extended capabilities of signalling application context provided by the 1993 specifications of TC (hereafter called TC93) over those of the earlier TC specification (hereafter called TC88) it is necessary to define a two-tier mechanism for application location and association initiation. The base functionality is used in the TC88 case and this is extended to allow the richer TC93 interactions.

2.2.3.1 Base Functionality (TC88)

Application Location

All TC-User Application Entities can be located by their Global Title. This is used as a naming context in the CORBA Naming Service. In order to facilitate easy location, there is a well defined name, called GT, used as the root of the global GT tree. This is defined within the scope of some naming-context in the CORBA Naming Service. The location of this naming context is published in the **TcServiceFinder** interface.

An individual Global Title is placed into the naming tree below the well-defined naming context GT. It is placed into the naming service as a naming context. If it is desired to support multiple TC-User applications on one Global Title, then an SS.7 SSN may be used to distinguish between them. To support this functionality use the same naming tree as for TC93 and use the SSN value as the name of an application context node.

Underneath a particular GT there is a naming context named **DefAc**. Beneath this is located an object name node called "**TcUserFactory**" for the default factory associated with that TC Application Entity. There are also two naming contexts named:

- "**RESP:<responder_interface_name>**" - where **<responder_interface_name>** is the **ScopedName** of the interface supported by the objects created by the factory
- "**INIT:<initiator_interface_name>**" - where **<initiator_interface_name>** is the **ScopedName** of the corresponding interface to the objects created by the factory.

In the case of a symmetric interface, both of these nodes will have the same name.

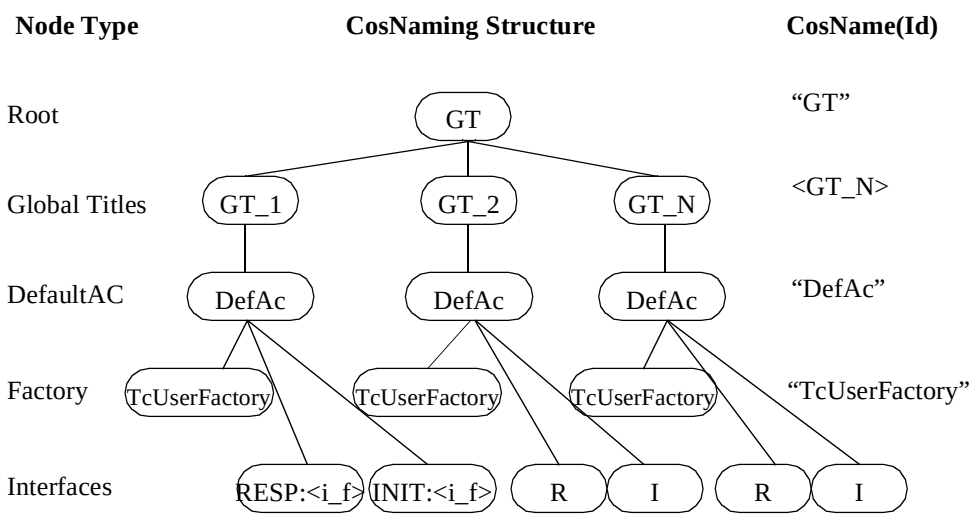


Figure 2-1 The Global Title Naming Tree in the COS Naming Service for TC88

Table 2-1 TC88 Global Title Descriptions

Node	IDL Type	Name - Id	Name - Kind
Root GT	Naming Context	"GT"	<null>
Particular GT	Naming Context	<specific GT>	<null>
Default Appl. Context	Naming Context	DefAc	<null>
TcUser factory	Named Object	"TcUserFactory"	<null>
Initiator iface name	Naming Context	"INIT:<iface_name>"	<null>
Responder iface name	Naming Context	"RESP:<iface_name>"	<null>

By setting up the naming tree like this, any CORBA object (including proxy objects at a gateway) can find a factory associated with a GT and the name of the interface supported by that factory. In the TC88/Type I description case this is also the interface that must be supported by the initiating server object (or a server object associated with an initiating CORBA client object) as the interfaces involved in the association are symmetrical.

Association Initiation

Initiating an association may be done by invoking a:

- **create_<interface_name>_responder** operation on the **TcUserFactory**
- **create_<interface_name>_responder_with_dialog_data** operation on the **TcUserFactory**
- **new_association** operation on a TC-User object of the correct type already created.
- **new_association_with_dialogdata** operation on a TC-User object of the correct type already created.

When a new association is started, both responder and initiator exchange **AssociationIds**. If the **TcContextSetting** selected for the association supports **AssociationIds**, both must place the **AssociationId** received during association initiation in the **TcContext** parameter of all outgoing messages (operations, results, exceptions) within that association.

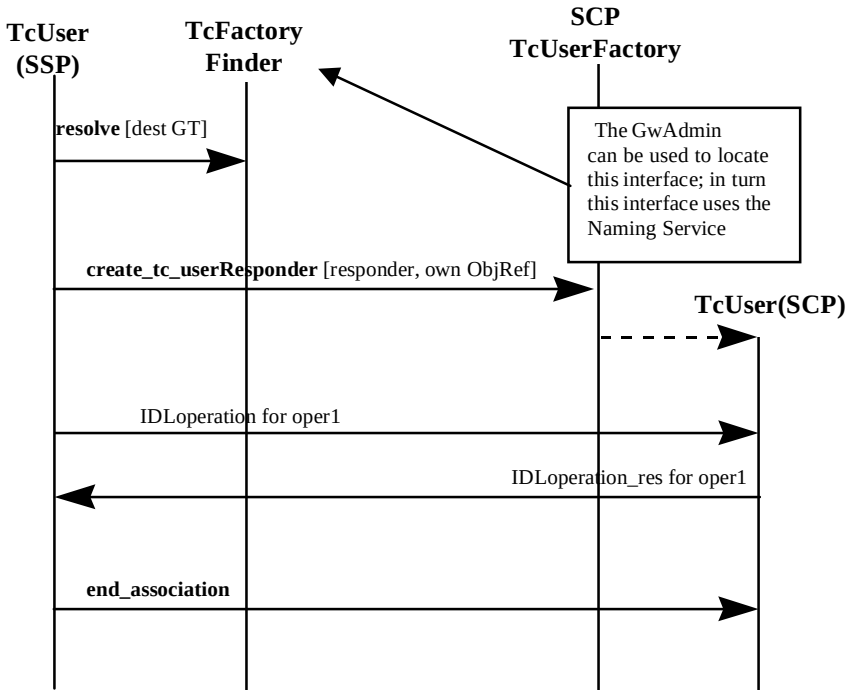


Figure 2-2 Application Location and association initiation by a TC88-User (no dialog Portion)

The process of initiating an association is shown in the Message Sequence Chart (MSC) in Figure 2-2. This MSC considers the case of association initiation and operation invocation in the CORBA domain. Furthermore, to reduce complexity, we have considered the use of TC without the use of Application Context (i.e., the dialog Portion). The steps, briefly, are as follows.

1. The **TcUser**(SSP) can use the **GwAdmin** interface (not shown in the MSC) to obtain the reference to an **TcFactoryFinder** object, on which it invokes a **resolve** operation to obtain the object references of factory interfaces bound to the GT passed as a parameter in the invocation. The **TcFactoryFinder** interface makes use of the CORBA Naming Service (not shown in the MSC) to perform the name resolution. (It is assumed that the name bindings have been performed at some earlier time).

2. The **TcUserFactory** reference is returned by the address resolution is used to create an instance of the target CORBA object, the SCP.

3. The SSP invokes the desired IDL equivalent of a specific INAP operation on the SCP. (The MSC for converting the identifier of a received TC/ROS operation to its IDL scoped name is shown in Figure 2-4 on page 2-34. In Figure 2-2, those steps are assumed to have been performed.)

4. The result is returned to the SSP which then ends the association with an **end_association** operation.

2.2.3.2 Extensions for TC93

TC93 adds the possibility of sending a dialog PDU within a BEGIN PDU. The dialog PDU can contain the following pieces of information of relevance to association setup:

- Protocol Version number
- Application Context
- User Data

The exact use of user-data is TC-User object and Factory dependent. The Naming Service structure can be extended by adding additional, specific, Application Context nodes below a Global Title node as shown in Figure 2-3. This allows both TC88 and TC93 communication with CORBA implementations of entities using the same Global Title.

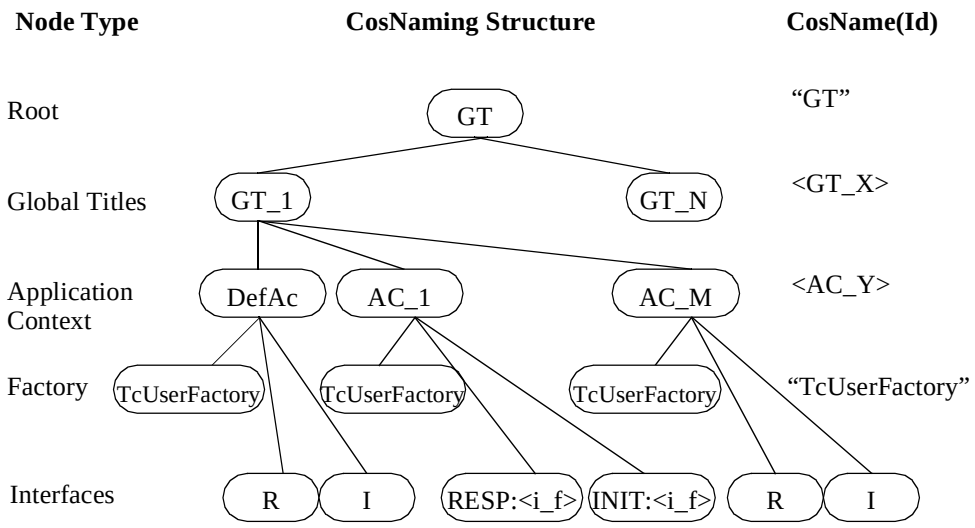


Figure 2-3 The Global Title Naming Tree in the COS Naming Service for TC93

Table 2-2 TC93 Global Title Descriptions

Node	IDL Type	Name - Id	Name - Kind
Root GT	Naming Context	"GT"	<null>
Particular GT	Naming Context	<specific GT>	<null>
TC-User object factory	Named Object	"TcUserFactory"	<null>
Application Context	Naming Context	<AC name>	<null>
Initiator iface name	Naming Context	INIT:<iface_name>	<null>
Responder iface name	Naming Context	RESP:<iface_name>	<null>

As protocol version does not change frequently, instead of adding another tier of nodes in the Naming Service, it is passed as a creation parameter in the **create_with_dialogdata** operation. User data may be passed as a creation parameter in the **create_with_dialogdata** operation.

The process for association setup is as before, with the difference that the responder and initiator interfaces may now be asymmetric.

2.2.4 Association Maintenance

TC-user applications communicate with each other through associations in which operations may be invoked in either direction. The application location and association initiation procedure ensures that the two CORBA objects communicating have an object reference to one another.

The **TcUser** interface provides two levels of association control:

- *standard*: to be supported by all TC-User objects. This is defined by the operations on the **TcUser** interface. This allows an object to notify the other that the association has aborted or request the end of an association.
- *full control*: may be supported by TC-User proxy objects residing at a gateway or CORBA objects that wish to use the **DialogFlowCtr** parameter (part of **TcContext** and **TcLinkedContext**) to optimize dialog support. This allows explicit use of TC dialog flow control; so a proxy object can minimize the traffic generated on the SS.7 network and a normal CORBA object can end associations without needing to invoke the **new_association** and **end_association** operations.

In addition to the control of an association, a multi-association TC-User object maintains information on current associations by allocating a locally unique **AssociationId** when starting an association. It also receives a unique **AssociationId** from the other TC-User CORBA objects in the association (or its factory). All TC-User CORBA objects must place the **AssociationId** received in the **TcContext** parameter of all operations, results, and exceptions within an association. Similarly all operations, results, and exceptions that it receives during an association will have the **AssociationId** it allocated at association initiation.

2.2.5 Operation Invocation

Given the ST already specified, it is possible for CORBA-based TC-user applications to interact via associations. If operation invocations and returns are to be translated into the non-CORBA domain, the information lost from the ASN.1 during the translation process must be replaced.

To support this activity the TC Repository has been specified. This is similar to the SMI OID Repository specified in the TMN/CORBA Interworking specification. The TC Repository provides a way to find the operation or error name (as stored in the IR) given the operation or error code and the interface name.

The following MSC illustrates the use of the TC Repository when translating operations between CORBA and TC/ROS. Various implementation scenarios could make different use of the TC and Interface Repositories; for instance, a gateway implemented entirely as CORBA static stubs could have the information in the TC Repository and the IR “hard-coded” into the code associated with the stub; however, a fully dynamic gateway will need these services.

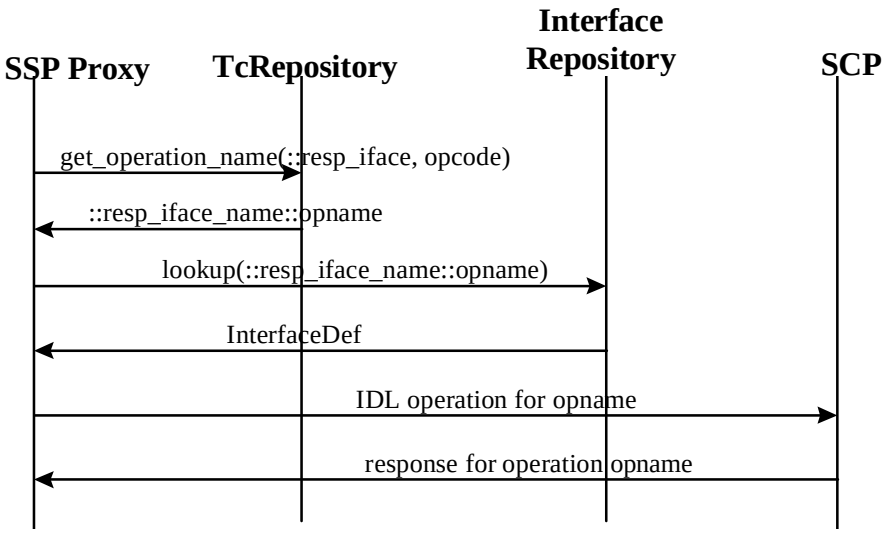


Figure 2-4 Using the TcRepository

2.2.5.1 The TcRepository

This interface standardizes access to the information generated during Specification Translation to store ASN.1 identifier to IDL scoped name mappings. It is also a place where operation timer and priority values may be stored.

The interface to access the TC repository is as follows:

```
enum IdType {LOCAL_ID, OID};
union IdValue switch(IdType) {
    case LOCAL_ID: ASN1_Integer local_id;
    case OID: ASN1_ObjectIdentifier oid;
};

interface TcRepository {

        ScopedName get_error_name(in ScopedName iface,
                in IdValue code );

        ScopedName get_operation_name(in ScopedName iface,
                in IdValue code );

        ScopedName get_extension_name(in ScopedName iface,
                in IdValue code );

        IdValue get_id(in ScopedName scoped_name);

    ASN1_ExtensionCriticality get_extension_criticality(
                in ScopedName extension_scoped_name);
```

```
unsigned long get_operation_timer(in ScopedName
                    op_scoped_name );

}; //end TcRepository

enum IdType {LOCAL_ID, OID};
```

IdType is used to discriminate between an ASN.1 locally assigned identifier and an ASN.1 object identifier. These are the two ways an ASN.1 macro may be assigned an identifier in an ASN.1 module.

```
union IdValue switch(IdType) {
    case LOCAL_ID: ASN1_Integer local_id;
    case OID: ASN1_ObjectIdentifier oid;
};
```

IdValue holds either an object identifier (OID) or local identifier value.

```
ScopedName get_error_name(in ScopedName iface,
                in IdValue code );
```

The **get_error_name** operation returns the **ScopedName** in the IR of the exception mapped to a particular ASN.1 Error during ST. The parameter **iface** takes the scoped name of the interface in the IR to which the exception belongs. The **IdValue** takes the ASN.1 local identifier or OID of the Error.

```
ScopedName get_operation_name(in ScopedName iface,
            in IdValue code );
```

The **get_operation_name** operation returns the **ScopedName** in the IR of the IDL operation mapped to a particular ASN.1 operation during ST. The parameter **iface** takes the scoped name of the interface in the IR to which the operation belongs. The **IdValue** takes the ASN.1 local identifier or OID of the operation.

```
IdValue get_id(in ScopedName scoped_name);
```

The **get_id** operation returns the ASN.1 identifier of the ASN.1 Type mapped to IDL whose scoped name in the IR is passed as an **in** parameter.

```
ASN1_ExtensionCriticality get_extension_criticality(in ScopedName
extension_scoped_name);
```

The **get_extension_criticality** operation returns the **ASN1_ExtensionCriticality** value of the IDL type (a mapped ASN.1 extension type) whose scoped name in the IR is passed as an **in** parameter.

```
unsigned long get_operation_timer(in ScopedName
        op_scoped_name );
```

The **get_operation_timer** operation returns the timer value in seconds of the operation whose scoped name in the IR has been passed as an **in** parameter.

2.2.5.2 Mapping of ROS/TC Rejects to CORBA System Exceptions

Both ROS and TC provide an exception reporting capability through the use of the Reject Application Protocol Data Unit (APDU). The Reject type provides an APDU for reporting erroneous use of the other ROS/TC APDUs or to inform a peer ROS/TC-user that a problem has been detected with an APDU that was previously received.

The Reject APDU carries a problem code parameter that describes the nature of the error that has occurred. The problem code may indicate one of four categories of problems: **`GeneralProblem`**, **`InvokeProblem`**, **`ReturnResultProblem`**, or **`ReturnErrorProblem`**. As the **`GeneralProblem`** or **`InvokeProblem`** Reject problem codes are implicitly available to all ROS/TC operations, they correspond to standard CORBA exceptions that are implicitly included in all CORBA operation signatures. However, a **`GeneralProblem`**, **`ReturnResultProblem`**, or **`ReturnErrorProblem`** generated while processing an operation result or exception has no corresponding CORBA procedure.

Each standard CORBA exception includes a minor code to designate the subcategory of the exception.

To uniquely identify ROS/TC exceptions in CORBA, new minor field values of the system exceptions must be defined. Here only the lower order bits of the minor field are specified in Table 2-3 on page 2-41. The higher order bits (VMCID) are administratively assigned by the OMG. The OMG list of assigned VMCIDS is located at: http://www.omg.org/cgi-bin/doc?tags

---

Note – The lower order bit values do not interfere with the lower order bit values of the exception in the Objects by Value specification [**18**]. The high order bits (VMCID) can be either the same as in the Objects by Value submission, or a new VMCID for domain-specific purposes can be assigned. This decision should be made during the post-processing of the specification.

The existing CORBA system exceptions may then be used to uniquely represent ROS/TC Reject problem codes generated while processing a ROS/TC operation.

Most Reject problem codes represent protocol errors whereas the standard CORBA exceptions are at the application level. Consequently, not all Reject problem codes can be mapped directly to standard CORBA exceptions and an alternative solution is required. The solution is as follows:

- If a native ROS/TC-user server application generates a **`Reject`** during the processing of an operation, then in the CORBA domain, a CORBA standard exception, identified by a distinct minor code, is thrown. The type of this exception is explained in “Mapping of TC/ROS General Problem” on page 2-37 and “Mapping of TC/ROS Invoke Problem” on page 2-38.
- If a native TC-user server application generates a **`Reject`** during the processing of a CORBA operation result or exception, then the **`abort_association`** operation on the **`TcUser`** interface is invoked with appropriate parameters. This is described in “Handling of TC/ROS ReturnResult Problem” on page 2-39 and “Handling of TC/ROS ReturnError Problem” on page 2-39.

- If a CORBA system exception is raised by a CORBA-based server, then the processing of such an event is explained in “Processing of System Exceptions raised by CORBA-based servers” on page 2-41.

Mapping of TC/ROS General Problem

In TC/ROS, **`GeneralProblems`** apply to the TC component sub-layer in general, and have no specific relationship to a particular component type such as an **`Invoke`**. There are three General Problems: **`unrecognizedPDU`**, **`mistypedPDU`**, and **`badlyStructuredPDU`**. These may occur within the context of an operation (CORBA exception generated) or while processing an exception or result (uses the **abort_association** operation on the **TcUser** interface). These are mapped as follows:

unrecognizedPDU Problem

The **`unrecognizedPDU`** problem is received if the component type tag is not recognized as one of **`ROIV`**, **`RORS`**, **`ROER`**, or **`RORJ`**. If this occurs within the context of an operation, it is mapped to the CORBA exception DATA_CONVERSION, with minor code value as given in Table 2-3 on page 2-41 and with a completion status of **COMPLETED_NO**. If this occurs outside the scope of an operation invocation, this problem cannot be ignored due to its general impact. The solution proposed here is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

mistypedPDU Problem

The **`mistypedAPDU`** problem is received if the structure of a component does not conform to that described in the ROS/TC specifications. If this occurs within the context of an operation, it is mapped to the CORBA exception DATA_CONVERSION, with minor code value as given in Table 2-3 on page 2-41 and with a completion status of **COMPLETED_NO**. If this occurs outside the scope of an operation invocation, this problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

badlyStructuredPDU Problem

The **`badlyStructuredAPDU`** problem is received if the contents of a component do not conform to the encoding rules defined for this APDU, as described in the TC/ROS specifications. If this occurs within the context of an operation, it is mapped to the CORBA exception MARSHAL, with minor code value as given in Table 2-3 on page 2-41 and with a completion status of **COMPLETED_NO**. If this occurs outside the scope of an operation invocation, this problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

Mapping of TC/ROS Invoke Problem

`Invoke Problems` are generated by the TC component sub-layer or the ROS/TC-user and relate only to the **`Invoke`** APDU. There are eight Invoke Problems: **`duplicateInvocation`**, **`unrecognizedOperation`**, **`mistypedArgument`**, **`resourceLimitation`**, **`initiatorReleasing`**, **`unrecognizedLinkedID`**, **`linkedResponseUnexpected`**, and **`unexpectedChildOperation`**. These are mapped as described in the following sub-sections.

duplicateInvocation Problem

The **`duplicateInvocation`** problem is received if the **`InvokeID`** is one for a previously invoked operation for which a response has not been received. It is mapped to the CORBA exception BAD_INV_ORDER, with minor code value as given in Table 2-3 on page 2-41 and with a completion status of **COMPLETED_NO**.

unrecognizedOperation Problem

The **`unrecognizedOperation`** problem is received if the operation code was unknown or not agreed between the two ROS/TC-Users. It is mapped to the CORBA exception BAD_OPERATION with a minor code value as given in Table 2-3 on page 2-41 and a completion status of **COMPLETED_NO**.

mistypedArgument Problem

The **`mistypedArgument`** problem is received if the type of parameter in an **`Invoke`** component is unknown or not that agreed to between the two ROS/TC-Users for that operation. It is mapped to the CORBA exception BAD_PARAM, with minor code value as given in Table 2-3 on page 2-41 and a completion status of **COMPLETED_NO**.

resourceLimitation Problem

The **`resourceLimitation`** problem is received if sufficient resources to perform the requested operation are not available. It is mapped to the CORBA exception NO_RESOURCES with minor code value as given in Table 2-3 on page 2-41 and a completion status of **COMPLETED_MAYBE**.

initiatorReleasing Problem

The **`initiatorReleasing`** problem is received if the requested operation cannot be invoked due to soonest release of the dialog. It is mapped to exception COMM_FAILURE, with minor code value as given in Table 2-3 on page 2-41 and with a completion status of **COMPLETED_NO**.

unrecognizedLinkedID Problem

The **`unrecognizedLinkedID`** problem is received if the given **`LinkedID`** does not relate to any active operation. It is mapped to the CORBA exception BAD_INV_ORDER with minor code value as given in Tabl e2-3 on pa ge2-41 and a completion status of **COMPLETED_NO**.

linkedResponseUnexpected Problem

The **linkedResponseUnexpected** problem is received if linked invocations are not allowed for the operation referred to by the given **LinkedID**. It is mapped to the CORBA exception NO_PERMISSION with minor code value as given in Tabl e2-3 on page 2-41 and with a completion status of **COMPLETED_NO**.

unexpectedChildOperation Problem

The **unexpectedChildOperation** problem is received if the operation referred to by the **linkedID** does not allow this linked operation. It is mapped to exception NO_PERMISSION with minor code value as given in Table 2-3 on page 2-41 and a completion status of **COMPLETED_NO**.

Handling of TC/ROS ReturnResult Problem

ReturnResult problems are generated by the TC component sub-layer or by the ROS/TC-User and relate only to erroneous use of the **ReturnResult** component type.

unrecognizedInvocationResult Problem

The **unrecognizedInvocationResult** problem is generated if no operation with the specified **InvokeID** was previously invoked by the peer ROS/TC-User. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

resultResponseUnexpected Problem

The **resultResponseUnexpected** problem is received if the previously invoked operation has not returned a response regarding success. This could only happen if the ASN.1 to IDL language mapping was not performed correctly and implies a general fault. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

mistypedResult Problem

The **mistypedResult** problem is received if a parameter in the **ReturnResult** component was wrong or not agreed between the two TC-Users. This could only happen if the ASN.1 to IDL language mapping was not performed correctly and implies a general fault. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

Handling of TC/ROS ReturnError Problem

ReturnError problems are generated by the TC component sub-layer or by the ROS/TC-User and relate only to the Return Error component type.

unrecognizedInvocation Problem

The **`unrecognizedInvocation`** problem is received if no operation with the specified **`InvokeID`** was previously invoked by the remote ROS/TC-User. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

errorResponseUnexpected Problem

The **`errorResponseUnexpected`** problem is received if the previously invoked operation was not defined to return a response to report an error. This could only happen if the ASN.1 to IDL language mapping was not performed correctly and implies a general fault. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

unrecognizedError Problem

The **`unrecognizedError`** problem is received if the error received is not one among the list of errors for all possible operations that are defined for the interaction between the two ROS/TC-Users. This could only happen if the ASN.1 to IDL language mapping was not performed correctly and implies a general fault. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

unexpectedError Problem

The **`unexpectedError`** problem is received if the error received is not one among the list of errors defined for the operation identified by the **`InvokeID`**. This could only happen if the ASN.1 to IDL language mapping was not performed correctly and implies a general fault. This problem cannot be ignored due to its general impact. The solution proposed here is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

mistypedParameter Problem

The **`mistypedParameter`** problem is received if the parameter in the **`ReturnError`** APDU is wrong or not that agreed between the two TC-Users. This could only happen if the ASN.1 to IDL language mapping was not performed correctly and implies a general fault. This problem cannot be ignored due to its general impact. The solution is to abort the ROS Association or the TC dialog and report the error through the TCUser base interface as described in Section 1.2, “Architectural Overview,” on page 1-3.

Processing of System Exceptions raised by CORBA-based servers

In the case where a CORBA system exception, not related to a ROS/TC Reject error, is thrown by the ORB or Object Adapter after an Invoke has been sent to a CORBA-based server Application Entity from a ROS/TC-based application, the association/dialog must be aborted as there is no way of mapping these exceptions to the fine-grained Reject problems in the non-CORBA domain. If the invocation is delivered successfully, the CORBA-based Application Entity may choose to raise a TC-specific exception based on the contents of the invocation received. Such exceptions are then mapped by the TC-User Proxy object to a particular InvokeProblem.

Table 2-3 Reject Problems and System Exceptions

Reject Problem	Action	System Exception	Minor Code
General Problem[1]			
unrecognizedPDU	user defined	DATA_CONVERSION	100
mistypedPDU	user defined	DATA_CONVERSION	101
badlyStructuredPDU	user defined	MARSHAL	100
Invoke Problem			
duplicateInvocation	user defined	BAD_INV_ORDER	100
unrecognizedOperation	user defined	BAD_OPERATION	100
mistypedArgument	user defined	BAD_PARAM	100
resourceLimitation	user defined	NO_RESOURCES	100
initiatorReleasing	user defined	COMM_FAILURE	100
unrecognizedLinkedId	user defined	BAD_INV_ORDER	101
linkedResponseUnexpected	user defined	NO_PERMISSION	100
unexpectedLinkedOperation	user defined	NO_PERMISSION	101
ReturnResultProblem			
unrecognizedInvocation	abort dialog	NA	NA
resultResponseUnexpected	abort dialog	NA	NA
mistypedResult	abort dialog	NA	NA
ReturnErrorProblem			
unrecognizedInvocation	abort dialog	NA	NA
errorResponseUnexpected	abort dialog	NA	NA
unrecognizedError	abort dialog	NA	NA
unexpectedError	abort dialog	NA	NA
mistypedParameter	abort dialog	NA	NA

1. GeneralProblems that do not occur as a result of an operation invoke are instead mapped to an **abort_association** operation on **TcUser**.

2.2.6 Asynchronous ROS/TC Operation Invocations

In ROS, the **OPERATION** information object class contains a field that permits the definition of whether an operation is synchronous or not (i.e., if defined as synchronous, another synchronous operation may not be invoked until the current operation instance has returned). TC specifies that all operations are invoked asynchronously. CORBA currently provides two modes of invocation:

1. *synchronous*: the client program or thread blocks when a remote invocation is made and waits until the result arrives.
2. *deferred synchronous*: the client thread continues processing, subsequently polling to see if results are available.

Currently, the deferred synchronous model is only available when using the Dynamic Invocation Interface (DII). As ROS and TC applications will require a truly asynchronous method invocation model for use with both static and dynamic stubs, the current CORBA specification alone will not provide the infrastructure required for full TC/CORBA Interworking. This specification proposes using the facilities provided by the CORBA Messaging Service to allow ROS/TC users in the CORBA domain access to a truly asynchronous invocation model.

2.2.7 Quality of Service in ROS/TC

ROS and TC permit the application designer to quote a priority for sending an invocation/result to aid the infrastructure in choosing the order of sending for the case where there are several invocations/results waiting to be sent. In TC, designers may provide, as ASN.1 comments, a timer value associated with an operation. This indicates a “time to respond” to an operation invocation, failing which the client assumes the operation failed and proceeds to the next task. The **TcRepository** defined in this specification may be used to store ROS/TC operation timer values. These may be accessed by a CORBA object to find an appropriate timer value to use with the CORBA Messaging Service.

TC PDU-oriented Interfaces 3

Contents

This chapter contains the following sections.

3.1 Introduction

The TC PDU-oriented interfaces are designed to standardize access by TC protocol aware CORBA objects (such as proxy objects at a gateway) to a TC/SS.7 protocol stack. This allows implementation of TC-aware applications that are independent of a particular stack vendor. It is not necessary to use the TC PDU-oriented interfaces to implement a TC/CORBA gateway, as a custom mapping to an individual SS7 stack may be part of the implementation of the proxy interfaces generated during Specification Translation.

These interfaces represent a very low level mapping that requires users to be aware of the details of the TC primitive interface as defined in ITU-T Rec. Q.771 [**16**]. It is also unusual in that it requires TC-protocol aware objects to be able to encode and decode native ASN.1/BER data based on ITU-T Rec. Q.773 [**16**].

3.2 TC PDU-oriented Interfaces Framework

The TC PDU-oriented interfaces provide a CORBA environment for the use of TC/SS.7 (not TC-User) transport and messaging services. This is done in a TC PDU-oriented fashion. The facilities exposed are:

- TC dialog handling by the TC/SS.7 stack and TC-aware CORBA objects
- TC component handling by the TC/SS.7 stack and TC-aware CORBA objects
- Initiation of TC sessions by TC-aware CORBA objects
- Registration/Deregistration of TC-aware CORBA objects to receive TC dialogs
- Setting of per-dialog parameters such as QoS

There are four interfaces defined here:

1. **TcPduProvider:** is supported by the TC/SS.7 stack.
2. **TcPduProviderFactory:** also supported by the TC/SS.7 stack.
3. **TcPduUser:** must be supported by a CORBA object wishing to be a user of the TC/SSS.7 stack.
4. **TcPduUserFactory:** which must also be supported by a CORBA object wishing to be a user of the TC/SSS.7 stack.

Access to these interfaces is through the **TcSignaling::GwAdmin** interface. These interfaces will not be used in pure CORBA (non-gateway) interactions.

All communication from a CORBA object to a TC/SS.7 protocol stack will be through a **TcPduProvider**. These **TcPduProvider** objects (each of which may have multiple dialogs) may be created at a gateway through the invocation of methods on a **TcPduProviderFactory** object. There is only one **TcPduProviderFactory** at a gateway.

All communication from a TC/SS.7 protocol stack to a CORBA object will be through a **TcPduUser**. In the case of a dialog initiated by a CORBA object, a **TcPduUser** object is defined at the start of the TC dialog. In the case of a dialog initiated by a TC/SS.7 protocol stack upon receipt of a TC message from the non-CORBA domain, the **TcPduUserFactory** that has been registered for a particular TC address (Global Title and optionally an Application Context) is used to create the appropriate **TcPduUser**.

Both the **TcPduProvider** and **TcPduUser** interfaces support TC primitive operations.

Each TC component is represented by IDL parameters, that include any header information for the component (such as invocation and linked IDs) and an **Asn1Data** type to carry any ASN.1 (e.g., operation code and arguments encoded using Basic Encoding Rules (BER)) associated with the component. The **DialogPortion** of TC PDUs is also represented as an **Asn1Data** Type.

3.3 Interface Definitions

3.3.1 Common Data Types for the TC PDU-oriented Interfaces

```
typedef unsigned short ProblemType;
typedef unsigned short ProblemCode;
```

These two values are used to carry the TC/ROSE error codes for rejected components. All of the standard values are available through **const** declarations of these types.

```
typedef unsigned long Timeout;
```

Timeout is used to specify the timer value in seconds to indicate the time taken for an operation to complete.

```
typedef unsigned short OperationClass;
```

OperationClass is used to specify the TC operation class for TC Invoke components. Standard values for **OperationClass** are defined as **const** declarations.

```
struct RejectProblem{
        ProblemType type;
        ProblemCode code;
};
```

A **RejectProblem** is used to carry TC Reject component information identifying via **ProblemType** the component that is being rejected and using **ProblemCode** to specify the specific problem. Standard values for the TC/ROS reject causes are defined as **const** declarations.

```
enum TerminationType{
        PREARRANGED,
        BASIC
};
```

TerminationType is used to specify in the **TcPduUser** or **TcPduProvider end** operations whether the termination requires the explicit reception/sending of an **END** PDU.

```
union DialogPortion switch(Boolean) {
        case TRUE : ApplicationContext a_c;
        case FALSE : Asn1Data dialog_info;
};
```

DialogPortion is used to carry **TC DialogPortion** information in **TcPduUser** or **TcPduProvider** operations. The **Asn1Data** field may be used to carry the full **DialogPortion** PDU or **ApplicationContext** may be used to carry just the Application Context to be encoded in the **DialogPortion** PDU.

typedef unsigned long DialogId;

DialogId is used to hold the TC transaction ID identifying a particular transaction between to TC-Users.

typedef short DialogQos;

DialogQos is used to carry the TC Quality of Service for a dialog. It identifies which SCCP Transport Class should be used for the PDU and whether to return the PDU in case it cannot be delivered. The standard values of **DialogQos** are defined as **const** declarations. All non-negative values are reserved for use in OMG specifications. Any negative value of **DialogQos** is considered a vendor extension.

Asn1Data sequence<octet>;

Asn1Data is used to carry an indefinite amount of BER-encoded ASN.1 data. The specific structure of the **Asn1Data** octets depend on the context in which it is used. See the specific type or operation in which **Asn1Data** is used for more information.

typedef short PAbortReason;

PAbortReason is used to carry the TC P-Abort Reason code. The standard values of **PAbortReason** are defined as **const** declarations. All non-negative values are reserved for use in OMG specifications. Any negative value of **PAbortReason** is considered a vendor extension.

```
enum ComponentType{
        INVOKE,
        RESULT_L,
        RESULT_NL,
        ERROR,
        U_REJECT,
        R_REJECT
};
```

ComponentType defines the set of TC components that may be passed in a **ComponentList**.

```
struct Invoke(
OperationClass op_class;
InvokeId ivk_id;
InvokeId lnk_id;
Asn1Data oper;
Timeout op_timer;
);
```

Invoke is used to carry a TC Invoke component and corresponds to the TC-INVOKE primitive. The **OperationClass** must be specified. The Invoke ID **ivk_id** must be specified. All operations within a dialog must have unique invoke IDs. If the operation is linked to a previous operation, then the Linked ID **lnk_id** must be set equal to the

ivk_id of the operation linked to. Otherwise **lnk_id** is set to **TcSignaling::NO_ID**. **Asn1Data** is used to encode the TC operation code and TC operation parameters. If the **Asn1Data** carries BER encoded data, it must have the following format:

```
Operation code tag
Operation code length
Operation code value
Parameter tag
Parameter length
Parameters
```

The **op_timer** value may be used to specify the timeout (in seconds) for the operation or 0 to use the default timer for the dialog when sending the component. A received component always has an **op_timer** value of 0.

```
struct ResultL(
InvokeId ivk_id;
Asn1Data result;
);
```

ResultL is used to carry the result of a TC operation and corresponds to the TC-RESULT-L primitive. The member **ivk_id** must carry the invoke ID of the original operation. **Asn1Data** is used to carry the actual result information. If the **Asn1Data** carries BER encoded data, it must have the following format:

```
Sequence tag
Sequence length
Operation code tag
Operation code length
Operation code value
Parameter tag
Parameter length
Parameters
```

```
struct ResultNl(
InvokeId ivk_id;
Asn1Data result;
);
```

ResultNl is used to carry a segment of a result of a TC operation and corresponds to the TC-RESULT-NL primitive. The member **ivk_id** must carry the invoke ID of the original operation. **Asn1Data** is used to carry the actual result information. If the **Asn1Data** carries BER encoded data, it must have the following format:

```
Sequence tag
Sequence length
Operation code tag
Operation code length
Operation code value
Parameter tag
Parameter length
```

```
Parameters

struct UError(
InvokeId ivk_id;
Asn1Data error;
);
```

Error is used to carry an error associated with a TC operation and corresponds to the TC-ERROR primitive. The member **ivk_id** must carry the invoke ID of the original operation. **Asn1Data** is used to carry the actual result information. If the **Asn1Data** carries BER encoded data, it must have the following format:

```
Error code tag
Error code length
Error code value
Parameter tag
Parameter length
Parameters

struct UReject(
InvokeId ivk_id;
RejectProblem problem;
);
```

UReject is used to carry a TC/ROSE reject cause associated with a TC component that is being rejected by the TC-User, and corresponds to the TC-U-REJECT primitive. The member **ivk_id** must carry the invoke ID of the rejected component.

```
struct RReject(
InvokeId ivk_id;
RejectProblem problem;
);
```

RReject is used to carry a TC/ROSE provider reject associated with a TC component and corresponds to the TC-R-REJECT primitive. The member **ivk_id** must carry the invoke ID of the rejected component.

```
union Component switch(ComponentType) {
case INVOKE : Invoke i;
case RESULT_L : ResultL r_l;
case RESULT_NL : ResultNl r_nl;
case U_ERROR : UError u_e;
case U_REJECT : UReject u_r;
case R_REJECT : RReject r_r;
};
```

Component defines the group of allowed **ComponentTypes**.

```
typedef sequence<Component> ComponentList;
```

A **ComponentList** is a sequence of Components.

3.3.2 The TcPduProviderFactory Interface

The **TcPduProviderFactory** interface is used to create new TC sessions for CORBA objects and to register and de-register factory objects that may create objects for receiving dialogs from TC/SS.7 entities.

```
TcPduProvider create_tc_pdu_provider(
        in TcPduUser user,
        out DialogId d_id)
    raises(NoMoreDialogs);
```

The first operation, **create_tc_pdu_provider**, allows a CORBA object to start a TC session by returning a reference to a **TcPduProvider** interface. This is initialized by passing a single **in** parameter of a reference to a **TcPduUser** interface associated with the invoking CORBA object. This is where all the asynchronous replies to the TC dialogs initiated on the **TcPduProvider** will be sent. A single **out** parameter, **d_id**, returns the first dialog ID to be used in a TC dialog. The NoMoreDialogs exception is raised if the **TcPduProviderFactory** is unable to create any more **TcPduProvider** objects at this time.

```
void register(in TcAddress dest,
        in ApplicationContext a_c,
        in TcPduUserFactory user_factory)
    raises(AlreadyBound);
```

The second operation, register, allows a CORBA object to register a specific **TcPduUserFactory** object as the factory object which creates **TcPduUser** objects associated with a particular destination TC/SS.7 address. The address is formed from the tuple (Global Title, Application Context Name). The Application Context Name may be null to indicate that all calls to the Global Title without Application Context information are to be sent to objects created by the registering factory. The Application Context Name may contain the string "_ALL_CONTEXTS_" to indicate that all calls to the Global Title are to be sent to objects created by the registering factory object. As only one destination factory object may register for each address, an exception of AlreadyBound is provided to signal that the specified address already has a factory registered.

```
void deregister(in TcAddress dest,
        in ApplicationContext a_c)
    raises(UnknownAddress);
```

Finally a **deregister** operation is provided so that factory objects may indicate that they no longer wish to create objects associated with the destination address specified in the deregister parameters. This has no effect on dialogs already in progress, it merely disallows any new dialogs directing creation requests to the deregistering object. The meaning of the contents of the Application Context Name parameter is interpreted as for the **register** operation. If the address specified in the operation parameters does not match any already registered, then the UnknownAddress exception is thrown.

3.3.3 The TcPduProvider Interface

This interface is used to communicate with the TC/SS.7 stack. It provides TC dialog handling primitives that may carry TC components if desired. There are also some component oriented primitives supported and operations for getting and setting the dialog Qos.

```
DialogId get_dialog_id(in TcPduUser user)
    raises (NoMoreDialogs);
```

The operation **get_dialog_id** is used to request a dialog ID to be used for a subsequent TC dialog. The parameter user allows the **TcPduProvider** to deal with multiple **TcPduUser** objects as it must be able to associate incoming TC service primitive requests with the correct call-back interface. The exception NoMoreDialogs is raised if the object can support no more dialogs at this time.

```
void uni_req(in DialogQos qos,
             in TcAddress dest,
             in TcAddress orig,
             in DialogId d_id,
             in DialogPortion d_p,
             in ComponentList c_list)
    raises (NoMoreDialogs, LReject);
```

The operation **uni_req** is used to send a UNIDATA PDU to the TC/SS.7 stack (i.e., start an unstructured dialog). This will contain any components already queued with the same Dialog ID. A **DialogId** is supplied to correlate earlier components or later errors with this unstructured dialog. The Global Titles of the originator and destination must be supplied in the **orig** and **dest** parameters. The default Qos for the dialog is also supplied. TC **DialogPortion** information may be supplied. Note that the PDU size restrictions of TC must be complied with when creating any PDU. The **ComponentList** holds the components (if any) to be included in this PDU. The exception NoMoreDialogs is generated when the interface cannot currently handle any more dialogs, for example due to resource limitations. The LReject exception may be raised if received components are in error. If the Qos is set to **ENABLE_ERRORS**, then **notice_ind** primitives may be generated at the **TcPduUser** interface.

```
void begin_req(in DialogQos qos,
               in TcAddress dest,
               in TcAddress orig,
               in DialogId d_id,
               in DialogPortion d_p,
               in ComponentList c_list)
    raises (NoMoreDialogs, LReject);
```

The **begin_req** operation is used to start a TC structured dialog by sending a BEGIN PDU. This will contain any components already queued with the same Dialog ID. A **DialogId** is supplied to correlate earlier components or later structured dialog primitives with this dialog. All further structured dialog operations in this dialog must supply the same dialog ID as an **in** parameter. The Global Titles of the originator and

destination must be supplied in the **orig** and **dest** parameters. The default Qos for the dialog is also supplied. TC **DialogPortion** information may be supplied. Note that the PDU size restrictions of TC must be complied with when creating any PDU. The **ComponentList** holds the components (if any) to be included in this PDU. The exception NoMoreDialogs is generated when the interface cannot currently handle any more dialogs, for example due to resource limitations. The LReject exception may be raised if received components are in error. If the Qos is set to **ENABLE_ERRORS**, then **notice_ind** primitives may be generated at the **TcPduUser** interface.

```
void continue_confirm_req(
            in TcAddress orig,
            in DialogId d_id,
            in DialogPortion d_p,
            in ComponentList c_list)
    raises (InvalidDialogId, LReject);
```

The **continue_confirm_req** operation is used to send a reply to a TC-BEGIN in an already open structured dialog. This will contain any components already queued with the same Dialog ID. A new originating address may be supplied in the **orig** parameter. The **DialogId** supplied must be the same as the dialog ID of the dialog already open. TC **DialogPortion** information may be supplied. The **ComponentList** holds the components (if any) to be included in this PDU. The exception invalidDialogId is raised when the **TcPduProvider** has no structured dialog with the same dialog ID currently open. The LReject exception may be raised if received components are in error.

```
void continue_req(in DialogId d_id,
            in DialogPortion d_p,
            in ComponentList c_list)
    raises (InvalidDialogId, LReject);
```

The **continue_req** operation is used to send another PDU in an already open structured dialog. This will contain any components already queued with the same Dialog ID. The **DialogId** supplied must be the same as the dialog ID of the dialog already open. TC **DialogPortion** information may be supplied. The **ComponentList** holds the components (if any) to be included in this PDU. The exception invalidDialogId is raised when the **TcPduProvider** has no structured dialog with the same dialog ID currently open. The LReject exception may be raised if received components are in error.

```
void end_req(in DialogId d_id,
            in DialogPortion d_p,
            in TerminationType term,
            in ComponentList c_list)
    raises (InvalidDialogId, LReject);
```

The **end_req** operation is used to send another PDU in an already open structured dialog and to close that dialog. This will contain any components already queued with the same Dialog ID. The **DialogId** supplied must be the same as the dialog ID of an already open dialog. TC **DialogPortion** information may be supplied. TC

TerminationType information must also be supplied. The **ComponentList** holds the components (if any) to be included in this PDU. The exception invalidDialogId is raised when the **TcPduProvider** has no structured dialog with the same dialog ID currently open. The LReject exception may be raised if received components are in error.

```
void u_abort_req(in DialogId d_id,
                 in DialogPortion d_p)
    raises (InvalidDialogId);
```

The **u_abort_req** operation is used to send an ABORT PDU in an already open structured dialog and to close that dialog. The **DialogId** supplied must be the same as the dialog ID of an already open dialog. TC **Dialog Portion** abort information may be supplied. The exception InvalidDialogId is raised when the **TcPduProvider** has no structured dialog with the same dialog ID currently open.

```
void set_dialog_qos(in DialogId d_id,
                    in DialogQos qos)
    raises(InvalidDialogId, InvalidParameter);
```

The operation **set_dialog_qos** is used to change the Qos of a structured dialog while it is in progress. The **DialogId** must be supplied. The exception InvalidDialogId is raised when the **TcPduProvider** has no structured dialog with the same dialog ID currently open. The exception InvalidParameter is raised when an unknown value of **DialogQos** is set.

```
DialogQos get_dialog_qos(in DialogId d_id)
    raises(InvalidDialogId);
```

The operation **get_dialog_qos** is used to find the current Qos of a structured dialog while it is in progress. The **DialogId** must be supplied. The exception InvalidDialogId is raised when the **TcPduProvider** has no structured dialog with the same dialog ID currently open.

```
void u_cancel_req(in DialogId d_id,
                  in InvokeId ivk_id)
    raises(InvalidDialogId, InvalidParameter);
```

The **u_cancel_req** operation is used to discard a previously queued component and corresponds to the TC-U-CANCEL primitive. The parameter **d_id** carries the dialog ID of the dialog to which this component belongs. The **member ivk_id** must carry the invoke ID of the rejected component. The exception InvalidDialogId is raised if the **DialogId** supplied has not been allocated by the **TcPduProvider** or refers to a closed dialog. The exception InvalidParameter is raised when no operation with the specified invoke ID is outstanding.

```
void destroy()
    raises(DialogStillOpen);
```

The operation destroy is used to end a TC session. All dialogs must be ended before a session can finish. The exception DialogStillOpen is raised if there are still TC structured dialogs open.

Figure 3-1 illustrates the use of the **TcPduProviderFactory** and **TcPduProvider** interfaces to support the initiation of a TC dialog from a TC-aware CORBA object. A **TcPduUser** requests the creation of a **TcPduProvider** object (step 1) providing its own object reference. The **TcPduUser** then invokes a **begin_req** operation (step 2) which carries a TC invoke request as component data. The **TcPduProvider** uses the SS7 stack to communicate with the external TC-User by generating the BEGIN PDU (step 3) and receives a CONTINUE PDU (step 4). It indicates receipt of a CONTINUE PDU by invoking a **continue_ind** operation which carries a TC result indication as component data to the **TcPduUser**.

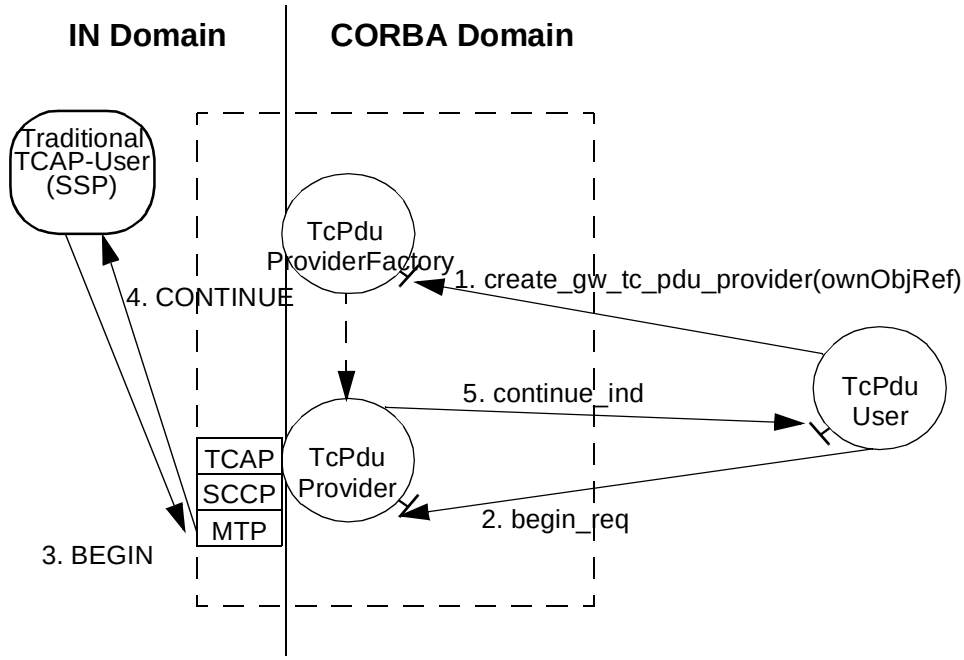


Figure 3-1 Use of a **TcPduProviderFactory** to support dialog initiation by a TC-aware CORBA object

3.3.4 The TcPduUserFactory Interface

The **TcPduUserFactory** interface is used to create new **TcPduUser** objects that may handle sessions for a particular application context.

```
interface TcPduUserFactory{
TcPduUser create_tc_pdu_user
        (in ApplicationContext application_context)
    raises(NoMoreDialogs);
}; // end TcPduUserFactory
```

The **create_tc_pdu_user** operation allows an object to start a session by returning a reference to a **TcPduUser** object. The **ApplicationContext** parameter identifies the context with which the **TcPduUser** will be identified with. The NoMoreDialogs exception is raised if the **TcPduUserFactory** is unable to create any more **TcPduUser** objects at this time.

3.3.5 The TcPduUser Interface

This interface is used by the TC/SS.7 stack to communicate with CORBA objects. There are two modes of use:

1. As a call back interface for a particular TC session started by a CORBA object.
2. As an initial call handler interface for a particular TC/SS.7 address (Global Title, Application Context) whose factory object (a **TcPduUserFactory**) has been registered with the **TcPduProviderFactory**.

It provides TC dialog handling primitives that may carry TC components if desired. There are also some component-oriented primitives supported. Finally, there are **get** and **set** operations for dialog Qos so that the TC/SS.7 stack can inform the CORBA object of changes in Qos during a TC structured dialog.

```
void uni_ind(in TcPduProvider sender,
             in DialogQos qos,
             in TcAddress dest,
             in TcAddress orig,
             in DialogId d_id,
             in DialogPortion d_p,
             in ComponentList c_list)
    raises (NoMoreDialogs);
```

The operation **uni_ind** is used to indicate reception of a UNIDATA PDU to a CORBA object (start an unstructured dialog). The sending **TcPduProvider** object places its own reference in the first parameter. The Global Titles of the originator and destination must be supplied in the **orig** and **dest** parameters. The default Qos for the dialog is also supplied. TC **DialogPortion** information may be supplied. The **DialogId** for the current dialog is supplied to allow coordination with further messages in the dialog. The **ComponentList** holds the components (if any) included in this PDU. The exception NoMoreDialogs is generated when the interface cannot currently handle any more dialogs, for example due to resource limitations.

```
void begin_ind(in TcPduProvider sender,
               in DialogQos qos,
               in TcAddress dest,
               in TcAddress orig,
               in DialogId d_id,
               in DialogPortion d_p,
               in ComponentList c_list)
    raises (NoMoreDialogs);
```

The **begin_ind** operation is used to indicate reception of a UNIDATA PDU to a CORBA object (the start a TC structured dialog). The sending **TcPduProvider** object places its own reference in the first parameter. The Global Titles of the originator and destination must be supplied in the **orig** and **dest** parameters. The default Qos value for the dialog is also supplied. TC **DialogPortion** information may be supplied. A **DialogId** is supplied to correlate later dialog operations with this invocation. All further structured dialog operations in this dialog must supply the same dialog ID as an **in** parameter. The **ComponentList** holds the components (if any) included in this PDU. The exception NoMoreDialogs is generated when the interface cannot currently handle any more dialogs, for example due to resource limitations.

```
void continue_ind(in DialogId d_id,
                  in DialogPortion d_p,
                  in ComponentList c_list)
    raises (InvalidDialogId);
```

The **continue_ind** operation is used to indicate receipt of a CONTINUE PDU in an already open TC structured dialog. The **DialogId** supplied must be the same as the dialog ID of an already open dialog. TC **DialogPortion** information may be supplied. The **ComponentList** holds the components (if any) included in this PDU. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open.

```
void end_ind(in DialogId d_id,
             in DialogPortion d_p,
             in ComponentList c_list)
    raises (InvalidDialogId);
```

The **end_ind** operation is used to indicate reception of another PDU in an already open TC structured dialog and to end that dialog. The **DialogId** supplied must be the same as the dialog ID of an already open dialog. TC **DialogPortion** information may be supplied. The **ComponentList** holds the components (if any) included in this PDU. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open.

```
void u_abort_ind(in DialogId d_id,
                 in DialogPortion d_p)
    raises (InvalidDialogId);
```

The **u_abort_ind** operation is used to inform a CORBA object that a dialog has been aborted by the remote TC user (TC-U-ABORT primitive). The dialog has prematurely ended and all associated operations are also aborted. The **DialogId** references the dialog which has been aborted. TC **DialogPortion** information may be supplied. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open.

```
void notice_ind(in DialogId d_id,
                in short report_cause)
    raises (InvalidDialogId);
```

The **notice_ind** operation is used to inform a CORBA object that there has been a delivery failure for a PDU in the current dialog (TC-NOTICE primitive). It shall only be invoked when the Qos for the dialog has been set to report errors. The **DialogId** references the dialog which has been aborted. The **report_cause** parameter holds the TC NOTICE report cause information. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open.

```
void l_cancel_ind(in DialogId d_id
                  in InvokeId ivk_id)
    raises(InvalidDialogId, InvalidParameter);
```

The **l_cancel_ind** operation is used to indicate timer expiry of a TC operation invocation. The CORBA object must now treat this operation as failed. The **DialogId** supplied must be the same as the dialog ID of an already open dialog. The **InvokeID** of the operation whose timer has expired is also supplied. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open. The exception InvalidParameter is raised when no operation with the specified invoke ID is outstanding.

```
void p_abort_ind(in DialogId d_id,
                 in PAbortReason reason)
    raises (InvalidDialogId);
```

The **p_abort_ind** operation is used to inform a **TcPduUser** object that a dialog has been aborted by the lower layers (TC-P-ABORT primitive). All associated operations are also aborted. The **DialogId** references the dialog that has been aborted. The **PAbortReason** field is used to carry the provider abort information. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open.

```
DialogQos get_dialog_qos(in DialogId d_id)
    raises(InvalidDialogId);
```

The **get_dialog_qos** operation allows the TC/SS.7 stack to query the current value of QoS in a particular dialog. The **DialogId** of the queried dialog must be supplied. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open.

```
void set_dialog_qos(in DialogId d_id,
                    in DialogQos qos)
    raises(InvalidDialogId, InvalidParameter);
```

The **set_dialog_qos** operation allows the TC/SS.7 stack to inform the CORBA object of changes in the QoS of a structured dialog. The **DialogId** identifies which of the currently open dialogs is changing the QoS. The exception InvalidDialogId is raised when the **TcPduUser** has no structured dialog with the same dialog ID currently open. The exception InvalidParameter is raised when an unknown value of **DialogQos** is set.

Figure 3-2 shows the use of the **TcPduUser** and the **TcPduUserFactory** interfaces when a dialog is initiated from the SS7 domain. It is assumed that a **TcPduUserFactory** object has registered with the **TcPduProviderFactory** its interest (step 1) in receiving notice of received TC PDUs destined for a particular GT and a particular AC.

The **TcPduProviderFactory** has created an appropriate **TcPduProvider** object if one with the appropriate characteristics does not already exist. When the **TcPduProvider** receives (see step 2) a TC BEGIN PDU with an Invoke component containing operation1, it invokes (see step 3) a **create_tc_pdu_user** operation on the registered **TcPduUserFactory** object.

A **begin_ind** operation is then sent (step 4) to the **TcPduUser** object with an **in** parameter identifying operation1. The result of this operation invocation is sent as an **in** parameter **result_op1** of another invocation **continue_ind** (see step 5) on the **TcPduProvider**, which formats a TC CONTINUE PDU for sending to the external TC-User.

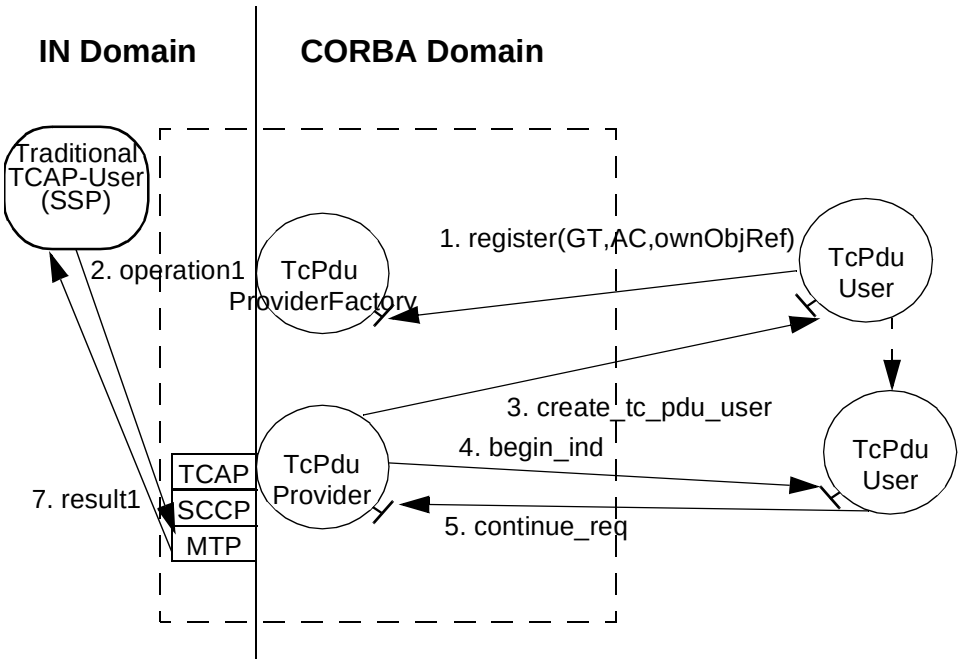


Figure 3-2 Role of a TcPduUser interface upon initiation of a Dialog from the SS7 domain

3.4 Integration of Interfaces

3.4.1 Integration of TC PDU-oriented Interfaces and Interworking Interfaces

The **TcPduUser** interfaces define a framework for building TC-aware CORBA applications. Their use depends on a detailed knowledge of TC. The TC-User interfaces generated by Specification Translation reflect the semantics of TC-Users

within a CORBA framework. Their use is not limited to gateway implementations and is less constrained by the standard TC requirements. The implementation at a gateway of a proxy object (based on the Specification Translation interfaces) will be constrained by TC. Hence a proxy object may be implemented using the TC PDU-oriented interfaces instead of the proprietary API of a specific TC/SS.7 stack. These two possibilities are illustrated in the figure below.

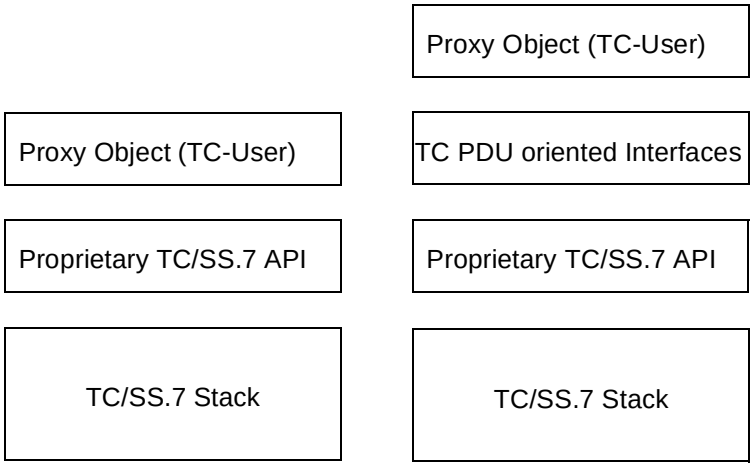


Figure 3-3 Two Proxy Object Implementation Strategies

The TC PDU-oriented interfaces can be used to build generic incoming dialog handlers at a gateway. In the scenario illustrated in Figure 3-4 on page 3-17, a **TcPduUser** object uses the results of a **resolve** operation to determine what type of proxy object to create for a particular incoming call. The proxy object then takes over the dialog with the receiving “SCP” (another object generated by specification translation).

The IDL interfaces defined for TC PDU handling also allow proxy objects to be easily distributed across multiple nodes instead of residing on the same node as the SS.7 stack hardware.

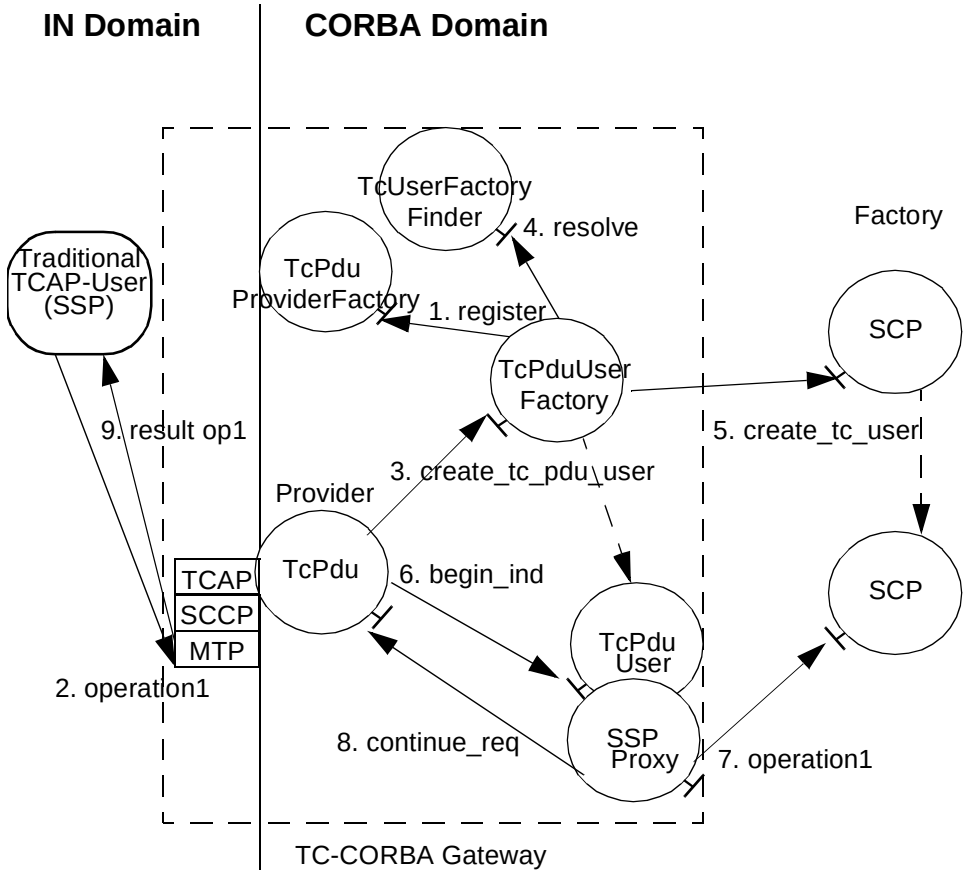


Figure 3-4 Using the TC PDU-oriented interfaces with Interworking Interfaces

3.4.2 Application Location and Dialog Initiation

The process of initiating a dialog is shown in the Message Sequence Chart (MSC) in Figure 3-3 on page 3-16. This MSC considers the case of dialog initiation and continuation from the external (IN) legacy domain. Furthermore, to reduce complexity, we have considered the use of TC without the use of Application Context (i.e., the dialog Portion). The steps, briefly, are as follows.

1. A **TcPduUserFactory** object registers as the factory for creation of **TcPduUser** objects which are to receive TC messages destined for a given Global Title (GT) at a SS7/TC-to-CORBA gateway. It does so by invoking a **register** operation on a **TcPduProviderFactory** object, providing its own object reference (at which it wishes to receive creation requests for **TcPduUsers**) and the GT in question. The **TcPduProviderFactory** creates an instance of a **TcPduProvider** object assuming that one for that GT has not already been created. The **TcPduProvider** object at the gateway is the CORBA proxy for the underlying TC/SS7 protocol stack.

2. When a BEGIN APDU is received at a gateway from the external system (the SSP) addressed to a GT for which there is a registered **TcPduUserFactory**, the **TcPduProvider** invokes a **create_tc_pdu_user** operation on the **TcPduUserFactory**. The reference to the **TcPduUser** is returned and the **TcPduProvider** invokes a **begin_ind** operation on the **TcPduUser** interface passing in as parameters, among other things, any invocations received with the BEGIN as well as an object reference by which the **TcPduUser** may invoke operations on the **TcPduProvider**.

3. The **TcPduUser** can use the **GwAdmin** interface (not shown in the MSC) to obtain the reference to a **TcFactoryFinder** object, on which it invokes a **resolve** operation to obtain the object references of factory interfaces bound to the GT passed as a parameter in the invocation. The **TcFactoryFinder** interface makes use of the CORBA Naming Service (not shown in the MSC) to perform the name resolution. (It is assumed that the name bindings have been performed at some earlier time).

4. The **TcUserFactory** reference (a single one in the case where Application Context is not supported) returned by the address resolution is used to create an instance of a SSP (a proxy object at the gateway) and an instance of the target CORBA object, the SCP.

5. The SSP proxy invokes the IDL equivalent of the received TC/ROS invocation on the SCP. (The MSC for converting the identifier of the received TC/ROS operation to its IDL scoped name is shown in Figure 3-5 on page 3-19. In Figure 3-3 on page 3-16, those steps are assumed to have been performed.)

6. The returned result is included in a **continue_req** operation on the **TcPduProvider** interface, where a CONTINUE PDU is generated to be sent to the SSP.

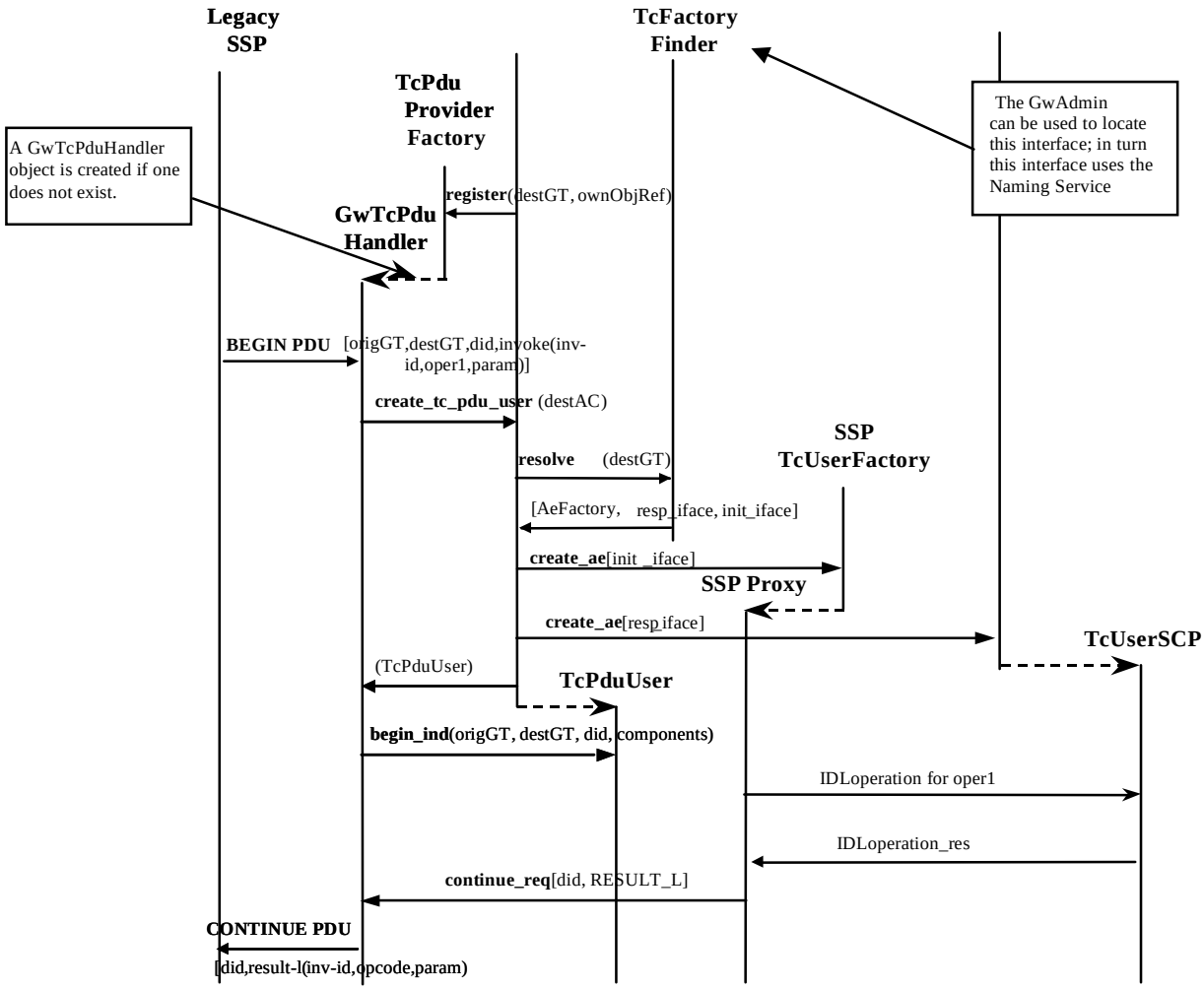


Figure 3-5 Dialog initiation with TC PDU-oriented interfaces with Interworking Interfaces

SCCP Inter-ORB Protocol (SIOP) 4

Contents

This chapter contains the following sections.

The mapping of GIOP message transfer to SCCP class 1 service (sequenced connectionless service) is called SCCP Inter-ORB Protocol (SIOP). SCCP is applied as defined in ITU-T Recommendations Q.711 through Q.714 [**16**].

SIOP version 1.2 clients must support GIOP 1.0 or GIOP 1.1 or GIOP 1.2. SIOP version 1.2 servers must support GIOP 1.0 and GIOP 1.1 and GIOP 1.2. An SIOP server that receives GIOP requests with a particular version must answer with GIOP messages of the same version.

4.1 Usage of SCCP Services

The object reference of a server being reachable over SIOP is constructed as explained in the next section. The IOR contains an SCCP address that will identify the signalling point where the server is located.

SCCP class 1 is a connectionless service. The sender and recipient of SCCP messages are identified by an SCCP address, which consists of a global title (a string) and some additional information described below. The SCCP address identifies a signalling point (a node, e.g., a workstation), which is connected to the Signalling System No. 7. There may be situations where on this node a magnitude of servers are running possible in different capsules [**17**] (e.g., different operation system processes). It is also possible

that several clients in different capsules have to communicate over an SCCP. In this case, the communication of the servers and clients has to be multiplexed over a single access point to the Signalling System No. 7. SCCP has no mechanism to identify a group of clients or servers that belong together (e.g., by belonging to the same capsule). Other protocols like TCP/IP provide such a mechanism (e.g., with a port number).

To facilitate such a multiplexing mechanism, endpoint identifiers are defined for SIOP. An endpoint identifier is an arbitrary unsigned short greater than zero which identifies a communication endpoint for a group of objects that can be reached over an access to the Signalling System No. 7 identified by an SCCP address. An endpoint identifier with the value 0 is reserved for error handling purposes. If a client communicates with a server over SIOP both have to provide their endpoint identifier. It is outside the scope of this specification to discuss how endpoint identifiers are allocated and distribution of messages to different capsules is provided. It is also not specified here how clients and servers are associated with a specific endpoint identifier. However one object may be associated to different endpoint identifiers at the same time (e.g., one for the role as client and another for the role as server).

An SIOP message starts with the CDR encapsulation of **MessageHeader** as defined in the module **SIOP**:

```
module SIOP { // IDL
    typedef unsigned short EndpointId;
    struct Version {
        octet major;
        octet minor;
    };

    const octet ERROR_MESSAGE = 0;
    const octet NORMAL_MESSAGE = 1;

    struct MessageHeader {
        char magic [4];
        Version SIOP_version;
        octet flags;
        octet message_type;
        EndpointId calling_endpoint;
        EndpointId called_endpoint;
        unsigned long message_size;
    };
};
```

The value of **message_type** is either **ERROR_MESSAGE** or **NORMAL_MESSAGE**.

Note – Octet constants are allowed according to the solution to issue 725 of the Core RTF report ptc/98-07-05.

If ...	Then ...
the **message_type** field of **MessageHeader** identifies a message of type **ERROR_MESSAGE**	no other components follow the **MessageHeader**. This message is only sent if a received message did not comply to the message format stated here or if the **called_endpoint** field in a received message with **message_type** of **NORMAL_MESSAGE** did not address a valid group of objects. It is not specified here what it means that a **calling_endpoint** is valid.
the message had a wrong format	the **calling_endpoint** and **called_endpoint** both have to be set to 0. In the case of invalid **called_endpoint**, the **calling_endpoint** has to be the same as the **called_endpoint**, which was wrong in the received message and the **called_endpoint** has to be the same as the **calling_endpoint** in the received message. For instance - if a message with **message_type** set to **NORMAL_MESSAGE** is received with a **calling_endpoint** of 5 and a **called_endpoint** of 3, but 3 does not address a valid group of objects, then a message with **message_type** set to **ERROR_MESSAGE** is returned with **calling_endpoint** 3 and **called_endpoint** 5.
the message could not be decoded at all (it did not comply to the message format)	a message with **message_type** set to **ERROR_MESSAGE** is returned with **calling_endpoint** 0 and **called_endpoint** 0.
the message_type field is **NORMAL_MESSAGE**	a GIOP message follows.

The fields in **MessageHeader** are defined as follows:

- **magic** identifies the SIOP protocol with the string “SIOP“ encoded in ISO Latin-1 (8859.1).
- **SIOP_version** identifies the version of SIOP this message complies to. The **SIOP_version** is independent from the GIOP version of the GIOP message. SIOP specified here has a major version 1 and a minor version 2.
- **flags** is used to identify the byte order of the members **calling_endpoint**, **called_endpoint**, and **message_size**. A zero value at the least significant bit indicates big-endian byte ordering, and a one indicates little-endian byte ordering.
- **message_type** identifies the type of the message, which is either **ERROR_MESSAGE** or **NORMAL_MESSAGE**.
- **calling_endpoint** identifies the endpoint identifier of the sender of the message. It is encoded according to the byte ordering specified in **flags**.

- **called_endpoint** identifies the endpoint identifier of the receiver of the message. It is encoded according to the byte ordering specified in **flags**.
- **message_size** contains the number of octets of the GIOP message following the SIOP header in case of a **message_type** with the value **NORMAL_MESSAGE**. The value of **message_size** is zero in case of **ERROR_MESSAGE**. It is encoded according to the byte ordering specified in **flags**.

The encoded **SIOP::MessageHeader** and the GIOP message in case of a **NORMAL_MESSAGE** will form together an SIOP message.

The client and the server will send SIOP messages using the SCCP class 1 N-UNITDATA primitive. The sequence control field of N-UNITDATA is set to “sequence guaranteed.” The return option is set to “return message on error.” It is not mandatory that the beginning of an SIOP message be aligned with the beginning of the N-UNITDATA user data field. It is also not mandatory that the end of an SIOP message be aligned with the end of the N-UNITDATA user data field. However the user data field of N-UNITDATA messages will contain nothing other than SIOP messages (i.e., no additional padding is applied).

A subsystem number used for SIOP over SCCP will be selected by the network operator. If international interoperability is necessary, transparent use of the subsystem number across international boundaries can be used with multi-lateral agreement between the involved parties. If further standardization is needed for interoperability, a subsystem number can be assigned by ITU-T together with a revision of this document.

Note – ITU-T considers that subsystem numbers currently used in traffic between different national networks with bilateral agreement can be transferred unchanged over international boundaries. According to the old SCCP standard it should be set to 0. See COM11-D909/WP5 for the ITU SG11 May meeting. For the standardization of a new SSN for SIOP, the usage of SIOP should be first monitored. If there is a real need, the assignment of a standard SSN could happen quickly.

SCCP class 1 is a connectionless service; therefore, it is not necessary to open a connection to be able to make client server communication. However, it is not an error that one side sends a GIOP **CloseConnection** message. The receipt of this message has no effect. The sending of the message is deprecated.

Each SCCP message contains an SCCP address for the calling and called party number. The calling party number identifies the signalling point where the client is located in case of **Request**, **LocateRequest**, **Fragment**, and **CancelRequest** messages. The calling party number identifies the signalling point where the server is reachable in case of **Reply**, **Fragment**, or **LocateReply** messages. The called party number identifies the signalling point according to the SCCP address in the SCCP IOR profile if the client sends a **Request**, **CancelRequest**, **Fragment**, or **LocateRequest** message. The called party number in case of a **Reply**, **LocateReply**, or **Fragment** is the same as the calling party number of the related **Request**, **LocateRequest**, or **Fragment** message. The called party number of **MessageError** messages is the calling party number of the message that was erroneous. The calling party number addresses the signalling point where the message was received.

The **request_id** unambiguously associates replies with a request per tuple of calling party number, called party number, calling_endpoint, and called_endpoint.

The reception of an N-NOTICE indicates that the network was not able to deliver a certain N-UNITDATA to the called party number. In this case, all requests with outstanding answers are considered unsuccessful. The clients are informed with a COMM_FAILURE exception if appropriate. Since SCCP is connection-less bi-directional GIOP is used by default. No special **IOP::ServiceContext** is defined (i.e., no **BiDirSIOPServiceContext**). Setting a **BiDirectionalPolicy** has no effect.

4.2 SIOP IOR Profiles

An IOR of an object being reachable over SCCP class 1 has either a profile with the tag **TAG_SCCP_IOP** present, or a **TAG_MULTIPLE_COMPONENTS** profile with the components **TAG_COMPLETE_OBJECT_KEY**, **TAG_SCCPADDRESS**, and **TAG_SIOP_VERSION**, or profiles of both kinds. The following definitions are used:

```
module SIOP { // IDL

    typedef octet TranslationTypeIndicator;

    union TranslationType switch (boolean) {
        case TRUE:
            TranslationTypeIndicator translation_type_indicator;
    };

    typedef unsigned short NatureOfAddressIndicator;

    union NatureOfAddress switch (boolean) {
        case TRUE:
            NatureOfAddressIndicator nature_of_address_indicator;
    };
    const NatureOfAddressIndicator
            SUBSCRIBER_NUMBER = 1;
    const NatureOfAddressIndicator
            NATIONAL_SIGNIFICANT_NUMBER = 3;
    const NatureOfAddressIndicator
            INTERNATIONAL_NUMBER = 4;

    typedef unsigned short NumberingPlanIndicator;

    union NumberingPlan switch (boolean) {
        case TRUE:
            NumberingPlanIndicator numbering_plan_indicator;
    };

    const NumberingPlanIndicator UNKNOWN = 0;
    const NumberingPlanIndicator ISDN_TELEPHONY_E164 = 1;
```

```
        struct GlobalTitleIndicator {
            string global_title;
            TranslationType translation_type;
            NatureOfAddress nature_of_address;
            NumberingPlan numbering_plan;
        };

        union GlobalTitle switch (boolean) {
            case TRUE: GlobalTitleIndicator global_title_indicator;
        };

        typedef unsigned short SPCIndicator;

        union SPC switch (boolean) {
            case TRUE: SPCIndicator spc_indicator;
        };

        typedef octet SSNIndicator;

        union SSN switch(boolean) {
            case TRUE: SSNIndicator ssn_indicator;
        };

        struct SCCPAddress {
            GlobalTitle global_title;
            SPC     signalling_point_code;
            SSN     sub_system_number;
        };

        struct ProfileBody {
            Version SIOP_version;
            SCCPAddress address;
            EndpointId endpoint;
            sequence<octet> object_key;
            sequence <IOP::TaggedComponent> components;
        };

        struct ContactInfo {
            Version SIOP_version;
            SCCPAddress address;
            EndpointId endpoint;
        };
    };

    module IOP { // IDL

        const ProfileId TAG_SCCP_IOP = 2;

        const ComponentId TAG_SCCP_CONTACT_INFO = 24;
    };
```

4.2.1 Multiple Component Profile

If the object is addressed by a **TAG_MULTIPLE_COMPONENTS** profile, the **TaggedComponents** include components for **TAG_COMPLETE_OBJECT_KEY** and **TAG_SCCP_CONTACT_INFO**. Other components may be present.

The component for **TAG_COMPLETE_OBJECT_KEY** is defined in the DCE ESIOP chapter in [CORBA2.2]. It identifies the key of the object being addressed by the IOR.

4.2.2 The SCCP Contact Info Component

A **TAG_MULTIPLE_COMPONENTS** profile must contain one or more components with the tag **TAG_SCCP_CONTACT_INFO**. The **component_data** contains the CDR encapsulation of an **SIOP::ContactInfo**. Each such contact info identifies:

- in the **address** member - the SCCP address of the signalling point to which messages can be sent to reach the identified object (see below),
- in the **SIOP_version** member - the SIOP version. The version consists of a major and a minor version number of SIOP that the agent at the specified address is prepared to receive. The agent must be able to accept any SIOP message with the specified version or messages with smaller minor version numbers. SIOP versions with different major numbers may not be compatible.
- in the **endpoint** member - the endpoint identifier that identifies the group to which the object belongs. This endpoint identifier must be used as **called_endpoint** in the **SIOP::MessageHeader** in each message directed to the object.

An **SCCPAddress** has the members **signalling_point_code, sub_system_number**, and **global_title** which are of a **union** type with a **boolean** discriminator. A discriminator value TRUE indicates that a value for that member is given.

The signalling point code is given as value of the type **SPCIndicator** that has a value between 0 and 65535. Some networks may further restrict this value range. The global title is given as a value of the type **GlobalTitleIndicator**. The subsystem number has a value between 0 and 255.

The **global_title** of **GlobalTitleIndicator** contains the global title that addresses the signalling point. It consists of the characters **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **A**, **B**, **C**, **D**, **E**, **F** in ISO Latin-1 (8859.1) only. These characters result in the following BCD encoded address signals. An SCCP address contains multiple BCD encoded digits - called address signals - for the global title:

Table 4-1 BCD Codes

Character	BCD Code
0	**0000**
1	**0001**
2	**0010**

Table 4-1 BCD Codes

Character	BCD Code
3	**0011**
4	**0100**
5	**0101**
6	**0110**
7	**0111**
8	**1000**
9	**1001**
A	**1010**
B	**1011**
C	**1100**
D	**1101**
E	**1110**
F	**1111**

The length of the global title may be odd or even. Only BCD encoding is used for SCCP address information.

The members **translation_type**, **nature_of_address** and **numbering_plan** are of a union type with a **boolean** discriminator. The value TRUE of the discriminator indicates that a value for this field is given. The value FALSE of the discriminator indicates that no value was provided in the appropriate field of the SCCP address.

The following combinations of discriminator values are allowed:

Table 4-2 Discriminator Values

discriminator value for	translation_type	numbering_plan	nature_of_address
Type 1	FALSE	FALSE	FALSE
Type 2	TRUE	TRUE	FALSE
Type 3	TRUE	TRUE	FALSE
Type 4	TRUE	TRUE	TRUE

The member **translation_type_indicator** indicates the 8 bits necessary for the translation type.

The member **nature_of_address_indicator** has a value between 0 and 127 (including 0 and 127). The value indicates the nature of address of the global title. The three values **SUBSCRIBER_NUMBER**, **NATIONAL_SIGNIFICANT_NUMBER**, and **INTERNATIONAL_NUMBER** are predefined for this field.

The member **numbering_plan_identification** has a value between 0 and 15 (including 0 and 15). This value indicates the numbering plan of the global title. The two values **UNKNOWN** and **ISDN_TELEPHONY_E164** are predefined for this field.

4.2.3 The TAG_SCCP_IOP profile

In case the IOR contains a profile with the tag **TAG_SCCP_IOP**, the **profile_body** is an instance of an **SIOP::ProfileBody** that is marshaled into an encapsulation.

The members of **SIOP::ProfileBody** are defined below:

- **SIOP_version** describes the version of SIOP that the agent at the specified address is prepared to receive. The agent must be able to accept any SIOP message with the specified version or messages with smaller minor version numbers. SIOP versions with different major numbers may not be compatible.
- **address** identifies the signalling point where messages for the specified object may be sent. See Section 4.2.2, “The SCCP Contact Info Component,” on page 4-7 for details of the values of this member.
- **endpoint** specifies the endpoint identifier of the group the object belongs to. See Section 4.2.2, “The SCCP Contact Info Component,” on page 4-7 for details of the values of this member.
- **object_key** is an opaque value supplied by the agent producing the IOR. The semantics is the same as the member object_key of **IIOP::ProfileBody_1_0** and **IIOP::ProfileBody_1_1** in [CORBA2.2].
- **components** is a sequence of **TaggedComponent**, which contains additional information that may be used in making invocations on the object described by this profile. TaggedComponents that apply to SIOP 1.0 are the following:
  - **TAG_ORB_TYPE**
  - **TAG_CODE_SETS**
  - **TAG_JAVA_CODEBASE**

For a description of these components see [CORBA2.2] and [CORBASecurity].

References A

A.1 List of References

[1] Subrata Mazumdar and Nilo Mitra, “ROS-to-CORBA Mappings: First Step towards Intelligent Networking using CORBA”, Proceedings of Conference on Intelligence in Services and Networks, 1997, Como, Italy.

[2] Subrata Mazumdar and Nilo Mitra, “Design of a ROS-CORBA Gateway for Interoperable Intelligent Networking Applications”, second part of AT&T Response to OMG RFI on Issues Concerning Intelligent Networking with CORBA.

[3] JIDM Interaction Translation, Final submission to OMG's CORBA/TMN Interworking RFP, telecom/98-

[4] The Open Group, Preliminary Specification Inter-Domain Management: Specification Translation, X/Open Document Number: P509, ISBN: 1-85912-150-0

[5] Lucent Technologies, Initial Submission to OMG’s CORBA/TMN Interworking RFP, SNMP Part

[6] N. Mitra and S. D. Usiskin, “Interrelationship of the SS7 Protocol Architecture and the OSI Reference Model and Protocols,” The Froehlich/Kent Encyclopedia of Telecommunications, Volume 9, Marcel Dekker, Inc., 1995.

[7] ITU-T Rec. Q.1400 “Architecture Framework for the Development of Signalling and OA&M Protocols using OSI Concepts”

[8] ITU-T Rec. Q.1218 “Interface Recommendation for Intelligent Network CS-1”, Geneva, 1995.

[9] ITU-T Rec. Q.771, Signalling System No. 7 - Functional Description of Transaction Capabilities

ITU-T Rec. Q.772, Signalling System No. 7 - Transaction Capabilities Information Element Definitions

ITU-T Rec. Q.773, Signalling System No. 7 - Transaction Capabilities Formats and Encodings

ITU-T Rec. Q.774, Signalling System No. 7 - Transaction Capabilities Procedures

ITU-T Rec. Q.775, Signalling System No. 7 - Guidelines for Using Transaction Capabilities

[10] ITU-T Rec. X.680 through 683 (1994) | ISO/IEC 8824-1/2/3/4:1995, Information technology - Open Systems Interconnection - Abstract Syntax Notation One (ASN.1).

[11] ETSI, ETS 300 374-1, “Intelligent Network(IN); Intelligent Network Capability Set 1 (CS1); Core Intelligent Network Application Protocol(INAP); Part1:Protocol Specification”, 1997.

[12] ETSI, European digital cellular telecommunications system (phase1); Mobile application part specification, I-ETS 300 044 (GSM 09.02)

[13] EURESCOM, “Introduction to Distributed Computing Middleware in Intelligent Networks: A Eurescom P508 Perspective,” OMG document orbos/97-09-11

[14] ITU-T Rec. X.880 (1994) | ISO/IEC 13712-1:1995, Information technology - Remote Operations: Concepts, model and notation.

[15] OMG Request for Proposal on Real-Time CORBA, orbos/98-01-08.

[16] ITU-T Rec. Q.711, Signalling System No. 7 - Functional Description of the Signalling Connection Control Part, Geneva, March 1993

ITU-T Rec. Q.712, Signalling System No. 7 - Definition and Function of SCCP Messages, Geneva, March 1993

ITU-T Rec. Q.713, Signalling System No. 7 - SCCP Formats and Codes, Geneva, March 1993

ITU-T Rec. Q.714, Signalling System No. 7 - SCCP Procedures, Geneva, March 1993

[17] ISO/IEC 10746-2: Open Distributed Processing - Reference Model: Architecture, March 1993

[18] OMG TC document orbos/98-11-31, Objects by Value, Joint Revised Submission (version 2)

Complete IDL B

B.1 The TcSignalling Module

```
//File: TcSignaling.idl
#ifndef _TC_SIGNALING_
#define _TC_SIGNALING_

// This module defines base and utility interfaces for
// CORBA-based Telecom SS.7 Transaction Capabilities(TC) users

#include <Naming.idl>
#include <CosLifeCycle.idl>
#include “ASN1Types.idl”
#pragma prefix “omg.org”
module TcSignaling{

    typedef short ASN1_ExtensionCriticality ;
    // ASN1_ExtensionCriticality values
    const ASN1_ExtensionCriticality ASN1_EXTENSION_ABORT = 0;
    const ASN1_ExtensionCriticality ASN1_EXTENSION_IGNORE = 1;

    typedef string ScopedName;
    enum IdType {LOCAL_ID, OID};
    union IdValue switch(IdType) {
        case LOCAL_ID: ASN1_Integer local_id;
        case OID: ASN1_ObjectIdentifier oid;
    };

    typedef CosNaming::Istring Istring;
    typedef long AssociationId;
    typedef Istring TcAddress;
    typedef Istring ApplicationContext;
    typedef long InvokeId;
```

```
// Range -128 to +127 for Q.773
// InvokeID values
const InvokeId NO_ID = 2000000000;

typedef short DialogFlowCtr;
// DialogFlowCtr values
const DialogFlowCtr BEGIN = 0;
const DialogFlowCtr CONTINUE = 1;
const DialogFlowCtr END = 2;
const DialogFlowCtr QUEUE_COMPONENT = 3;
const DialogFlowCtr UNIDIRECTIONAL = 4;
const DialogFlowCtr NOT_SPECIFIED = 5;

typedef short TcContextSetting;
// TcContextSetting values
const TcContextSetting TC_CONTEXT_BASE = 0;
const TcContextSetting TC_CONTEXT_NO_FLOW = 1;
const TcContextSetting TC_CONTEXT_ALL = 2;

struct TcContext{
     DialogFlowCtr ctr;
     InvokeId ivk_id;
     InvokeId lnk_id;
     AssociationId a_id;
     };

typedef sequence<CosLifeCycle::NameValuePair> DialogUserData;
typedef short PAbortReason;
// PAbortReason values
const PAbortReason UNRECOG_MESSAGE_TYPE = 0;
const PAbortReason UNRECOG_TID = 1;
const PAbortReason BAD_FORMAT_TRANSACTION = 2;
const PAbortReason INCORRECT_TRANSACTION = 3;
const PAbortReason RESOURCE_LIMIT = 4;
enum AbortType {P_ABORT,
               UNSUPPORTED_APPLICATION_CONTEXT,
               USER_DEFINED_INFO
               };
union AbortValue switch(AbortType) {
case P_ABORT: PAbortReason reason;
case UNSUPPORTED_APPLICATION_CONTEXT:
ApplicationContext a_c;
case USER_DEFINED_INFO: DialogUserData d_u_d;
};
exception UnknownAssociation{};
exception NoMoreAssociations{};
exception InvalidParameter{};
exception UnsupportedTcContext{};

interface TcUser:CosLifeCycle::LifeCycleObject {
```

```
void abort_association(in AssociationId a_id)
    raises (UnknownAssociation);

 void abort_association_with_data(in AbortValue abort_value,
            in AssociationId a_id)
    raises (UnknownAssociation, InvalidParameter);

void end_association (in AssociationId a_id)
    raises (UnknownAssociation);

AssociationId new_association(in TcUser initiator,
            in AssociationId a_id)
    raises(NoMoreAssociations);

AssociationId new_association_with_dialogdata(
            in TcUser initiator,
            in AssociationId a_id,
            in string protocol_version,
            in DialogUserData d_u_d)
    raises(NoMoreAssociations, InvalidParameter);

readonly attribute TcContextSetting tc_context_setting;

}; //end TcUser

interface TcUserGenericFactory{

    TcUser create_tc_user_responder(
                in ScopedName responder,
                in TcUser initiator,
                in AssociationId a_id,
                in TcContextSetting tc_context_setting,
                        out AssociationId a_id_rtn)
        raises(CosLifeCycle::NoFactory,
            NoMoreAssociations, UnsupportedTcContext);

    TcUser create_tc_user_responder_with_dialog_data(
                in ScopedName responder,
                in TcUser initiator,
                in AssociationId a_id,
                in string protocol_version,
                in DialogUserData d_u_d,
                in TcContextSetting tc_context_setting,
                        out AssociationId a_id_rtn)
        raises(CosLifeCycle::NoFactory, NoMoreAssociations,
            InvalidParameter, UnsupportedTcContext);

    TcUser create_tc_user_initiator(
                in ScopedName initiator)
        raises(CosLifeCycle::NoFactory);
```

```
}; // end TcUserGenericFactory

interface TcFactoryFinder{

    void bind(in TcAddress addr,
              in ApplicationContext a_c,
              in ScopedName resp_iface,
              in ScopedName init_iface,
              in TcUserGenericFactory resp_tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
              CosNaming::NamingContext::CannotProceed,
              CosNaming::NamingContext::InvalidName,
              CosNaming::NamingContext::AlreadyBound);

    void unbind(in TcAddress addr,
              in ApplicationContext a_c,
              in ScopedName resp_iface,
              in ScopedName init_iface,
              in TcUserGenericFactory resp_tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
              CosNaming::NamingContext::CannotProceed,
              CosNaming::NamingContext::InvalidName);

    void rebind(in TcAddress addr,
              in ApplicationContext a_c,
              in ScopedName resp_iface,
              in ScopedName init_iface,
              in TcUserGenericFactory resp_tc_user_factory)
        raises(CosNaming::NamingContext::NotFound,
              CosNaming::NamingContext::CannotProceed,
              CosNaming::NamingContext::InvalidName);

    TcUserGenericFactory resolve(in TcAddress addr,
              in ApplicationContext a_c,
              out ScopedName resp_iface,
              out ScopedName init_iface)
        raises (CosNaming::NamingContext::NotFound,
              CosNaming::NamingContext::CannotProceed,
              CosNaming::NamingContext::InvalidName);

}; // end TcFactoryFinder

interface TcPduProviderFactory; //forward definition

interface GwAdmin{

    readonly attribute TcFactoryFinder tc_user_factory_naming_if;

    readonly attribute TcPduProviderFactory
    tc_pdu_provider_factory_if;
```

```
}; // end GwAdmin

interface TcRepository {

ScopedName get_error_name(in ScopedName iface,
                in IdValue code );

ScopedName get_operation_name(in ScopedName iface,
                in IdValue code );

ScopedName get_extension_name(in ScopedName iface,
                in IdValue code );

IdValue get_id(in ScopedName scoped_name);

ASN1_ExtensionCriticality get_extension_criticality(
                in ScopedName extension_scoped_name);

unsigned long get_operation_timer(in ScopedName
                op_scoped_name );

}; //end TcRepository


interface TcServiceFinder{

    readonly attribute CosNaming::NamingContext gt_root;
    readonly attribute GwAdmin gw_admin_if;
    readonly attribute TcRepository tc_repository_if;

}; //end TcServiceFinder

typedef short ProblemType;
typedef short ProblemCode;
// ProblemType values
const ProblemType GENERAL_PROBLEM = 0;
const ProblemType INVOKE_PROBLEM = 1;
const ProblemType RETURN_RESULT_PROBLEM = 2;
const ProblemType RETURN_ERROR_PROBLEM = 3;
// ProblemCode values
const ProblemCode GP_UNRECOGNIZED_COMPONENT = 0;
const ProblemCode GP_MISTYPED_COMPONENT = 1;
const ProblemCode GP_BAD_STRUCTURED_COMPONENT = 2;
const ProblemCode IP_DUPLICATE_INV_ID = 0;
const ProblemCode IP_UNRECOG_OPERATION = 1;
const ProblemCode IP_MISTYPED_PARAM = 2;
const ProblemCode IP_RESOURCE_LIMIT = 3;
const ProblemCode IP_INIT_RELEASE = 4;
const ProblemCode IP_UNRECOG_LINK_ID = 5;
const ProblemCode IP_LINKED_RESP_EXPECTED = 6;
const ProblemCode IP_UNEXPECTED_LINKED_OP = 7;
```

```
const ProblemCode RRP_UNRECOG_INV_ID = 0;
const ProblemCode RRP_RR_UNEXPECTED = 1;
const ProblemCode RRP_MISTYPED_PARAM = 2;
const ProblemCode REP_UNRECOG_INV_ID = 0;
const ProblemCode REP_RE_UNEXPECTED = 1;
const ProblemCode REP_UNRECOG_ERROR = 2;
const ProblemCode REP_UNEXPECTED_ERROR = 3;
const ProblemCode REP_MISTYPED_PARAM = 4;

typedef unsigned long Timeout;
typedef sequence<octet> Asn1Data;
typedef unsigned short OperationClass;
// OperationClass values
const OperationClass TC_CLASS_1 = 1;
const OperationClass TC_CLASS_2 = 2;
const OperationClass TC_CLASS_3 = 3;
const OperationClass TC_CLASS_4 = 4;

struct RejectProblem{
    ProblemType type;
    ProblemCode code;
};
enum TerminationType{
            PREARRANGED,
            BASIC
};
union DialogPortion switch(boolean) {
        case TRUE: ApplicationContext a_c;
        case FALSE: Asn1Data dialog_info;
};

typedef unsigned long DialogId;
typedef short DialogQos;
// DialogQos values
const DialogQos SCCP_CLASS_0_NO_ERROR = 0;
const DialogQos SCCP_CLASS_0_WITH_ERROR = 1;
const DialogQos SCCP_CLASS_1_NO_ERROR = 2;
const DialogQos SCCP_CLASS_1_WITH_ERROR = 3;
const DialogQos QOS_NOT_SPECIFIED = 3;

exception DialogStillOpen{};
exception InvalidDialogId{};
exception NoMoreDialogs{};
exception LReject {InvokeId ivk_id; RejectProblem problem;};

enum ComponentType{
        INVOKE,
        RESULT_L,
        RESULT_NL,
        U_ERROR,
        U_REJECT,
```

```
            R_REJECT
        };
    struct Invoke{
            OperationClass op_class;
            InvokeId ivk_id;
            InvokeId lnk_id;
            Asn1Data oper;
            Timeout op_timer;
    };
// can use default timer with timeout of 0
// note incoming msgs always have a time of 0

    struct ResultL{
            InvokeId ivk_id;
            Asn1Data result;
    };
    struct ResultNl{
            InvokeId ivk_id;
            Asn1Data result;
    };
    struct UError{
            InvokeId ivk_id;
            Asn1Data error;
    };
    struct UReject{
            InvokeId ivk_id;
            RejectProblem problem;
    };
    struct RReject{
            InvokeId ivk_id;
            RejectProblem problem;
    };
    union Component switch(ComponentType) {
            case INVOKE      :   Invoke i;
            case RESULT_L    :   ResultL r_l;
            case RESULT_NL   :   ResultNl r_nl;
            case U_ERROR     :   UError u_e;
            case U_REJECT    :   UReject u_r;
            case R_REJECT    :   RReject r_r;
    };

    typedef sequence<Component> ComponentList;

    interface TcPduUser; //forward definition

    interface TcPduProvider{

        DialogId get_dialog_id(in TcPduUser user)
                raises (NoMoreDialogs);

        void uni_req(in DialogQos qos,
```

```
                in TcAddress dest,
                in TcAddress orig,
                in DialogId d_id,
                in DialogPortion d_p,
                in ComponentList c_list)
        raises (NoMoreDialogs, LReject);

    void begin_req(in DialogQos qos,
                in TcAddress dest,
                in TcAddress orig,
                in DialogId d_id,
                in DialogPortion d_p,
                in ComponentList c_list)
        raises (NoMoreDialogs, LReject);

    void continue_confirm_req(
                in TcAddress orig,
                in DialogId d_id,
                in DialogPortion d_p,
                in ComponentList c_list)
        raises (InvalidDialogId, LReject);

    void continue_req(in DialogId d_id,
                in DialogPortion d_p,
                in ComponentList c_list)
        raises (InvalidDialogId, LReject);

    void end_req(in DialogId d_id,
                in DialogPortion d_p,
                in TerminationType term,
                in ComponentList c_list)
        raises (InvalidDialogId, LReject);

    void u_abort_req(in DialogId d_id,
                in DialogPortion d_p)
        raises (InvalidDialogId);

    void set_dialog_qos(in DialogId d_id,
                in DialogQos qos)
        raises(InvalidDialogId, InvalidParameter);

    DialogQos get_dialog_qos(in DialogId d_id)
        raises(InvalidDialogId);

    void u_cancel_req(in DialogId d_id,
                in InvokeId ivk_id)
        raises(InvalidDialogId, InvalidParameter);

    void destroy()
        raises(DialogStillOpen);
```

```
}; // end TcPduProvider

interface TcPduUser{

    void uni_ind(in TcPduProvider sender,
                 in DialogQos qos,
                 in TcAddress dest,
                 in TcAddress orig,
                 in DialogId d_id,
                 in DialogPortion d_p,
                 in ComponentList c_list)
        raises (NoMoreDialogs);

    void begin_ind(in TcPduProvider sender,
                 in DialogQos qos,
                 in TcAddress dest,
                 in TcAddress orig,
                 in DialogId d_id,
                 in DialogPortion d_p,
                 in ComponentList c_list)
        raises (NoMoreDialogs);

    void continue_ind(in DialogId d_id,
                 in DialogPortion d_p,
                 in ComponentList c_list)
        raises (InvalidDialogId);

    void end_ind(in DialogId d_id,
                 in DialogPortion d_p,
                 in ComponentList c_list)
        raises (InvalidDialogId);

    void u_abort_ind(in DialogId d_id,
                 in DialogPortion d_p)
        raises (InvalidDialogId);

    void notice_ind(in DialogId d_id,
                 in short report_cause)
        raises (InvalidDialogId);

    void l_cancel_ind(in DialogId d_id,
                 in InvokeId ivk_id)
        raises(InvalidDialogId, InvalidParameter);

    void p_abort_ind(in DialogId d_id,
                 in PAbortReason reason)
        raises (InvalidDialogId);

    DialogQos get_dialog_qos(in DialogId d_id)
        raises(InvalidDialogId);
```

```
            void set_dialog_qos(in DialogId d_id,
                                in DialogQos qos)
                raises(InvalidDialogId, InvalidParameter);
        }; // end TcPduUser

        exception UnknownTcAddress{};
        exception AlreadyBound{};

        interface TcPduUserFactory{

            TcPduUser create_tc_pdu_user(
                    in ApplicationContext application_context)
                raises(NoMoreDialogs);

        }; // end TcPduUserFactory

        interface TcPduProviderFactory{

            TcPduProvider create_tc_pdu_provider(
                    in TcPduUser user,
                    out DialogId d_id)
                raises(NoMoreDialogs);

            void register(in TcAddress dest,
                    in ApplicationContext a_c,
                    in TcPduUserFactory user_factory)
                raises(AlreadyBound);

            void deregister(in TcAddress dest,
                    in ApplicationContext a_c)
                raises(UnknownTcAddress);

    }; // end TcPduProviderFactory
    }; //end TcSignaling

    #endif  // _TC_SIGNALING_
```

B.2 The SIOP Module

```
    //FILE: SIOP.idl
    #ifndef _SIOP_IDL_
    #define _SIOP_IDL_

    // SCCP Inter-ORB Protocol definitions

    #include <IOP.idl>

    #pragma prefix "omg.org"
    module SIOP { // IDL
```

```
typedef unsigned short EndpointId;
struct Version {
    octet major;
    octet minor;
};

const octet ERROR_MESSAGE = 0;
const octet NORMAL_MESSAGE = 1;

struct MessageHeader {
    char magic [4];
    Version SIOP_version;
    octet flags;
    octet message_type;
    EndpointId calling_endpoint;
    EndpointId called_endpoint;
    unsigned long message_size;
};

typedef octet TranslationTypeIndicator;
union TranslationType switch (boolean) {
    case TRUE:
        TranslationTypeIndicator translation_type_indicator;
};
typedef unsigned short NatureOfAddressIndicator;
union NatureOfAddress switch (boolean) {
    case TRUE:
        NatureOfAddressIndicator nature_of_address_indicator;
};
const NatureOfAddressIndicator
        SUBSCRIBER_NUMBER = 1;
const NatureOfAddressIndicator
        NATIONAL_SIGNIFICANT_NUMBER = 3;
const NatureOfAddressIndicator
        INTERNATIONAL_NUMBER = 4;
typedef unsigned short NumberingPlanIndicator;
union NumberingPlan switch (boolean) {
    case TRUE:
        NumberingPlanIndicator numbering_plan_indicator;
};
const NumberingPlanIndicator UNKNOWN = 0;
const NumberingPlanIndicator ISDN_TELEPHONY_E164 = 1;
struct GlobalTitleIndicator {
    string global_title;
    TranslationType translation_type;
    NatureOfAddress nature_of_address;
    NumberingPlan numbering_plan;
};
union GlobalTitle switch (boolean) {
    case TRUE: GlobalTitleIndicator global_title_indicator;
};
```

```
        typedef unsigned short SPCIndicator;
        union SPC switch (boolean) {
            case TRUE: SPCIndicator spc_indicator;
        };
        typedef octet SSNIndicator;
        union SSN switch(boolean) {
            case TRUE: SSNIndicator ssn_indicator;
        };
        struct SCCPAddress {
            GlobalTitle global_title;
            SPC     signalling_point_code;
            SSN     sub_system_number;
        };

        struct ProfileBody {
            Version SIOP_version;
            SCCPAddress address;
            EndpointId endpoint;
            sequence<octet> object_key;
            sequence <IOP::TaggedComponent> components;
        };
        struct ContactInfo {
            Version SIOP_version;
            SCCPAddress address;
            EndpointId endpoint;
        };
};
#endif // _SIOP_IDL_
module IOP { // IDL
    // the numbers XXX must be assigned by OMG
    const ProfileId TAG_SCCP_IOP = XXX;
    const ComponentId TAG_SCCP_CONTACT_INFO = XXX;
};
```

Specification Translation Example C

The following example illustrates how TC-User protocol definitions, specified in ASN.1, may be mapped to OMG IDL using the JIDM Specification Translation rules. ROS/TC-User information object class definitions (or macro definitions) are mapped according to the rules specified in this document. The ASN.1 definitions shown below in IN-CS-1-Operations module and IN-CS-1-datatypes module are taken from the Intelligent Network CS-1 Application Protocol Abstract Syntax specified in ITU-T Recommendation Q.1218 [**8**].

C.1 IN-CS-1-Operations module

```
IN-CS-1-Operations { ccitt recommendation q 1218 modules(0)
cs-1-operations(0) version1(0) }
DEFINITIONS ::=
BEGIN
IMPORTS
…
OriginationAttemptAuthorized ::= OPERATION

    ARGUMENT
    OriginationAttemptAuthorizedArg

    ERRORS {
    MissingCustomerRecord,
    MissingParameter,
    SystemFailure,
    TaskRefused,
    UnexpectedComponentSequence,
    UnexpectedDataValue,
    UnexpectedParameter }
…
END
```

C.2 IN-CS-1-datatypes module

```
IN-CS-1-datatypes { ccitt recommendation q 1218 modules(0)
cs-1-datatypes(0) version1(0) }
DEFINITIONS
BEGIN
IMPORTS
…
OriginationAttemptAuthorizedArg ::= SEQUENCE {

    dpSpecificCommonParameters [0] DpSpecificCommonParame-
ters,
    dialledDigits[1] CalledPartyNumber OPTIONAL,
    callingPartyBusinessGroupID [2] CallingPartyBusinessGroupID
OPTIONAL,
    callingPartySubaddress [3] CallingPartySubaddress
OPTIONAL,
    callingFacilityGroup [4] FacilityGroup OPTIONAL,
    callingFacilityGroupMember  [5] FacilityGroupMember
OPTIONAL,
    travellingClassMark [6] TravellingClassMark OPTIONAL,
    extensions[7] SEQUENCE SIZE(0..MAX) OF ExtensionField
OPTIONAL }
…
END
```

As specified by the JIDM Specification Translation rules, each ASN.1 module maps to an IDL module contained in a separate IDL file. IDL modules and files are named using the “nickname” that has been assigned to the ASN.1 module. IMPORT clauses are mapped as a list of **#include** directives for the appropriate IDL files. The ASN.1 **OPERATION**s, defined in the IN-CS-1-Operations module, are mapped to IDL operation signatures as shown below. Arguments of the **OPERATION**s, defined in the IN-CS-1-datatypes module, are translated according to the JIDM rules. The exception parameters are defined in an included IDL file which is created from the mapping of the IN-CS-1-Errors ASN.1 module.

C.3 Generated IDL Interface for Type I ASN.1 description

```
// IDL Filename : Q1218IN_.idl
// Generated from ASN.1 module :
// IN-CS-1-Operations { ccitt recommendation q 1218 modules(0) cs-1-opera-
tions(0) version1(0) }

#include "ASN1Types.idl"
#include "ASN1Limits.idl"
#include "Q1218IN_1.idl"
#include "Q1218IN_2.idl"
#ifndef _Q1218IN__IDL_
#define _Q1218IN__IDL_
```

```
module Q1218IN_ {
…
… parameters to exceptions are defined in IDL module Q1218IN_2 (IN-CS-1-
Errors)
…
    exception MissingCustomerRecord {DialogFlowCtr d_f_c;};
    exception MissingParameter {DialogFlowCtr d_f_c;};
    exception SystemFailure {
        UnavailableNetworkResourceType unavailableNetworkResource;
        DialogFlowCtr d_f_c;};
    exception TaskRefused {
        TaskRefusedErrorArgType taskRefusedErrorArg;
        DialogFlowCtr d_f_c;};
    exception UnexpectedComponentSequence {
        DialogFlowCtr d_f_c;};
    exception UnexpectedDataValue {DialogFlowCtr d_f_c;};
    exception UnexpectedParameter {DialogFlowCtr d_f_c;};
…
interface DefAc{
…
void OriginationAttemptAuthorized(
    in   OriginationAttemptAuthorizedArgType
        OriginationAttemptAuthorizedArg,
    inout TcContext ctext)
    raises (MissingCustomerRecord,
            MissingParameter,
            SystemFailure,
            TaskRefused,
            UnexpectedComponentSequence,
            UnexpectedDataValue,
            UnexpectedParameter)
    };

};
#endif /* _Q1218IN__IDL_ */
```

C.4 Generated IDL Types

```
// IDL Filename : Q1218IN_1.idl
// Generated form ASN.1 module :
// IN-CS-1-datatypes { ccitt recommendation q 1218 modules(0) cs-1-
datatypes(0) version1(0) }

#include "ASN1Types.idl"
#include "ASN1Limits.idl"
#include "Q1400Ext.idl"

#ifndef _Q1218IN_1_IDL_
#define _Q1218IN_1_IDL_
```

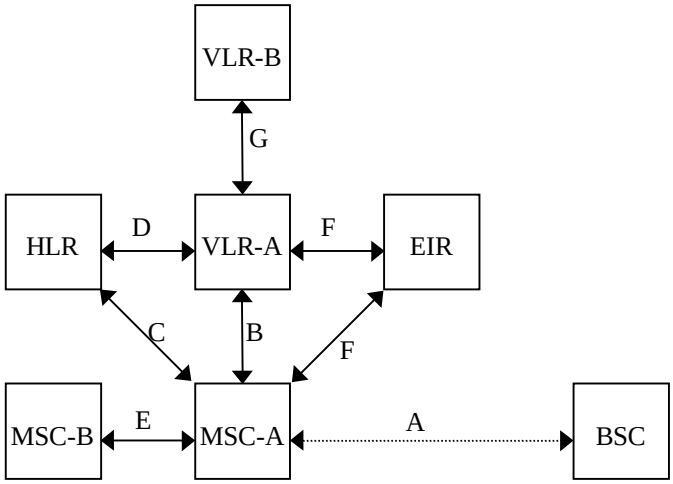
```
module Q1218IN_1 {
…
... nested types in OriginationAttemptAuthorizedArgType defined here ..
…
struct OriginationAttemptAuthorizedArgType
{
    DpSpecificCommonParametersTypedpSpecificCommonParameters;
    CalledPartyNumberTypeOptdialledDigits;
    CallingPartyBusinessGroupIDTypeOpt callingPartyBusinessGroupID;
    CallingPartySubaddressTypeOptcallingPartySubaddress;
    FacilityGroupTypeOptcallingFacilityGroup;
    FacilityGroupMemberTypeOpt callingFacilityGroupMember;
    TravellingClassMarkTypeOpttravellingClassMark;
    ExtensionsTypeOptextensions;
};
...
};
#endif // _Q1218IN__IDL_
```

Applicability to Non-IN Protocols *D*

D.1 Overview

This specification addresses interworking between CORBA and TC-User application entities, which covers a range of applications and is not strictly limited to traditional IN (as defined by the INAP protocol). An important family of protocols also supported is the SS7 Mobile Application Part (MAP) [**12**]. The various MAP specifications (MAP-GSM, IS41, etc.) define signalling traffic for mobile telephony networks. In addition to basic signalling and addressing the special signalling requirements of mobility, MAP supports a number of supplementary services that are similar in many respects to traditional IN services. Integration of the infrastructure for traditional IN services and mobile services is enabled by the approach of this specification.

The elements of the GSM signalling network and the possible signalling interactions between them (as standardized by ETSI) are illustrated below.



BSC: Base Station Controller
EIR: Equipment Identity Register
HLR: Home Location Register
MSC: Mobile Switching Centre
VLR: Visitor Location Register

MAP/SS.7 Signalling Link
Non-MAP Signalling Link

Figure D-1 Entities and Interfaces in a MAP-GSM Signalling Network

The current MAP-GSM specifications [**12**] do not define Application Contexts/Contracts in the MAP definition, so the specification translation approach in this specification for a Type I ASN.1 specification is valid. This would allow the use of the CORBA Naming Service as defined in Section 2.2.3, “Application Location and Association Initiation,” on page 2-28 of this specification.

D.2 Use of MAP-GSM Interfaces

The MAP-GSM specifications [**12**] include the concept of named interfaces between the elements of the signalling network (shown as single letter labels on the signalling paths in the above figure). These interfaces are defined in terms of supported and required operations and may form the basis for OMG IDL interface definitions. This requires a

small extension to the ASN.1 mapping algorithm defined in this specification. As an example, the definition of operations for a MAP HLR (section 6.5.3 in [**12**]) is shown in the table below.

Table D-1 Operations for HLR

Operation	**Interface**	**S/R**
activateSS	D	R
activateTraceMode	D	S
alertServiceCentre	C	S
beginSubscriberActivity	D	R
cancelLocation	D	S
deactivateSS	D	R
deactivateTraceMode	D	S
deleteSubscriberData	D	S
deregisterMobileSubscriber	D	R
eraseSS	D	R
forwardcheckssindication	D	S
forwardSsNotification	D	S
getPassword	D	S
insertSubscriberData	D	S
interrogateSS	D	R
noteMsPresent	D	R
updateLocation	D	R
processUnstructuredSsData	D	R
provideRoamingNumber	D	S
registerChargingInformation	C	R
registerPassword	D	R
registerSS	D	R
reset	D	Both
sendRoutingInfoForSM	C	R
sendRoutingInformation	C	R
setMessageWaitingData	C	R

Unfortunately each of these MAP interfaces is bi-directional (unlike a TC/ROS Application Context). This means that the concepts of AC initiator and responder do not easily map onto a MAP interface. Instead, it is necessary to define two pseudo-ACs replacing the MAP interface. These are defined as the dialog initiation destination entity

name concatenated onto the MAP interface name with a preceding underscore (e.g., HLR_C and MSC_C). Two IDL interfaces may then be generated for the MAP interface, one for the supported operations of each entity in the interaction. Of course, the supported operations of one entity correspond to the required operations of the other entity. These interfaces are named by concatenating the three elements:

1. the MAP entity name (e.g., HLR)
2. the MAP interface name preceded by an underscore (e.g., _C)
3. the string "_supported"

In the CORBA Naming Service the structure for the interfaces thus generated is shown below.

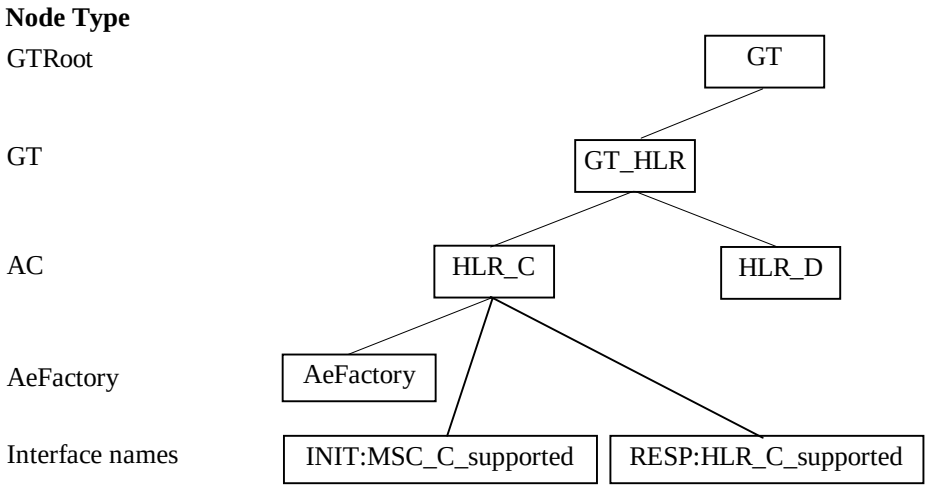


Figure D-2 Structure of MAP information in CORBA Naming Service

This structure is compatible with the structure of TC information defined for the TC'93 case described in Section 2.2.3.2, "Extensions for TC93," on page 2-31 of this specification. However, as no AC information is provided in the signalling APDUs, the gateway implementation must be configured to map particular calling addresses to types of calling entities. For pure CORBA interactions, the initiating CORBA object must know which pseudo-AC it wishes to interact with.

The table of operations for the HLR would map to OMG IDL as follows:

```
interface HLR_D_supported : TcUserGenericFactory {
// corresponds to GSM interface D (HLR side)
…  activateTraceMode( … ) raises ( … );
…  cancelLocation( … ) raises ( … );
…  deactivateTraceMode( … ) raises ( … );
…  deleteSubscriberData( … ) raises ( … );
…  forwardcheckssindication( … ) raises ( … );
…  forwardSsNotification( … ) raises ( … );
```

```
... getPassword( ... ) raises ( ... );
... insertSubscriberData( ... ) raises ( ... );
... provideRoamingNumber( ... ) raises ( ... );
... reset( ... ) raises ( ... );
}; // end interface HLR_D_supported
interface VLR_D_supported : TcUserGenericFactory {
// corresponds to GSM interface D (VLR side)
... activateSS( ... ) raises ( ... );
... beginSubscriberActivity( ... ) raises ( ... );
... deactivateSS( ... ) raises ( ... );
... deregisterMobileSubscriber( ... ) raises ( ... );
... eraseSS( ... ) raises ( ... );
... interrogateSS( ... ) raises ( ... );
... noteMsPresent( ... ) raises ( ... );
... updateLocation( ... ) raises ( ... );
... processUnstructuredSsData( ... ) raises ( ... );
... registerPassword( ... ) raises ( ... );
... reset( ... ) raises ( ... );
}; // end interface VLR_D_supported

interface HLR_C_supported : TcUserGenericFactory {
// corresponds to GSM interface C (HLR side)
... alertServiceCentre ( ... ) raises ( ... );
}; // end interface HLR_C_supported

interface MSC_C_supported : TcUserGenericFactory {
// corresponds to GSM interface C (MSC side)
... registerChargingInformation ( ... ) raises ( ... );
... sendRoutingInfoForSM ( ... ) raises ( ... );
... sendRoutingInformation ( ... ) raises ( ... );
... setMessageWaitingData ( ... ) raises ( ... );
}; // end interface MSC_C_supported
// additional interfaces for other AEs omitted

interface AeFactory : TcUserGenericFactory {
   HLR_D_supported create_HLR_D_supported (
in VLR_D_supported corresponding_iface,
                    in AssociationId a_id,
in boolean explicit_flow_control);
HLR_D_supported create_HLR_D_supported_with_dialogdata (
in VLR_D_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control
in string protocol_version,
in DialogUserData d_u_d));
VLR_D_supported create_VLR_D_supported (
in HLR_D_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control);
```

```
VLR_D_supported create_VLR_D_supported_with_dialogdata (
in HLR_D_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control
in string protocol_version,
in DialogUserData d_u_d));
HLR_C_supported create_HLR_C_supported (
in MSC_C_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control);
HLR_C_supported create_HLR_C_supported_with_dialogdata (
in MSC_C_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control
in string protocol_version,
in DialogUserData d_u_d));
MSC_C_supported create_MSC_C_supported (
in VLR_C_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control);
   MSC_C_supported create_MSC_C_supported_with_dialogdata (
in VLR_C_supported corresponding_iface,
in AssociationId a_id,
in boolean explicit_flow_control
in string protocol_version,
in DialogUserData d_u_d));
// additional create operations for other AEs omitted
}; // end interface AeFactory
```

Ros Definitions E

ITU-T Rec. X.880 | ISO/IEC 13712-1 [**14**] defines a number of concepts and constructs to describe the interaction between objects that follow the ROS request/reply interaction paradigm. Such objects are called ROS-objects, and the basic interaction is specified by the invocation of an operation by one ROS-object (the invoker) and its performance by another (the performer). Performance of an operation may lead to a return from the performer to the invoker of a report of the outcome - a result, to report a successful completion, or an error otherwise. During the performance of an operation, the performer may invoke linked operations to be performed by the invoker (of the original operation).

E.1 Operation and Error

The **OPERATION** information object class (whose syntax is shown below) collects together all the syntactic aspects of what constitutes a remote operation, which is shared by the invoker and performer, and for some aspects the infrastructure through which they communicate. This includes:

- The data type of the value (identified by the keyword **ARGUMENT**) to be conveyed with the requested operation.
- The data type of the result (identified by the keyword **RESULT**), if any, returned upon successful completion of the operation.
- A set of errors (identified by the keyword **ERRORS**), any of which may be returned to signal the unsuccessful performance of the operation.
- Other operations "linked" to this one (identified by the keyword **LINKED**), which may be invoked by the performer before completing the originally invoked operation.
- Whether this operation is invoked synchronously (identified by the keyword **SYNCHRONOUS**).

- A means to identify this operation from others (identified by the keyword **CODE**) that may be invoked by this invoker on the same performer.

The syntax for the **OPERATION** information object class is as follows:

```
OPERATION ::= CLASS {
   &ArgumentType              OPTIONAL,
   &argumentTypeOptional      BOOLEAN OPTIONAL,
   &returnResult              BOOLEAN DEFAULT TRUE,
   &ResultType                OPTIONAL,
   &resultTypeOptional        BOOLEAN OPTIONAL,
   &Errors                    ERROR OPTIONAL,
   &Linked                    OPERATION OPTIONAL,
   &synchronous               BOOLEAN DEFAULT FALSE,
   &alwaysReturns             BOOLEAN DEFAULT TRUE,
   &InvokePriority            Priority OPTIONAL,
   &ResultPriority            Priority OPTIONAL,
   &operationCode             Code UNIQUE OPTIONAL
}
WITH SYNTAX
{
   [ARGUMENT&ArgumentType     [OPTIONAL&argumentTypeOptional]]
   [RESULT&ResultType         [OPTIONAL&resultTypeOptional]]
   [RETURN RESULT             &returnResult]
   [ERRORS                    &Errors]
   [LINKED                    &Linked]
   [SYNCHRONOUS               &synchronous]
   [ALWAYS RESPONDS           &alwaysReturns]
   [INVOKE PRIORITY           &InvokePriority]
   [RESULT-PRIORITY           &ResultPriority]
   [CODE                      &operationCode]
}
```

The equivalent description in the earlier ASN.1 MACRO notation is as follows:

```
OPERATION MACRO ::=
BEGIN
TYPE NOTATION ::= Parameter Result Errors Linked Operations
VALUE NOTATION ::= value(VALUE CHOICE{
   localValue INTEGER,
   globalValue OBJECT IDENTIFIER})
Parameter ::= “PARAMETER” NamedType | empty
Result ::= “RESULT” ResultType | empty
ResultType ::= NamedType | empty
Errors ::= “ERRORS” “{“ErrorNames”}” | empty
LinkedOperations ::= “LINKED” “{“LinkedOperationNames”}” | empty
ErrorNmaes ::= ErrorList | empty
ErrorList ::= Error | ErrorList”,” Error
Error ::= value (ERROR) | type
LinkedOperationNames ::= OperationList | empty
OperationList ::= Operation | OperationList”,”Operation
Operation ::= value (OPERATION) | type
NamedType ::= identifier type | type
END
```

An error is a report of the unsuccessful performance of an operation. The information object class **ERROR**, to which all errors belong, is specified as follows:

```
ERROR ::= CLASS
{
   &ParameterType          OPTIONAL,
   &parameterTypeOptional  BOOLEAN OPTIONAL,
   &ErrorPriority          Priority OPTIONAL,
   &errorCode              Code UNIQUE OPTIONAL
}
WITH SYNTAX
{
   [PARAMETER   &ParameterType [OPTIONAL &parameterTypeOptional]]
   [PRIORITY    &ErrorPriority]
   [CODE        &errorCode]
}
```

The description of errors using the earlier ASN.1 MACRO notation is as follows:

```
ERROR MACRO ::=
BEGIN
TYPE NOTATION ::= Parameter
VALUE NOTATION ::= value(VALUE CHOICE{
   localValue INTEGER,
   globalValue OBJECT IDENTIFIER})
Parameter ::= “PARAMETER” NamedType | empty
NamedType ::= Named type | type
END
```

Note – The **OPERATION** and **ERROR** information object classes are a more precise specification of the features of an operation (and any associated errors) than that provided by the earlier **MACRO** notation with the same names. In fact, the user-defined syntax provided by the **WITH SYNTAX** clause (which must be used for the definition of individual operations) closely mimics the syntax based on the earlier **MACRO** notation. The only changes are minor rearrangements of some items, and the inclusion of additional fields for features, which in the earlier notation were expressed purely as comments, if at all. Note also that TC-Users do not specify the invocation and response priority fields.

E.2 Operation Package (Application Service Element)

The interaction between (pairs of) ROS-objects belonging to some ROS-object-class are defined in terms of sets of related operations, each set being described by an information object class called **OPERATION-PACKAGE**. An instance of the operation package class defines which operations each ROS-object in the pair may invoke the other. Thus, unlike a traditional client-server model, which defines the operations that a client may invoke of the server, the ROS model simultaneously describes both the client and server aspects of a ROS-object.

If both objects can only invoke the same set of operations of the other, then the package is said to be symmetrical. Otherwise, if there is a set of operations that one object can invoke, and a different set that the complementary object can invoke, then the package is said to be asymmetrical. In this case, based on some intuitive judgment of their roles, or arbitrarily, one of these objects is called the consumer (of the operation package) while the other is the supplier.

The **OPERATION-PACKAGE** information object class (whose syntax is provided below) defines, given an assignment of the roles - consumer and supplier - played by a pair of ROS-objects:

- the set of operations, if any, identified by the keyword **OPERATIONS**, which each may invoke of the other,
- the set of operations, if any, identified by the keyword **CONSUMER INVOKES,** which one object, called the consumer, may invoke of the other - the supplier,
- the set of operations, if any, identified by the keyword **SUPPLIER INVOKES**, which the supplier may invoke on the consumer, and
- an identifier, identified by the keyword **ID**, by which this operation package may be distinguished from others that may operate between this pair of ROS-objects

It is important to note that a given ROS-object could be playing the role of a consumer with respect to some operation packages, and that of a supplier with respect to some others.

The syntax of the **OPERATION-PACKAGE** information object class is as follows:

```
OPERATION-PACKAGE ::= CLASS
{
   &Both         OPERATION OPTIONAL,
   &Consumer     OPERATION OPTIONAL,
   &Supplier     OPERATION OPTIONAL,
   &id           OBJECT IDENTIFIER UNIQUE OPTIONAL
}
WITH SYNTAX
{
   [OPERATIONS         &Both]
   [CONSUMER INVOKES&Supplier]
   [SUPPLIER INVOKES   &Consumer]
   [ID                 &id]
}
```

The **OPERATION-PACKAGE** information object class replaces the earlier notation for a collection of operations defined by the **APPLICATION-SERVICE-ELEMENT MACRO**.

Some TC-User specifications (called Type II descriptions in this specification) still use the macro notation to define ASEs. Therefore, for completeness, definition of the **APPLICATION-SERVICE-ELEMENT** macro is provided below:

```
APPLICATION-SERVICE-ELEMENT MACRO ::=
BEGIN
TYPE NOTATION ::= SymmetricAse | ConsumerInvokes SupplierInvokes |
empty
VALUE NOTATION ::= value(VALUE OBJECT IDENTIFIER)
SymmetricAse ::= “OPERATIONS” “{“OperationList”}”
ConsumerInvokes ::= “CONSUMER INVOKES” “{“OperationList”}” | empty
SupplierInvokes ::= “SUPPLIER INVOKES” “{“OperationList”}” | empty
OperationList ::= Operation | OperationList “,” Operation
Operation ::= value(OPERATION)
END
```

Note – The user-defined (and governing) syntax for the **OPERATION-PACKAGE** information object class has been chosen to reproduce the same syntax as that produced by the **APPLICATION-SERVICE-ELEMENT** macro.

E.3 Connection Package

A pair of ROS-objects must have an association between them to serve as a context for the invocation and performance of operations. If the association is dynamically established, one of the ROS-objects plays the role of the initiator (of the association set-up), while the other is the responder. ROS defines a connection package as two special operations, called bind and unbind, that are available as an option to an application designer to dynamically establish and release, respectively, the association between two ROS-objects.

An information object class, **CONNECTION-PACKAGE**, describes the bind and unbind operations used to establish/release the association, whether the responder can unbind and if the attempt to unbind can fail.

Note – TC-Users do not, save in one instance, namely the IN CS2 SCF-SDF interface, make use of the explicit bind and unbind mechanism.

E.4 Contract

In addition to the means by which an association is established between two ROS-objects, the association is governed by an association contract, which is specified in terms of a set of packages that collectively determine the operations that can be invoked during the lifetime of the association.

The association contract, specified by the ASN.1 information object class **CONTRACT** (see the definition below), is therefore the mutual agreement between a pair of ROS-objects on:

- The connection package, if any, identified by the keyword **CONNECTION**, which is used to establish and release the association.
- The operation packages, if any, identified by the keyword **INITIATOR CONSUMER OF**, for which the association initiator assumes the role of the consumer.

- The operation packages, if any, identified by the keyword **INITIATOR SUPPLIER OF**, for which the association initiator assumes the role of supplier.
- The packages, if any, identified by the keyword **OPERATIONS OF**, which are symmetrical or where the initiator may assume either the consumer or the supplier role.
- An identification, identified by the keyword **ID**, of this contract.

The syntax for the **CONTRACT** information object class is as follows:

```
CONTRACT ::= CLASS
{
   &connection             CONNECTION-PACKAGE OPTIONAL,
   &OperationsOf           OPERATION-PACKAGE OPTIONAL,
   &InitiatorConsumerOf    OPERATION-PACKAGE OPTIONAL,
   &InitiatorSupplierOf    OPERATION-PACKAGE OPTIONAL,
   &id                     OBJECT IDENTIFIER UNIQUE OPTIONAL
}
WITH SYNTAX
{
   [CONNECTION                &connection]
   [OPERATIONS OF             &OperationsOf]
   [INITIATOR CONSUMER OF     &InitiatorConsumerOf]
   [RESPONDER CONSUMER OF     &InitiatorSupplierOf]
   [ID                        &id]
}
```

The **CONTRACT** information object class replaces the **APPLICATION-CONTEXT MACRO** defined in an earlier version of the ROS and used in Type II descriptions.

The **APPLICATION-CONTEXT** macro definition is provided below:

```
APPLICATION-CONTEXT MACRO ::=
BEGIN
TYPE NOTATION ::= Symmetric | InitiatorConsumerOf ResponderConsum-
erOf | empty
VALUE NOTATION ::= value(VALUE OBJECT IDENTIFIER)
Symmetric ::= "OPERATIONS OF" "{" ASEList "}"
InitiatorConsumerOf ::= "INITIATOR CONSUMER OF" "{" ASEList "}" |
empty
ResponderConsumerOf ::= "RESPONDER CONSUMER OF" "{" ASEList "}" |
empty
ASEList ::= ASE | ASEList "," ASE
ASE ::= type - - shall reference an APPLICATION-SERVICE-ELEMENT
type.
END
```

Note – The user-defined (and governing) syntax for the **CONTRACT** information object class has been chosen to reproduce the same syntax as that produced by the **APPLICATION-CONTEXT** macro.

Conformance F

F.1 General Conformance Requirements

All implementations claiming conformance to this specification shall:

a) provide a complete implementation of an interface specification (mandatory or otherwise) for which conformance is claimed unless some part of the interface specification is identified as optional.

b) conform to the mappings of ASN.1 to IDL, as specified in XoJIDM ST [4] and extended by this document, where support of a TC-User protocol specified in ASN.1 is also claimed. Note that the GDMO and SMI to IDL mappings also provided in XoJIDM ST are not required for conformance to this specification.

F.1.1 Specific Conformance Requirements

An implementation can claim conformance to this specification at four conformance points named, respectively:

- TC ASN.1 to IDL Complier;
- TC-User Facilities;
- TC/SS7 Stack Interface;
- SIOP;

Conformance to the TC/SS7 Stack interface shall be either in the Provider or User role, or both.

All conformance points are minimal; conformant implementations can always implement additional functionality.

TC ASN.1 to IDL Complier Conformance Point

Implementations claiming conformance shall:

1. correctly perform the mapping of ASN.1 to IDL, as specified in XoJIDM ST [4] and extended by this document. Note that the GDMO and SMI to IDL mappings also provided in XoJIDM ST are not required for conformance to this specification.

TC-User Facilities Conformance Point

Implementations claiming conformance shall:

1. correctly exercise the client behaviour of those interfaces specified in the TcSignaling module required to perform its role as a TC-User, namely:

- **TcSignaling::TcUser**
- **TcSignaling::TcUserGenericFactory**

2. if the TC-User protocol being supported defines one or more TcUser interfaces, correctly implement these interfaces and behaviours supported by the TC-User and correctly exercise the client behaviour of the corresponding interfaces. TC-User objects shall support name resolution of TC-User objects as defined in this document either through a **CosNaming::NamingContext** interface or through a **TcSignalling::TcFactoryFinder** interface. An appropriate implementation of factory object(s) for any supported interfaces shall also be supplied. TC-User objects shall correctly exercise the client behaviour of the interfaces **TcSignalling::GwAdmin**, **TcSignalling::TcRepository**, **TcSignalling::ServiceFinder**, if required

3. correctly implement the interfaces and behaviours supported by **TcSignalling::GwAdmin**, **TcSignalling::TcRepository**, **TcSignalling::ServiceFinder**, if required

TC/SS7 Stack Interface Conformance Point

(a) Provider Role

Implementations claiming conformance in the Provider role shall:

1. correctly exercise the client behaviour of those interfaces specified in the **TcSignaling** module required to perform its role as a Provider, namely:

- **TcSignaling::TcPduUser**

2. correctly implement the interface and behaviours specified for the **TcSignalling::TcPduProvider** object

3. correctly implement the interface and behaviours specified for the **TcSignalling::TcPduProviderFactory** object

(b) User Role

Implementations claiming conformance in the User role shall:

1. correctly exercise the client behaviour of those interfaces specified in the TcSignaling module required to perform its role as a User, namely:

- **TcSignaling::TcPduProvider**

- **TcSignaling::TcPduProviderFactory**

2. correctly implement the interface and behaviours specified for the **TcSignalling::TcPduUser** object

SIOP Conformance Point

Implementations claiming conformance shall:

1. use the SCCP Inter-ORB Protocol (SIOP) for inter-ORB communication over the Signaling System No. 7 SCCP class 1 protocol.