

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

– REMOTE OPERATIONS –
CONCEPTS, NOTATION
AND CONNECTION-ORIENTED
MAPPINGS

TR/31

December 1985

Free copies of this document are available from ECMA,
European Computer Manufacturers Association
114 Rue du Rhône – 1204 Geneva (Switzerland)

ECMA

EUROPEAN COMPUTER MANUFACTURERS ASSOCIATION

– REMOTE OPERATIONS –
CONCEPTS, NOTATION
AND CONNECTION-ORIENTED
MAPPINGS

TR/31

December 1985

Printed by the Government Printer
at the Government Printing Office,
Wellington, New Zealand.
1951

Brief History

CCITT Study Group VII has done valuable work on the formal specification of interactive application protocols during the development of their Message Handling Systems (MHS) Recommendations.

In CCITT Rec. X.409 and X.410 there is a powerful notation for specifying some external interactions of distributed applications (the "remote operations macro"). From this the complete structure and transfer syntax of operation protocol data units ("OPDUs") can be automatically generated, together with the main elements of protocol procedure.

The ECMA view is that this methodology is likely to be a key factor in the overall success of OSI standardisation. The purpose of this ECMA Technical Report is to progress and package the methodology so that it can be more generally used.

This ECMA Technical Report is one of a set of standards and technical reports for Open Systems Interconnection. Open Systems Interconnection standards are intended to facilitate homogeneous interconnection between heterogeneous information processing systems. This Technical Report is within the framework for the coordination of standards for Open Systems Interconnection which is defined by ISO 7498.

This ECMA Technical Report is based on the practical experience of ECMA member companies world-wide, and on the results of their active participation in the current work of ISO, CCITT and national standard bodies in Europe and the USA. It represents a pragmatic and widely based consensus.

A particular emphasis of this Technical Report is to specify the homogeneous externally visible and verifiable characteristics needed for interconnection compatibility, while avoiding unnecessary constraints upon and changes to the heterogeneous internal design and implementation of the information processing systems to be interconnected.

In the interest of a rapid and effective standardization, this Technical Report is oriented towards urgent and well understood needs. It is intended to be capable of modular extension to cover future developments in technology and needs.

Adopted by the General Assembly of ECMA as ECMA TR/31 on December 12, 1985.

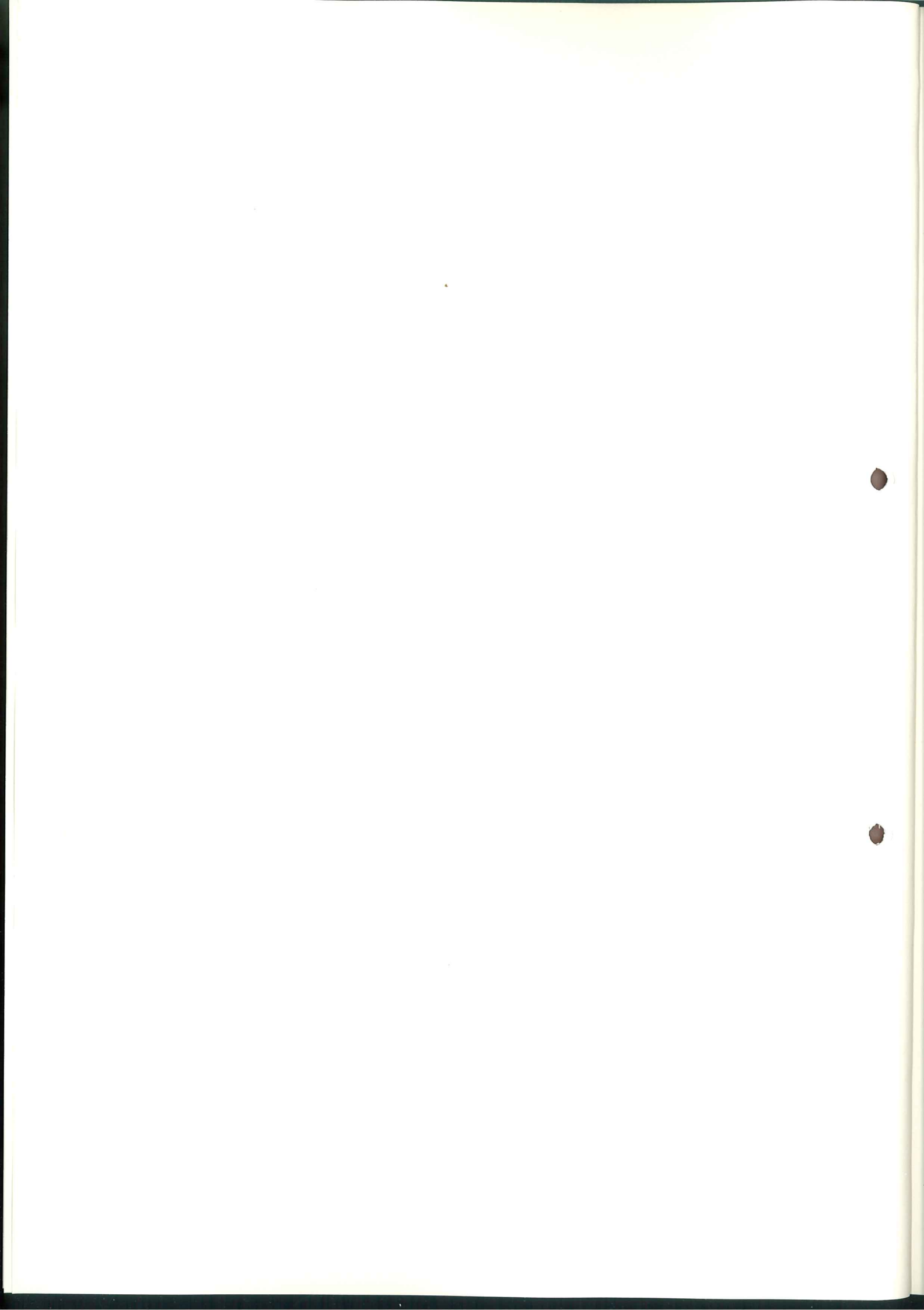


TABLE OF CONTENTS

	<u>Page</u>
1. GENERAL	1
1.1 Scope	1
1.2 Rationale	1
1.3 References	2
1.4 Definitions	3
1.4.1 General Terminology	3
1.4.2 Specific Terminology	3
1.4.3 Acronyms	5
2. REQUIREMENTS	6
3. CONCEPTS	7
3.1 General	7
3.2 Abstract Data Types	7
3.3 Object Model	8
3.4 Operation Structure	8
3.5 Reliability	9
3.6 Execution	10
3.7 Naming	13
3.8 Bindings and Context	13
3.9 Remote Operations	13
3.10 Summary	14
4. NOTATION	17
4.1 General	17
4.2 Notation for Remote Operations	17
4.2.1 Type of Operations	17
4.2.2 Definitions of Macros and Data Types	17
4.3 Notation Clarifications	18
4.4 Using Remote Operation Macros	20
4.4.1 Templates for Ordinary-Operations and Errors	20
4.4.2 Templates for Bind- and Unbind-Operations	22
5. SERVICE AND PROTOCOL	25
5.1 CO-ROS Service Overview	25
5.1.1 Operation Modes	26
5.1.2 Types of Operations	26
5.1.3 Operation Classes	26
5.1.4 ROS-association Classes	26
5.2 CO-ROS: Service Primitives	27
5.2.1 Overview and Description	27
5.3 CO-ROS Service Schematic Representation	37
5.3.1 Introduction	37
5.3.2 Relation between ROS Service Primitives, OPDUs and Error and Reject Situations	37
5.4 Remote Operation Protocol	41
5.5 Mapping of Macros onto CO-ROS	43

TABLE OF CONTENTS (cont'd)

	<u>Page</u>
5.5.1 Establishing and Releasing ROS-associations	43
5.5.2 Mapping of Macros onto ROS Primitives	43
6. MAPPINGS ONTO UNDERLYING SERVICES	47
6.1 General	47
6.2 Mapping CO-ROS onto Reliable Transfer Server	47
6.2.1 Introduction	47
6.2.2 Creating and Releasing ROS-association	47
6.2.3 Transferring OPDUs	48
6.2.4 Managing the Turn	48
6.2.5 Action on RT-EXCEPTION.indication	48
6.3 Mapping CO-ROS onto ISO 8326 BCS Session Services	49
6.3.1 Introduction and Scope	49
6.3.2 Session Connection Establishment Phase	49
6.3.3 Data Transfer Phase - Action of Sending ROS Entity	51
6.3.4 Data Transfer Phase - Actions of Receiving ROS Entity	51
6.3.5 Session Connection Termination Phase	52
7. GUIDELINES FOR REFERENCING THIS TECHNICAL REPORT	55
APPENDIX A - Bibliography	57

1. GENERAL

1.1 Scope

This ECMA Technical Report describes a methodology for specification of external interactions of distributed applications. It contains:

- A statement of the requirements to be satisfied by this specification methodology (see section 2.);
- A tutorial explanation of the fundamental concepts (see section 3.);
- Definition of the specification notation (see section 4.);
- Definition of the connection-oriented service, protocol and mapping (see sections 5 and 6.);
- Recommendations for conformance statements to be included by applications which use this methodology (see section 7.).

The methodology may be the basis for a future ECMA standard on this subject. But before that, there is a need for further study and liaison with ISO and the CCITT.

1.2 Rationale

1.2.1 ECMA Experience using Remote Operations

Several draft ECMA standards have used the remote operations structure defined in CCITT Rec. X.410. It has generally proved to be highly satisfactory, but there have been some difficulties. There are two main problems to resolve: details of how to use remote-operations (particularly bindings) and how to map remote operations to different underlying services.

Some ad hoc solutions have been used. They include a set of detailed services and protocol layer mappings. These are constrained to be compatible with the existing CCITT Recommendations, with consequent difficulties in achieving sufficient generality. The mappings are also affected by the current instability in the ISO upper layer architecture and by some ambiguities concerning the X.410 Reliable Transfer Server.

This ECMA work results from the consequent need to provide an agreed, stable and common base for future ECMA work on distributed applications that use remote operations. This should also be applicable to the work of ISO and the CCITT.

1.2.2 Refinement of the problem

To avoid the underlying instability and variability, the problem has been reformulated at a more abstract level. This emphasises the characteristics of the specification constructs, and de-couples them from layered service mapping issues.

Remote operations are presented here as a methodology for specifying the external interaction of distributed-applications. The stable base proposed for these applications is a "language" and not high-level service-primitives. This approach stabilises the application designer's viewpoint, de-coupling it from the problems of mapping onto the various upper layer services. These underlying mappings are defined independently of the "language".

1.2.3 Productivity

Productivity is one of the main purposes of the methodology defined in this TR.

At the present early stage of OSI evolution, there are few OSI distributed-applications. As OSI matures over the next several years there should be hundreds and then thousands of distributed-applications using OSI. Many thousands of people who are not OSI experts must acquire the ability to produce new distributed-applications. Each distributed-application must also be produced within cost and timescale limits.

This points to the need for a highly productive methodology for specifying the interactive structure of distributed-applications. Productive in the sense that it can be easily learned, and in the sense that it can be efficiently used.

The Macro Notation in CCITT Rec. X.409 and the Operation Macro in X.410 are the natural starting point for achieving this productivity.

1.2.4 Future Extensions

Only basic connection-oriented communications services have been considered in this Technical Report. This satisfies the current most urgent need.

Use of other styles of communication to support Remote Operations is for urgent further study, and may include the following:

- Asynchronous communication via store-and forward messaging services (eg ECMA-93)
- Connection-oriented transaction-processing communication services (eg ECMA TR/29).
- Packet Exchange communication services (eg ECMA TR/29) and Remote Procedure Call services.
- Datagram communication services (eg ECMA TR/29).
- Multi-endpoint communication of all the above kinds.
- Integration of CCR provisions (Commitment, Concurrency and Recovery) with use of Remote Operations in the various styles above.
- Use of non-OSI communications, particularly for gatewaying Remote Operations between OSI networking and proprietary networking.

More comprehensive future services are likely to have different structures which would include the Connection-oriented Remote Operations Service (CO-ROS) defined in this Technical Report. The current protocol mappings for connection-oriented (see clauses 6.2 and 6.3), are unlikely to be changed in future Standards or Technical Reports.

1.3 References

ECMA-93	MIDA - Distributed Application for Message Interchange.
ECMA TR/29	OSI - Distributed Interactive Processing Environment (DIPE).
ISO 7498	Data-communications: Open-systems-interconnection: Basic Reference Model.

ISO 8326	Information Processing Systems, Open Systems Interconnection, Basic Connection Oriented Session Service Definition.
ISO DIS 8824	Specification of Abstract Syntax Notation One (ASN.1).
ISO DIS 8825	Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1).
CCITT Rec. X.409	Message Handling Systems: Presentation Transfer Syntax and Notation.
CCITT Rec. X.410	Message Handling Systems: Remote Operations and Reliable Transfer Server.

For other references see the Bibliography in Appendix A.

1.4 Definitions

1.4.1 General Terminology

The following terms are used with the meanings defined in ISO 7498:

Open systems interconnection. Service, layer service. Service-user, service-provider. Protocol, layer protocol.

1.4.2 Specific Terminology

The following terms are used with the meaning defined below.

1.4.2.1 Terms introduced in section 3:

- abstract data type: (1) a concept for classifying and structuring data and the manipulation of it; (2) the particular classification applied to the data referred to; (3) data viewed abstractly in terms of its classification.
- abstract type; type: (1) abbreviation for abstract data type, or object type; (2) a generalisation of those concepts.
- operation: an action which manipulates or interrogates a type.
- type operation: an operation specific to the type to which it refers.
- type specification: the specification of a type.
- instance of a type; type instance; instance: an entity which conforms with some particular type specification.
- type name: the identifier value used to refer to the type concerned.
- instance name: the identifier value used to refer to the instance concerned.
- type safe: only directly accessible via the defined type operations of its type.
- strongly typed: enforces type safe behaviour for all type operations.
- object model; typed object model; object oriented architecture: the extension of the concept of abstract type applied to the structuring of programmed systems.

- typed object; object: an entity whose behaviour is specified in terms of the object model.
- object type: the particular classification applied to the typed object referred to.
- typed object instance: a typed object which conforms with some particular type specification.
- type inheritance: the concept of new types being constructed from existing types with which they have some behaviour in common.
- remote operations: operations of the generic kind defined in CCITT Rec. X.410.
- normal outcome (of an operation) : the desired correct behaviour achieved by the operation.
- exception outcome (of an operation); exception: some behaviour other than the desired correct behaviour to be achieved by the operation.
- a total operation: an operation, the possible outcomes of which are completely defined by one normal outcome and a set of one or more exception outcomes, the occurrence of the outcomes all being mutually exclusive.
- a robust type: a type, every operation on which is a total operation.
- execution environment abstract type: an abstract type which models the execution characteristics of operations invoked on typed objects.
- execution exceptions: the exception outcomes of operations of the execution environment abstract type.
- binding: a relationship between two objects.
- execution binding: a binding resulting from an operation of the execution environment abstract type.
- object binding: a binding between objects which results from an operation invoked by one object on the other(s).
- stateless server; context free server: object behaviour in which the object in question does not need to preserve state resulting from previous operations.

1.4.2.2 Terms introduced in section 4:

- macro: a formal definition using the macro notation.
- macro notation: a formal specification technique defined in CCITT Rec. X.409 and ISO DIS 8824.
- mapping: the relationship between a set of protocol data units and the underlying service primitives.
- remote errors; errors: exception messages of the kind defined in CCITT Rec. X.410.
- result: a message that reports successful completion of a remote operation and transfers some data, defined in CCITT Rec. X.410.

- data type: in sections 4, 5 and 6 of this Technical Report it usually refers to the particular types defined in CCITT Rec. X.410.

1.4.2.3 Terms introduced in section 5:

- association: a kind of execution binding.
- ROS-association: an execution binding between ROS-entities.
- ROS-entity: the abstraction of the implementation of the ROS service provider in an open system.
- ROS-user: an application entity which interacts with the local ROS-entity for the purpose of interacting with a remotely located ROS-user via the ROS protocol.
- Remote Operation protocol; RO protocol: the protocol defined in this Technical Report.
- parameter: data that accompanies, qualifies or further specifies an operation.
- ROS-association class: a classification defined in clause 5.1.4.
- Operation class: a classification defined in clause 5.1.3.
- synchronous mode: defined in clause 5.1.1.
- asynchronous mode: defined in clause 5.1.1.
- turn: the right to send data in two-way-alternate (twa) communication.
- dialogue mode: defined in ISO 8326.

1.4.2.4 Terms introduced in section 6:

- Reliable Transfer Server: See CCITT Rec. X.410.

1.4.3 Acronyms

- AE: Application Entity, defined in ISO 7498.
- OPDU: Operation Protocol Data Unit.
- OSDU: Operation Service Data Unit.
- ROS: Remote Operation Service.
- RTS: Reliable Transfer Server.

2 . REQUIREMENTS

The requirements to be satisfied by this Technical Report are as follows:

Stability

The Technical Report shall provide a stable base for future use by ECMA distributed applications which use the Remote Operations notation to define their interactions.

ISO & CCITT

Compatibility is required with existing CCITT work on Remote Operations. The proposals in the Technical Report shall be a basis for constructive ECMA contributions to ISO and to CCITT.

Upper layer mappings

The TR shall de-couple distributed application specification from the differences between, and possible instabilities of, the various OSI "execution environments" likely to be used (RTS, CASE, etc.). It should also help to reconcile their differences.

Portability

The specification methodology shall support a high level of abstraction, such that the specifications of distributed application interactions are highly portable.

Precision

The specification methodology shall be such that the specifications of distributed application interactions are likely to be unambiguous.

Resilience

The specification methodology shall be such that application designers can specify highly resilient distributed applications interactions, with comprehensive error management.

Usability

The specification methodology shall be such that application designers without previous experience of it can quickly achieve proficiency.

Productivity

The specification methodology shall be such that application designers can quickly specify, develop and deliver the interactive component of distributed applications.

Implementability

The specification methodology shall be such that complete and compatible implementations (including layer mappings) can readily be derived for the interactive components of distributed applications.

3. CONCEPTS

3.1 General

This section explains the concepts of Remote Operations and the Object Model of which they are a component. It starts with the computer language concept of "abstract data types" which is the foundation. Other concepts are added in several stages, until finally the concepts of "remote operations" are complete. This demonstrates that the Remote Operation structure is firmly based in advanced system design concepts and computer language concepts.

The terminology is informally explained in the text. The exact definitions are in clause 1.4.2.

The text is intended for readers who are not experts in this subject. Those who are already familiar with the concepts should go direct to the later sections, and use this section as reference material.

3.2 Abstract Data Types

The generic concept of abstract data types is simple but very powerful. It is very abstract (and has a mathematical basis), but it is practical and easy to use, and has wide applicability. The concept is explained below in several stages:

- Data structures are classified according to their characteristic behaviour. Each class is referred to as a type (abbreviation for abstract data type).
- This classification is exclusively in terms of the externally visible behaviour of the type.
- This externally visible behaviour of a type is analysed into a set of unitary actions, each of which is distinct and logically complete. These are called operations.
- The operations are specified at a level of abstraction which define what happens, not how it happens. The data syntax of the type is defined abstractly, ie. independent of the concrete data representation used in implementations. The abstract data syntax notation is typically machine processable and machine checkable, and may have a formal mathematical basis.
- The set of all behaviours of a type is completely defined by the set of all operations of the type. These are its type operations.
- The specification of an abstract data type is referred to as its type specification.
- All instances of a type have the same type operations.
- A type is referenced by its type name.
- An instance is referenced specifically by an instance name, and generically by its type name.

The most familiar application of these concepts is the abstract data types of computer languages. The CCITT Rec. X.409 data types are another example of the use of this concept.

Computer languages have built-in support for their primitive types such as "integer", "boolean", "octet-string", etc. For example, the operations on type "integer" are "add", "subtract", etc.

The use of an instance of a type is said to be type safe if the instance can be directly accessed only via the defined type operations of its type. Languages and/or execution environments which enforce type safety over all operations are referred to as being strongly typed.

3.3 Object Model

The next step is to extend use of the concept of abstract types to model and specify the behaviour of programmed systems. When rigorously applied, this is referred to variously as the object model, or typed object model, or object oriented architecture. The scope is completely general, as illustrated by the following range of examples of possible object types: integers, arrays, queues, documents, bank accounts and procedures. But the concept is only applicable where type safe execution is (deemed to be) enforced.

Procedural characteristics are emphasised in the way in which the object model uses type concepts. The operations and the behaviour associated with objects are the main focus of interest, not the data structure. In implementation terms, an object consists of executable and storable procedures, bound with their data, and interacting with other objects via type safe use of the appropriate type operations.

Recursion is an important characteristic. Objects are data structures bound with procedures which themselves are data structures. Operations are manifested as data structures which are exchanged between objects. Hence everything reduces to data structures, which are inherently recursive, hence total and mutual recursiveness of all the concepts.

A powerful characteristic of these abstractions is the constructive specification of new constructive types from existing types. There is an important difference between the way object types combine and abstract data types aggregate. With abstract data types, aggregates (eg. a data record) may be built from other types (eg. integers and strings), but the operations generally remain those of the individual constituent types plus some basic operations on the aggregate data structure as a whole (eg. copy, delete, move). The internal structure of the aggregate remains visible.

With the more powerful object model, types can be combined to form a new type which maintains new invariants between its constituent parts and has its own set of type operations (typically including some of those from the previous types). Furthermore, the construction characteristics are not directly visible. This combining is related to the concept of inheritance which has an important role in the object model.

An essential characteristic of the object model is that the interface to an object type (the type operations) is separate from the internal implementation and actual concrete data structure of the object instance. This systematic hiding of implementation detail is of fundamental importance to the systems structure.

Some newer programming languages use the object model concepts and enforce the type safe behaviour. This capability is rare at present, but is likely to be widespread within 10 years. An example is the experimental language ARGUS (enhanced CLU) in which applications can be written and compiled for distributed execution. See reference LISKOV 82. Another example is the Smalltalk language, see reference GOLDBERG.

3.4 Operation Structure

In the object model, the generic structure of an operation (whether "remote" or "local") is an elementary request/reply interaction (sometimes without the reply). Although the interaction structure is simple and regular, the syntactic and semantic content may be arbitrarily complex.

The request element takes the form of a data structure which is here referred to as the Invoke. Its structure is:

<invoke> ::= <operation type> <arguments>

The <operation type> distinguishes between the different type operations of the object type. The <arguments> are the parameter data types whose values are supplied by the invoker.

The reply element is another data structure, which is here referred to as the Return. The normal return is here referred to as ReturnResults. Its structure is:

<ReturnResults> ::= <results>

The <results> are the parameter data types whose values are supplied by the object whose operation was invoked.

The operation exception return is here referred to as the ReturnError. Its structure is:

<ReturnError> ::= <error type> <error parameter>

The <error type> identifies the kind of exception which has occurred (see clause 3.5) and the <error parameter> is for any supporting diagnostic information supplied by the object whose operation was invoked, as defined for that <error type>.

The above structure (but not necessarily the above terminology) is generic to all formulations of the object model. The particular abstract syntax depends on the choice of specification language, etc.

3.5 Reliability

The systematic way in which the control structure of an operation can include a normal path <ReturnResults> and one or more exception paths <ReturnErrors> is an important contribution to reliability.

The clear separation of normal and exception outcomes is a natural and familiar way of simplifying systems structure and program structure. It helps to ensure that error management is well specified and comprehensively implemented. This style of exception handling is also an important characteristic of some modern programming languages such as CLU and Ada.

A formal methodology to achieve fault-tolerant design with this kind of structure is defined in reference CRISTIAN 82. It is summarized as follows:

The basic concept is that of a total operation. An operation is "total" if it includes predefined outcomes for all possible cases, including error cases. If every operation on type is a total operation, then the type is robust.

The outcome each such operation is either the predefined normal outcome (ie the operation achieved its objective), or a predefined exception outcome (ie the operation completed without achieving its objective, due to some specific kind of error in data values, wrong state, unavailable resources, etc). There is only one normal outcome, but there may be several different exception outcomes. All outcomes of an operation are mutually exclusive, and each is reported via a separate return.

By definition, the set of all outcomes for all operations specifies all possible behaviours of the type. Therefore, since the coverage of their outcomes is "total" (eg cover all error cases), the type is inherently robust (ie all its behaviours terminate in a controlled way for all cases).

Furthermore, the partitioning of its behaviour into separate normal and exception domains (arbitrarily many) simplifies the design and allows each separate subdomain of its behaviour to be analysed separately.

Proveably correct and robust designs may be constructed when the appropriate formalisms are used rigorously, see reference CRISTIAN 84.

A further consideration is that remote execution may involve some uncertainty about the occurrence of remote events, particularly when failures occur. Remote execution reliability characteristics may be classified as follows:

- Exactly once. It is guaranteed that for each invocation the remote execution does occur, and that it occurs once only. This is the strongest guarantee.
- At least once. It is guaranteed that for each invocation the remote execution does occur, but possibly more than once.
- At most once. It is not guaranteed for each invocation that the remote execution occurs; but if it does occur it is guaranteed to occur only once.

These guarantees exclude partial execution.

It may be technically difficult for the underlying execution environment to provide the strong guarantees when there are failures (this is a classical problem of fault-tolerant system design). The stronger guarantees may also involve unacceptable implementation complexity and performance overheads.

The normal engineering trade-offs are to guarantee "exactly once" or "at least once" semantics only when there is end-to-end confirmation of the remote execution.

For some operations on an object type, the weaker "at least once" semantics may be appropriate. Eg an operation which reads remote information (but does not change it) may be repeated without harm; the effects are essentially the same. This kind of operation is termed "idempotent". In such cases, there may be simplifications and performance gains if these less strong guarantees are specified.

In all cases the required guarantees should be carefully chosen.

These error management concepts also apply to the execution environment which is now described.

3.6 Execution

Operations on typed objects are executed in some kind of execution environment, as illustrated in Figure 1.

The execution environment is responsible for:

- enforcing type safe execution of the type operations (by compile time and/or run time checks).
- handling the naming and addressing which identifies and selects the object instances (a type operation does not generally include naming of the object instance on which it operates).
- the scheduling and providing of the execution resources (processor cycles, memory space, communication channels, etc.).

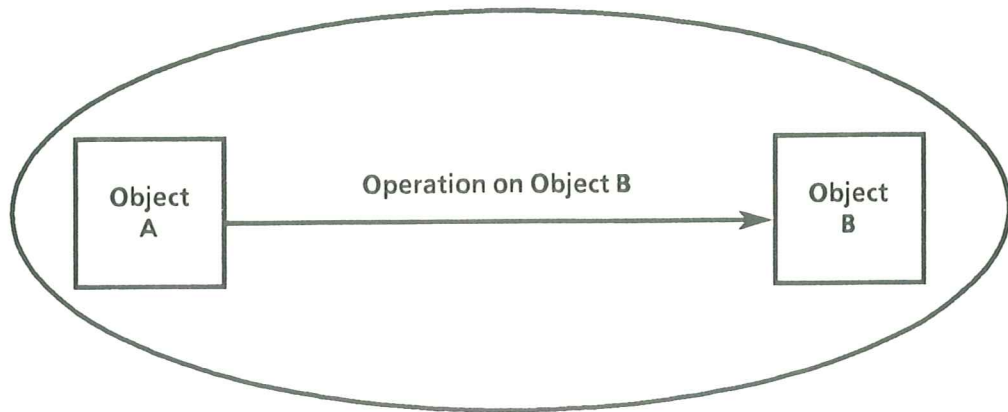


Figure 1. Execution Environment

- generating and communicating any execution exceptions (see clause 3.5 above) which occur from (attempted) execution of type operations.

A general classification of these execution exceptions is as follows:

- access control exceptions (eg. access not allowed);
- binding exceptions (eg. incorrect name);
- availability exceptions (eg. object broken, or busy);
- program execution exceptions (eg. fail stop, or time out);
- access method exceptions (eg. misuse or failure of the execution environment).

These execution exceptions are not inherent in the object type, and are dependent on the run time environment used. Therefore detailed execution exceptions should not be included in the type specification of the object; to do so would damage generality and modularity (separation of concerns).

The recommended way of modelling the relationships between objects and their execution environment is illustrated in Figure 2.

The execution environment is modelled as another abstract type (C), interposed between the ultimate objects A and B. The data structure which is the operation on B is carried transparently in the parameters of the operation on C (which "inherits" the behaviour of B).

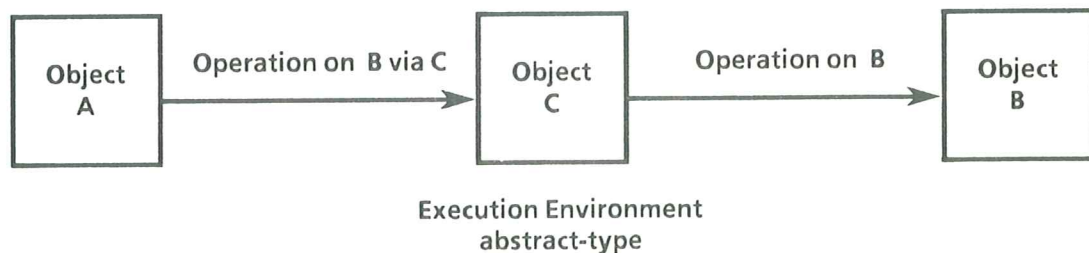


Figure 2. Abstract Execution Environment

The abstract type C varies according to what technique is used to implement the execution environment. Some examples of different techniques are:

- Remote operations based on queued transfer (eg. CCITT Rec. X.410 RTS);
- Synchronous local procedure call;
- Synchronous remote procedure call;
- Asynchronous message passing;
- Etc.

The structure in Figure 2 may be redrawn as in Figure 3, in the style of the OSI Layered Model. The two structures are equivalent.

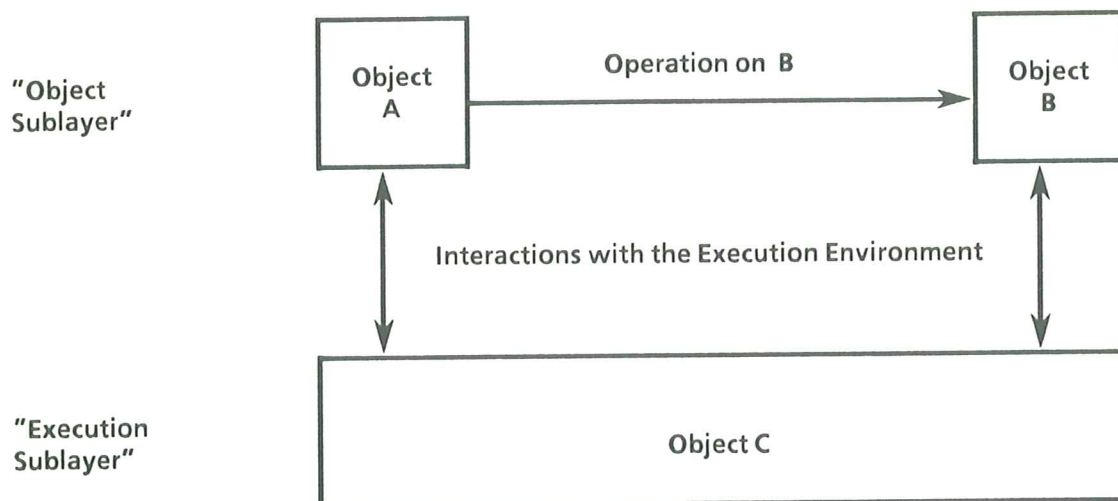


Figure 3. Layered View of Objects and Their Execution Environment

The "Object Sublayer" uses the macro notation (introduced in section 4). The interactions with the "Execution Sublayer" are generally described by using the OSI service primitives notation (eg. see clause 5.2).

An important point to note is that the procedure call mechanism is not inherent in type operations. Neither is the concept of "remote procedure call" synonymous with "remote operations". Eg. in CCITT Rec. X.410, Chapter 3, there are multiple asynchronous remote operations, and no concept of remote procedure calls, and no visibility of the local interfaces which might have procedure call structure. But, the (remote) procedure call is often an appropriate mechanism and fits well with (remote) operations structure.

Any execution environment only supports the execution of operations. Execution is inherently transient.

The persistence of object instances and the persistence of the results of operations on them is a property of the object type(s) concerned. This persistence (or absence of persistence) is defined in the type specification of the object type, and is not an inherent property of the execution environment.

3.7 Naming

The object type name defines the scope of its type operations.

Within this scope, the operation types are distinguished by operation name, and by an operation number which is a compact encoding of the operation name.

The above type name and operation name/number are specified in the type specification of the object type.

The naming and selection of an object instance are outside the scope of the object's type specification. They are matters for whatever execution environment is used. They are not generally visible in the type operations of the object type.

3.8 Bindings and Context

During the execution of an operation there is a relationship between the object instances which "binds" the objects together.

This binding may be only for the duration of that one operation occurrence, or may span multiple operations occurring between the object instances.

The bindings can be classified into two general kinds:

- Execution bindings: These are concerned with the allocation, use, and deallocation of execution resources. They are a matter for the execution environment (see 3.6) and should not be directly visible in the type operations of the object. The details of these bindings may vary in different execution environments.
- Object bindings: These are concerned with semantics of the relationship between the objects. They should be part of the type specification because they affect the behaviour of the type.

The primary interest here is the object bindings.

If the duration of the object binding is only for one operation, then the related concept of a "stateless server" may be useful. This is a characteristic of types in which operations do not depend on the destination object type preserving partial results of previous operations (if needed, they are preserved by the object which is the source of the operation, and are exchanged as parameters of operations).

The concept of object bindings may be viewed as a particular case of the more general concept of "context sensitivity". In this concept the "context" is the type characteristics and data content of the object instance on which the operation is performed, and any other information which is known to both object instances and which affects the behaviour resulting from their type operations. Some examples of context information to which object behaviour might be sensitive are: date, time, place, mutual authentication, object bindings, persistent results remembered from previous encounters, application status information, etc.

A general mechanism for supporting all kinds of context sensitive behaviour is the exchange of "handles" in type operations. A handle is a data value which references some previously identified information. Typically it is an integer with a value which references some local data structure which is implementation dependent.

3.9 Remote Operations

In principle, there is no architectural distinction between "remote" and "local" operations, as already explained. When an operation is "remote", the execution environment is necessarily

one which supports some appropriate kind of remote execution. Indeed, the "local" and "remote" distinction is essentially the choice of execution environment.

But in practice, the term remote operation is restricted to be used for the particular kind of type operation structure defined in CCITT Rec. X.410 section 2. See section 4 below.

This remote operation structure is standardised for distributed use (invoker and invoked objects are in separate computers). But it is also technically suitable for "local" use (invoker and invoked object are co located).

3.10 Summary

The object model is ideally suited to distributed processing.

- Objects and their type operations have a logical structure in which there is no notion of shared data representation or shared state between objects. Therefore, the instances of object types may be loaded into separate computers, between which the operations are communicated.
- The abstract syntax in which operations are defined is logically independent of the internal data representation of the computers and their language systems. An agreed concrete syntax is used to encode the operations, and appropriate conversion is applied at one or both ends (or not at all if the syntax is native to both).
- Communication between objects is decomposed into type operations which are interactions with a simple regular structure (see clause 3.4). Complex interactions are constructed by using these simple elements.
- Type operations can be specified declaratively as a data structure from which the interactive structure can be automatically derived. The specification is thus non procedural and relatively simple.
- The inherent enforcement of type safe behaviour is essential for orderly communication.
- The structure of a type operation can be the same whether it is executed between instances of object types in the same computer or in separate computers. This gives vital design freedom to (re-)configure distributed applications, etc.

The last point emphasises that the object model is not just for distributed processing.

In the field of distributed systems and distributed applications specification, the object model concepts are embodied in remote operations, which are the subject of this Technical Report.

The simple declarative structure of remote operations is highly beneficial as a means of achieving correct and unambiguous specifications of the interactions across distributed systems.

The machine processable syntax notations used to specify (remote) operations, and the general computer language orientation of the concepts have important implications. They open up possibilities of software tools being able to generate executable code for the protocol from the operation specifications and to link this to the user's application code and the (remote) execution infrastructure. This promises greatly simplified design, development, testing and maintenance of distributed applications. Reference BIRRELL 84 describes a remote procedure call system with interactions equivalent to remote operations and a similar kind of machine processable syntax notation. A high degree of automatic code generation is reported, together with consequent productivity benefits.

This section has made a clear distinction between the inherent characteristics of an object, as defined by its type operations, and the characteristics which depend on its execution environment.

This section has also introduced a rigorous basis for error management, with systematic distinctions between operation exceptions and execution exceptions. These concepts are potentially a basis for provably correct, resilient and fault tolerant distributed systems structure.

(Blank page)

4. NOTATION

4.1 General

The notation used in this Technical Report is defined as follows:

- The data syntax notation and encoding and the macro notation are defined in CCITT Rec. X.409.
- The Remote Operations macro is defined in CCITT Rec. X.410, section 2.
- Enhancements to the Remote Operations macro are defined in clause 4.2 of this Technical Report.

Guidelines for a simplified method of using the macro notation are described in clause 4.4 of this Technical Report.

4.2 Notation for Remote Operations

4.2.1 Type of Operations

A Bind-Operation defines where an object binding should begin. If such a binding is established, Ordinary-Operations may be invoked. An Unbind-Operation defines where an object binding should be released. If no object binding is required, Ordinary-Operations may be invoked without this object binding.

4.2.2 Definitions of Macros and Data Types

An interactive protocol is specified using the Operation and Error data types. This clause defines those types. It also explains the notational definitions of a particular Operation, and of the particular Errors it can report. The notation is defined by means of the X.409 macro definition. This macro definition allows a generalised specification of mapping onto the various execution environments.

A data value of type Operation represents the identifier for an operation that an AE in one open system may request to be performed by a peer AE in another open system. A single data value, the argument of the operation, accompanies the request. Some operations report their outcome, whether success or failure. Other operations report their outcome only if they fail, and still others never at all. A single data value, the result of the operation, accompanies a report of success; a report of failure identifies the exceptional condition that was encountered.

The notation for an Bind-Operation type is the keyword **BIND**, optionally followed by the keyword **ARGUMENT** and the type of the operations's argument, the reference name optionally assigned to it, and the nature of the operation's outcome reporting (if any). If the operation reports success, the keyword **RESULT** and the type of its result and the reference name optionally assigned to it are specified. If the operation reports failure, the keyword **BIND-ERROR** and the type of the error-information it reports and the reference name optionally assigned to it are specified. The notation for an Operation value is the operations's numeric identifier.

The notation for an Ordinary-Operation type is the keyword **OPERATION**, optionally followed by the keyword **ARGUMENT** and the type of the operations's argument, the reference name optionally assigned to it, and the nature of the operation's outcome reporting (if any). If the operation reports success, the keyword **RESULT** and the type of its result and the reference name optionally assigned to it are specified. If the operation

reports failure, the keyword **ERRORS** and the reference names of the errors it reports are specified. The notation for an Operation value is the operations's numeric identifier.

The notation for an Unbind-Operation type is the keyword **UNBIND**, optionally followed by the keyword **ARGUMENT** and the type of the operations's argument, the reference name optionally assigned to it, and the nature of the operation's outcome reporting (if any). If the operation reports success, the keyword **RESULT** and the type of its result and the reference name optionally assigned to it are specified. If the operation reports failure, the keyword **UNBIND-ERROR** and the type of the error-information it reports and the reference name optionally assigned to it are specified. The notation for an Operation value is the operations's numeric identifier.

A data value of type Error represents the identifier for an exceptional condition that an AE in one open system may report to a peer AE in another open system an exceptional outcome of a previously requested Ordinary Operation. A single data value, the parameter of the error, accompanies the report.

The notation for an Error type is the keyword **ERROR**, optionally followed by the keyword **PARAMETER** and the type of the error's parameter and the reference name optionally assigned to it. The notation for an Error value is the error's numeric identifier:

The macros which define the Operation types and the Error type are listed in Figure 4.

4.3 Notation Clarifications

There are still certain clarifications needed about the CCITT Rec. X.410 notation. It is expected that these problems will be addressed by the CCITT document "X.400-Series Implementor's Guide". Users of this Technical Report should therefore refer to that Guide.

The main reason for restricting the **BIND-ERROR** and **UNBIND-ERROR** to a single data type (instead of an Error Macro) is to retain backwards compatibility with the CCITT X.400-Series of Recommendations. This is the only respect in which the notation of the Bind Macro and the Unbind Macro differs from that of the Operation Macro.

```
RemoteOperations DEFINITIONS ::=
BEGIN

-- macro definition for Bind Operations

BIND MACRO ::=
BEGIN

TYPE NOTATION    ::= "ARGUMENT" NamedType Result Errors | empty
VALUE NOTATION   ::= value (VALUE INTEGER)

Result           ::= empty | "RESULT" NamedType
Errors           ::= empty | "BIND-ERROR" NamedType

NamedType        ::= identifier type | type

END

-- macro definition for Ordinary Operations

OPERATION MACRO ::=
BEGIN

TYPE NOTATION    ::= "ARGUMENT" NamedType Result Errors | empty
VALUE NOTATION   ::= value (VALUE INTEGER)

Result           ::= empty | "RESULT" NamedType
Errors           ::= empty | "ERRORS" "{" ErrorNames "}"

NamedType        ::= identifier type | type

ErrorNames       ::= empty | IdentifierList
IdentifierList    ::= identifier | IdentifierList "," identifier

END

-- macro definition for Ordinary Operations Errors

ERROR MACRO ::=
BEGIN

TYPE NOTATION    ::= "PARAMETER" NamedType | empty
VALUE NOTATION   ::= value (VALUE INTEGER)

NamedType        ::= identifier type | type

END

-- macro definition for Unbind Operations

UNBIND MACRO ::=
BEGIN

TYPE NOTATION    ::= "ARGUMENT" NamedType Result Errors | empty
VALUE NOTATION   ::= value (VALUE INTEGER)

Result           ::= empty | "RESULT" NamedType
Errors           ::= empty | "UNBIND-ERROR" NamedType

NamedType        ::= identifier type | type

END

END
```

Figure 4. Formal Definition of Remote Operations Data Types

4.4 Using Remote Operation Macros

Use of the macro notation may at first be difficult for those not familiar with it. This clause defines a simple template methodology which can be used to help overcome this problem.

4.4.1 Templates for Ordinary-Operations and Errors

Figure 5 illustrates a template for the structure of the complete Ordinary-Operation macro.

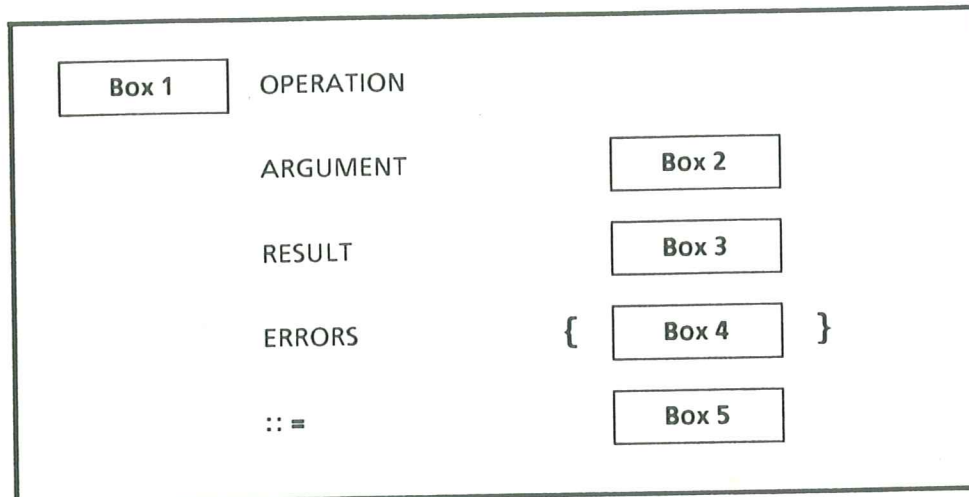


Figure 5. Normal Template for Specifying A Remote Operation

All that has to be done is to fill in the boxes as follows:

- Box 1. The name of the operation. This is a string comprising any letters A-Z (upper and lower case), decimal digits and hyphens (used singly), starting with a lower case letter. No spaces.
- Box 2. Type of the parameter which is to be passed to the destination when the operation is invoked. The type may be a set or a list of subtypes. These are specified in the X.409 abstract syntax notation.
- Box 3. Type of the return result parameter, specified as for box 2.
- Box 4. A list of error names (see explanation below).
- Box 5. The operation code which distinguishes this operation from others. Any integer.

Figure 6 illustrates the template for the specification of each error name.

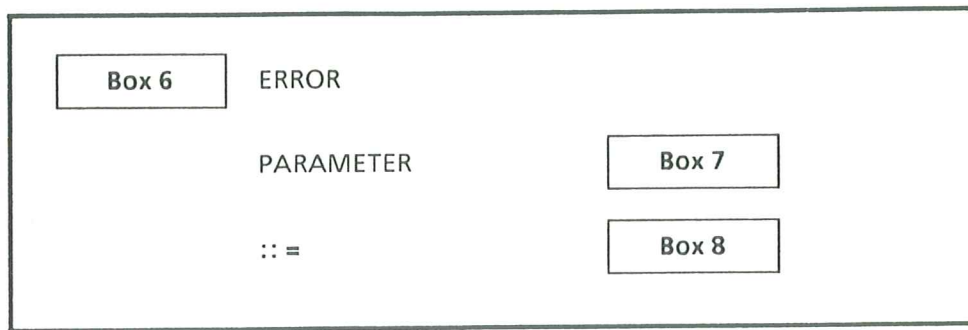


Figure 6. Template for Specifying the Details for Each Error Name

The template should be used for each error name listed in the operation template (eg. Figure 5).

- Box 6. The error name. Same syntax notation rules as for OPERATION name in the distributed-operation template.
- Box 7. Optional error parameters, specified in the X.409 abstract syntax notation.
- Box 8. The error code which distinguishes this error from others. Any integer.

To assist readability of the specification, the X.409 abstract syntax notation used for the contents of boxes 1, 2, 3, and 7 may include names (as allowed by CCITT Rec. X.409).

Some operations do not need all the boxes. See Figures 7 and 8.

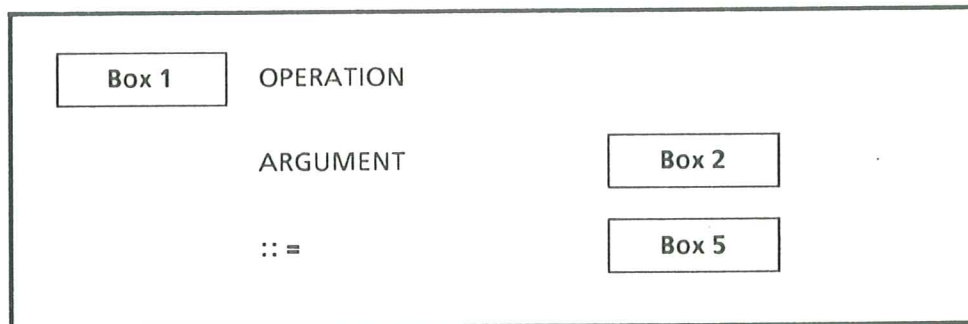


Figure 7. Template for Specifying An Operation Which Does Not Return Any Results or Errors.

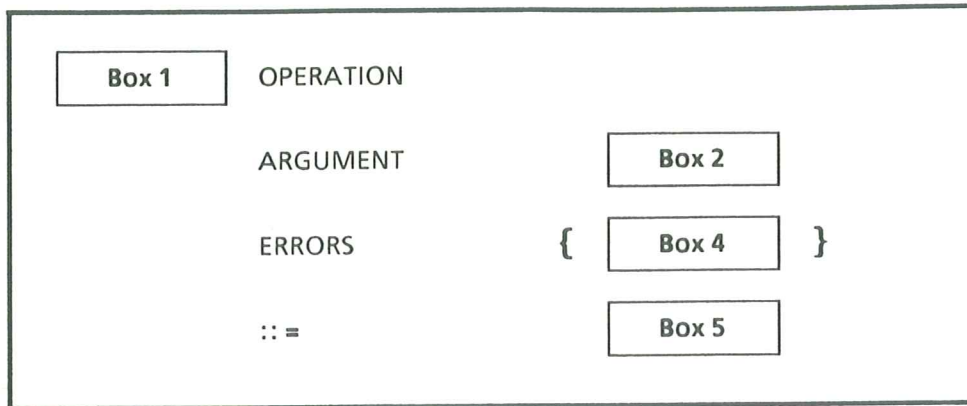


Figure 8. Template for Specifying An Operation Which Does Not Return Any Normal Results, But May Return Errors.

The specification of (remote) type operations provided by this macro and template notation is complete, formal and declarative. This has beneficial implications for specification correctness, software engineering tools and computer languages. The declarative structure may be particularly important for so called "fifth generation" languages and machine architectures which can then exploit the inherent opportunities for parallel execution.

4.4.2 Templates for Bind- and Unbind-Operations

Figure 9 illustrates a template for the structure of the complete Bind-Operation macro.

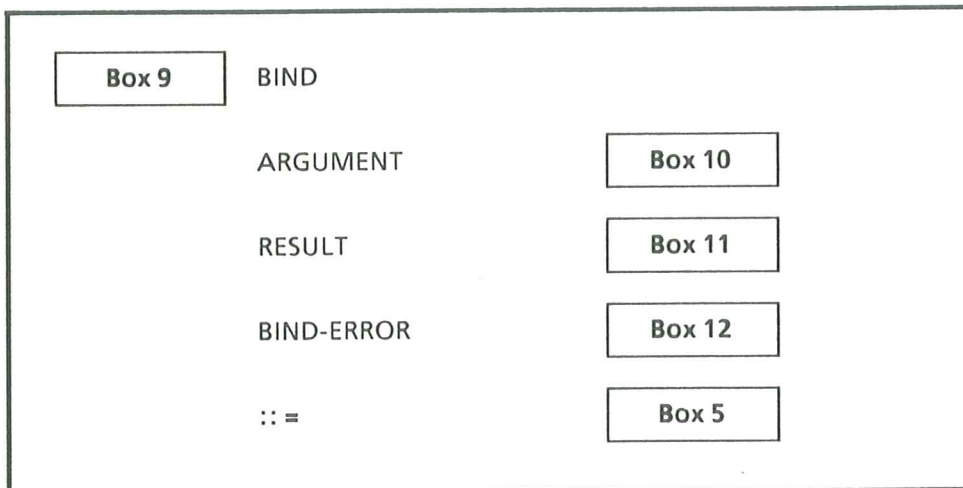


Figure 9. Normal Template for Specifying A Bind-Operation

All that has to be done is to fill in the boxes as follows:

- Box 9. The name of the Bind-operation. Same syntax notation rules as for OPERATION name (Box 1 of Figure 5).
- Box 10. Type of data associated with the establishment of an object-binding. Same syntax notation rules as for ARGUMENT in the Ordinary-Operation macro (Box 2 of Figure 5).

- Box 11. Type of data associated with the successful establishment of an object-binding. Same syntax notation as for Box 10.
- Box 12. Type of data associated with the refusal of an object-binding. Same syntax notation as for Box 10.
- Box 5. The operation code which distinguishes this operation from others. Any integer.

Figure 10 illustrates a template for the structure of the complete Unbind-Operation macro.

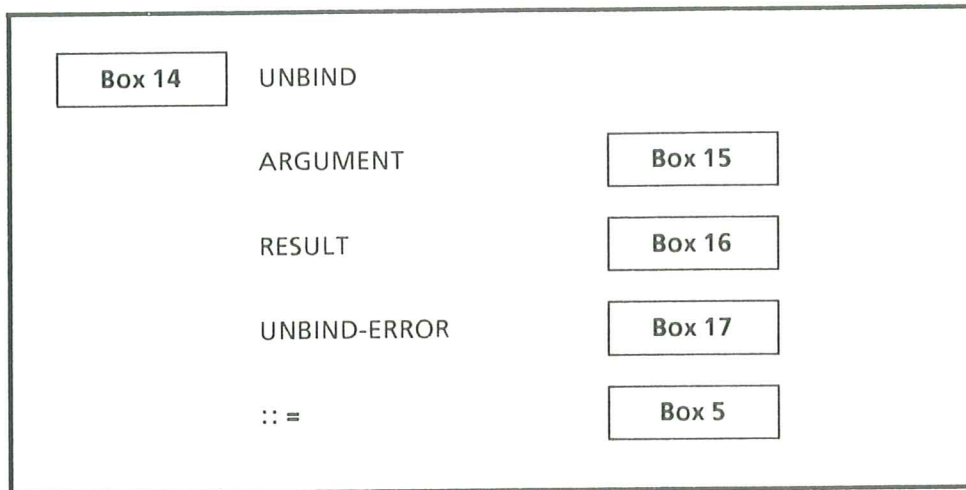


Figure 10. Normal Template for Specifying An Unbind-Operation

All that has to be done is to fill in the boxes as follows:

- Box 14. The name of the Unbind-Operation. Same syntax notation rules as for OPERATION name (Box 1 of Figure 5).
- Box 15. Type of data associated with the termination of an object-binding. Same syntax notation rules as for ARGUMENT in the Ordinary-Operation macro (Box 2 of Figure 5).
- Box 16. Type of data associated with the successful termination of an object-binding. Same syntax notation as for Box 15.
- Box 17. Type of data associated with the refusal of the termination of an object-binding. Same syntax notation as for Box 15.
- Box 5. The operation code which distinguishes this operation from others. Any integer.

Some of the Bind- and Unbind-Operations do not need all the boxes. See Figures 11 and 12.

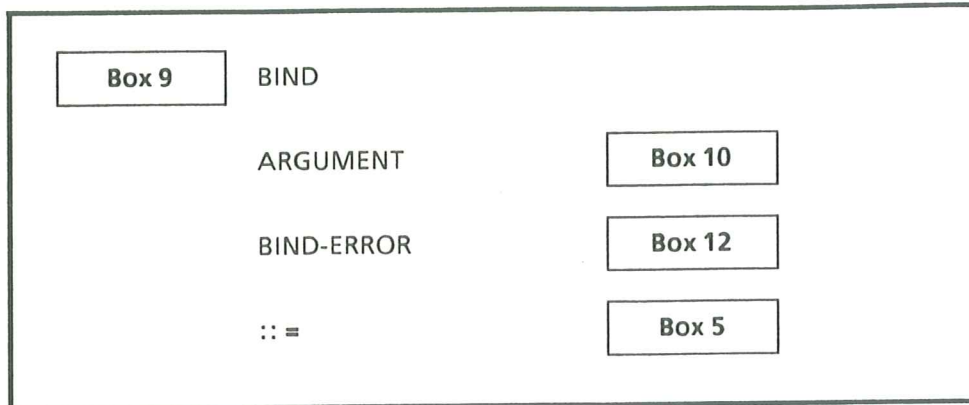


Figure 11. Template for Specifying a Bind-Operation Which Does Not Return A Result.

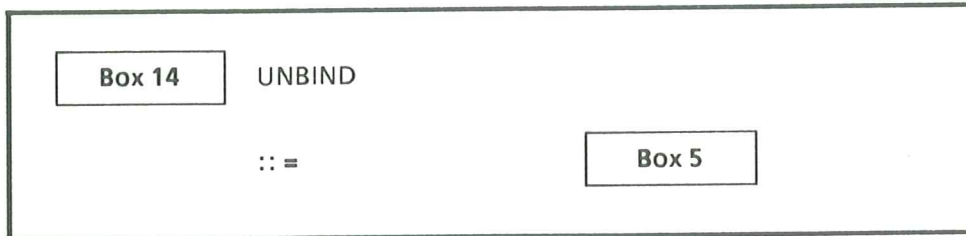


Figure 12. Template for Specifying an Unbind-Operation Requiring No Argument and Which Does Not Return Any Result or Error.

5. SERVICE AND PROTOCOL

5.1 CO-ROS Service Overview

The Connection-oriented Remote Operation Service (CO-ROS) is provided by a sublayer which is in the Application Layer of the OSI model. A fundamental characteristic of CO-ROS is that it encapsulates from the AE the OSI services. Its functionality is to provide a mechanism by which a pair of Application Entities (AE) can establish an ROS-association with each other, and thence one AE can request operations from, or perform operations for, the other AE.

Each AE is also provided with a mechanism for returning results of any operation it was asked to perform, or error or reject notification, to the other AE.

The Remote Operation Service embodies the concept of an ROS-association between two Application Entities.

An ROS-association is established as a result of a ROS-user issuing a RO-BEGIN.request. This request identifies the remote ROS-user with which a ROS-association is requested. It also contains any identification and authentication parameters that have been mutually agreed between the ROS-users in the Application protocol.

The ROS entity will inform the ROS-user of the creation of a ROS-association by a remote ROS-user, by issuing a RO-BEGIN.indication. This allows the ROS-user to match its resource to the expected level of activity from the remote ROS-user. It also allows the ROS-user to refuse the ROS-association, if necessary, by use of a RO-BEGIN.response.

When a ROS-association is no longer required by the initiating ROS-user, that ROS-user issues a RO-END.request. This tells the ROS entity that the ROS-user has no further OSDU to transmit, and causes the ROS to close down the association to the remote ROS entity.

Between issuing a RO-BEGIN.request and a RO-END.request a ROS-user may issue RO-INVOKE.requests, RO-RESULT.requests, RO-ERROR.requests and RO-REJECT.requests subject to the selected ROS-association Class defined for the particular protocol. The parameters of these primitives are OSDUs, for transfer to the remote ROS-user.

Certain parameters of the primitives allow ROS to manage and optimize its use of underlying OSI services:

- Priority. This defines the priority of the subject OSDU with respect to other OSDUs which can be exchanged between these AEs. The lower the value, the higher the priority.
- Operation Class. This defines, among others, whether a synchronous or an asynchronous response is expected and can be used by the ROS entity to manage the turn. (For details see clause 5.1.3).

A ROS-user can specifically cause a new ROS-association to be created by issuing another RO-BEGIN.request. This causes a RO-BEGIN.indication to the remote ROS-user, who may accept or reject the new ROS-association by use of the RO-BEGIN.response. Acceptance or refusal of the ROS-association is passed to the initial ROS-user in a RO-BEGIN.confirmation. This is a User Data parameter with RO-BEGIN.request/indication, and with RO-BEGIN.response/confirmation, which can be used by mutual agreement between the ROS-users, depending on the Application Protocol Standard.

Clause 5.3 gives a schematic representation of CO-ROS.

5.1.1 Operation Modes

There are two possible operation modes:

- Synchronous Mode of Operation. The invoking AE requires a reply from the AE performing the operation before it will invoke any other operation.
- Asynchronous Mode of Operation. The invoking AE may continue to issue further invocations without waiting for any reply.

5.1.2 Types of Operations

Operations are typed depending on the expected return:

- Result or Error;
- Error (if any)
- Nothing

A Reject can be returned for all types.

5.1.3 Operation Classes

The valid combinations of Operation Mode and Operation Type are defined as an Operation Class as follows:

- 1 - Synchronous Mode with Result or Error;
- 2 - Asynchronous Mode with Result or Error;
- 3 - Asynchronous Mode with Error (if any);
- 4 - Asynchronous Mode without return.

The Operation Class of each operation has to be agreed between the ROS-users (e.g. in an Application protocol).

Although all four forms are potentially allowed in CCITT Rec. X.410, the rest of this Technical report only considers the first two classes; the two other classes are for further study.

5.1.4 ROS-association Classes

The ROS-associations between two ROS-users are classified as follows:

- 1 - Only the initiating ROS-user can invoke ordinary operations;
- 2 - Only the responding ROS-user can invoke ordinary operations;
- 3 - The responding as well as the initiating ROS-user can invoke ordinary operations.

Although all three classes are potentially allowed in the CCITT X.410 Rec. the rest of this Technical Report only considers the first class; the two other classes are for further study.

The initiating ROS-user, the initiator, is the ROS-user which starts the ROS-association with a RO-BEGIN.request. The responding ROS-user, the responder, is the other ROS-user in a ROS-association.

The ROS-association Class used has to be agreed between the ROS-users (e.g. in an Application protocol standard).

5.2 CO-ROS: Service Primitives

5.2.1 Overview and Description

The Remote Operation Services are described as a set of service primitives and their parameters.

Name of primitives	Facility
RO-BEGIN .request .indication .response .confirmation	Initiate an ROS-association
RO-END .request .indication .response .confirmation	Terminate an ROS-association
RO-INVOKE .request .indication	Invocation of an Operation
RO-RESULT .request .indication	ROS-user Result of an Operation
RO-ERROR .request .indication	ROS-user Error of an Operation
RO-REJECT-U .request .indication	ROS-user Reject
RO-REJECT-P .indication	ROS-provider Reject

Table 1. CO-ROS Service Primitives.

5.2.1.1 **RO-BEGIN**

The RO-BEGIN primitives are used by a ROS-user to initiate a ROS-association with a peer ROS-user to create an execution binding. The RO-BEGIN may convey user data for object binding, e.g. authentication parameters.

The service structure of RO-BEGIN consists of four events, as illustrated in Figure 13.

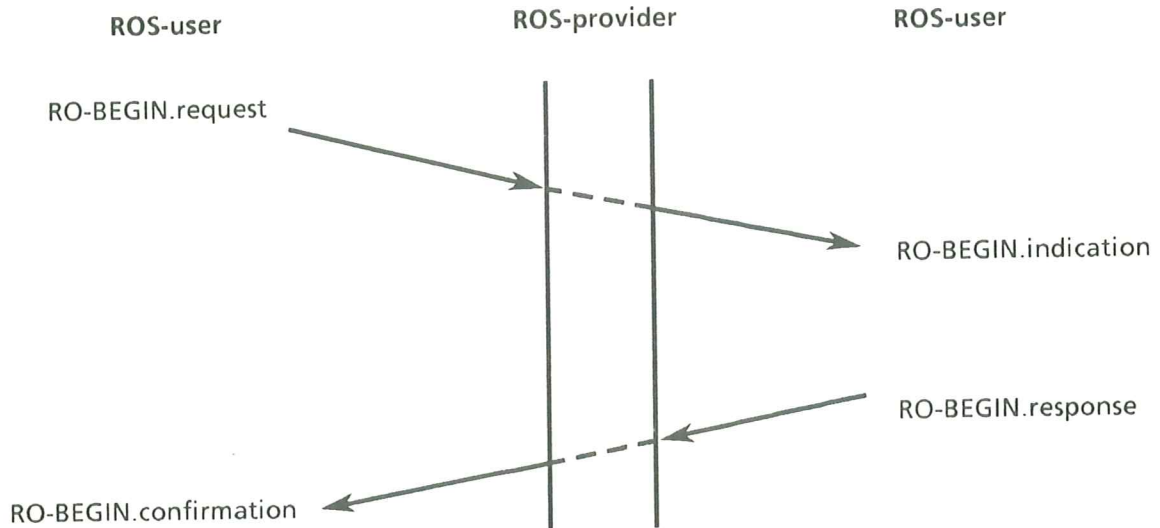


Figure 13. RO-BEGIN Primitive Events

The RO-BEGIN.request is issued by a ROS-user to initiate a ROS-association with another ROS-user.

Parameter Name	Type	Parameter Description	Note
Destination-reference	M	A unique identifier of the responding ROS-user	
Application-protocol	M	Designates the application protocol that will govern the ROS-association	1
User-data	C	User data associated with opening the ROS-association	1

Table 2. Parameters of the RO-BEGIN.request primitive

Note 1

This value has to be agreed between the Applications using ROS.

Parameter Name	Type	Parameter Description	Notes
Initiator-reference	M	A unique identifier for the ROS-user that issued the RO-BEGIN.request	
Application-protocol	M	Designates the application protocol that will govern the ROS-association	1, 2
User-data	C	User data associated with opening the ROS-association	1, 2

Table 3. Parameters of the RO-BEGIN.indication primitive

Note 2

The value of this parameter is the same as in the RO-BEGIN.request primitive.

Parameter Name	Type	Parameter Description	Notes
Disposition	M	The disposition of the request for an ROS-association: accepted or refused	
User-data	C	User data associated with accepting the ROS-association	1, 3
Refusal-reason	C	The reason for refusing the ROS-association	4

Table 4. Parameters of the RO-BEGIN.response primitive

Note 3

This parameter is present only if the ROS-association is accepted.

Note 4

This parameter is present only if the ROS-association is refused. Values defined are: authentication failure, busy.

The parameters of the RO-BEGIN.confirmation primitive are identical to, and have the same values as the RO-BEGIN.response primitive.

5.2.1.2 RO-END: Terminate a ROS-association

The initiating ROS-user issues the RO-END.request primitive to release or close the ROS-association in order to terminate an execution binding.

Four events occur when a ROS-association is released, as illustrated in Figure 14.

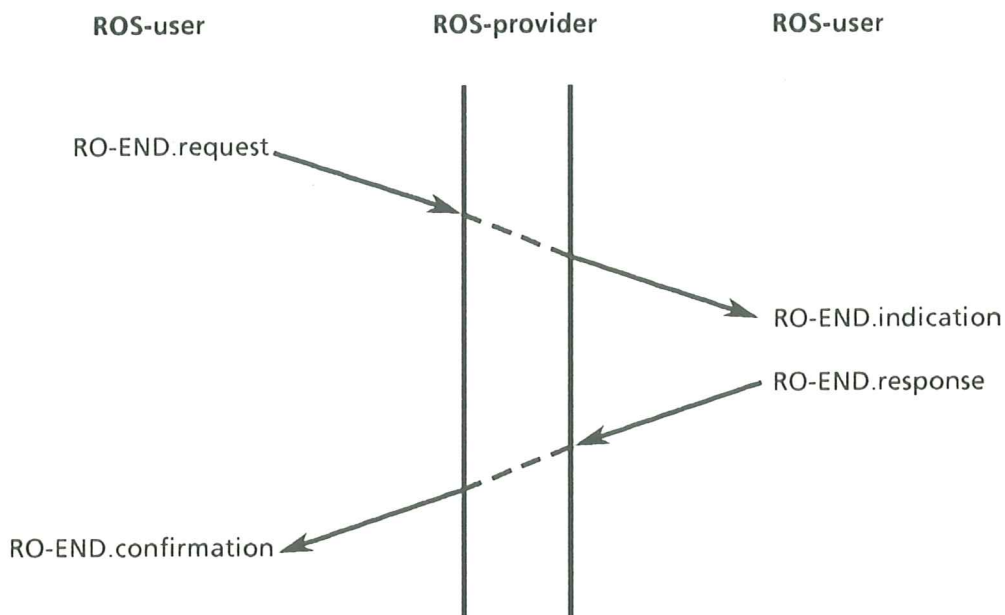


Figure 14. RO-END Primitive Events

Parameter Name	Type	Parameter Description	Note
Priority	C	The priority of OSDUs the ROS-user is prepared to receive before release of the ROS-association	5

Table 5. Parameters of the RO-END.request primitive

There are no parameters associated with the RO-END.indication/response/confirmation primitives.

Note 5
This is for further study.

5.2.1.3 RO-INVOKE: Invocation of an Operation

The RO-INVOKE primitives are used by a ROS-user to allow the ROS-user to request an operation to be performed by a remote ROS-user.

The related service structure consists of two events, as illustrated in Figure 15.

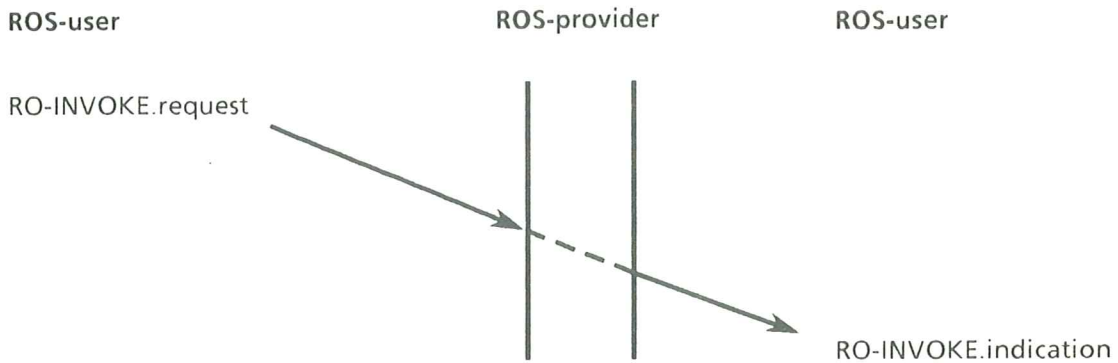


Figure 15. RO-INVOKE Primitive Events

The RO-INVOKE.request is issued by a ROS-user to invoke an operation performed by another ROS-user.

Parameter Name	Type	Parameter Description	Notes
Operation	M	Identifies the operation that has to be performed by the remote AE	1, 6
OP-class	C	See clause 5.1.3	1, 7
Arguments	M	Data to be transferred. It consists of the arguments of the operation	1, 6
Invoke-ID	M	Identifier of the request	
Priority	C	The priority assigned to the transmission of OPDUs, as defined by the Application protocol standard	

Table 6. Parameters of the RO-INVOKE.request primitive

Note 6

Operation codes and arguments are only significant to the ROS-user. The values of these parameters are the same as in the RO-INVOKE.indication.

Note 7

Used by ROS to optimize the use of the underlying OSI services. No optimization takes place if this is omitted.

The RO-INVOKE.indication is issued by the ROS-provider.

Parameter Name	Type	Parameter Description	Notes
Operation	M	Identifies the operation that has to be performed by the remote AE	1, 8
Arguments	M	Data transferred. It consists of the arguments of the operation	1, 6
Invoke-ID	M	Identifier of the request	9

Table 7. Parameters of the RO-INVOKE.indication primitive

Note 8

Operation codes are only significant to the ROS-user.

Note 9

The Invoke-ID is to be used in the RO-RESULT, RO-ERROR or RO-REJECT primitives.

5.2.1.4 RO-RESULT: Response to an Invocation

The RO-RESULT primitives are used by a ROS-user to transfer the response to a previous invocation, in the case of a successful remote operation.

The related service structure consists of two events, as illustrated in Figure 16.

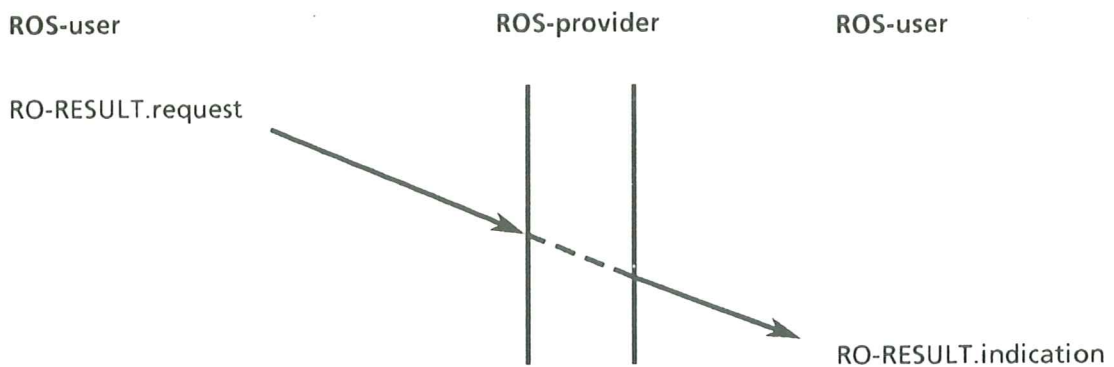


Figure 16. RO-RESULT Primitive Events

Parameter Name	Type	Parameter Description	Note
Invoke-ID	M	Identifies a previous invocation	1
Result	M	Result of the operation	
Priority	C	The priority assigned to the transmission of OPDUs, as defined by the Application protocol standard	

Table 8. Parameters of the RO-RESULT.request primitive

The RO-RESULT.indication is issued by the ROS-provider.

Parameter Name	Type	Parameter Description	Notes
Invoke-ID	M	Identifies a previous invocation	10
Result	M	Result of the operation	1, 10

Table 9. Parameters of the RO-RESULT.indication primitive

Note 10

The value of this parameter is the same as in the RO-RESULT.request primitive.

5.2.1.5 RO-ERROR: Error Response to an Invocation

The RO-ERROR primitives are used by a ROS-user to transfer an error response to a previous invocation, in the case of an unsuccessful remote operation.

The related service structure consists of two events, as illustrated in Figure 17.

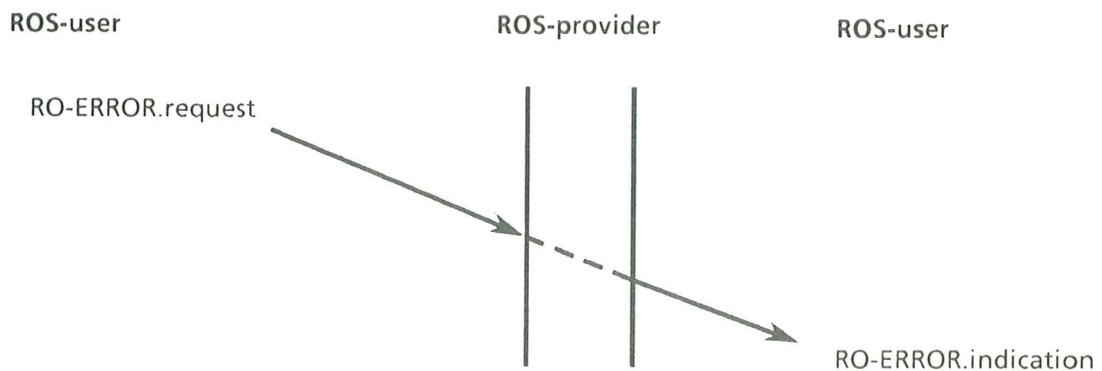


Figure 17. RO-ERROR Primitive Events

The RO-ERROR.request is issued by a ROS-user to notify an error response to an invocation.

Parameter Name	Type	Parameter Description	Notes
Invoke-ID	M	Identifies a previous invocation	9
Error	M	The error code	1
Error-parameters	C	The error parameters	1
Priority	C	The priority assigned to the transmission of OPDUs, as defined by the Application protocol	

Table 10. Parameters of the RO-ERROR.request primitive

The RO-ERROR.indication is issued by the ROS-provider to pass the received data.

Parameter Name	Type	Parameter Description	Notes
Invoke-ID	M	Identifies a previous invocation	9
Error	M	The error code	1, 11
Error-parameters	C	The error parameters	1, 11

Table 11. Parameters of the RO-ERROR.indication primitive

Note 11

The value of this parameter is the same as in the RO-ERROR.request primitive.

5.2.1.6 RO-REJECT-U

The RO-REJECT-U primitives are used to advise the ROS-user of a problem detected by the remote ROS-user.

The service structure of RO-REJECT-U consists of two events, as illustrated in Figure 18.

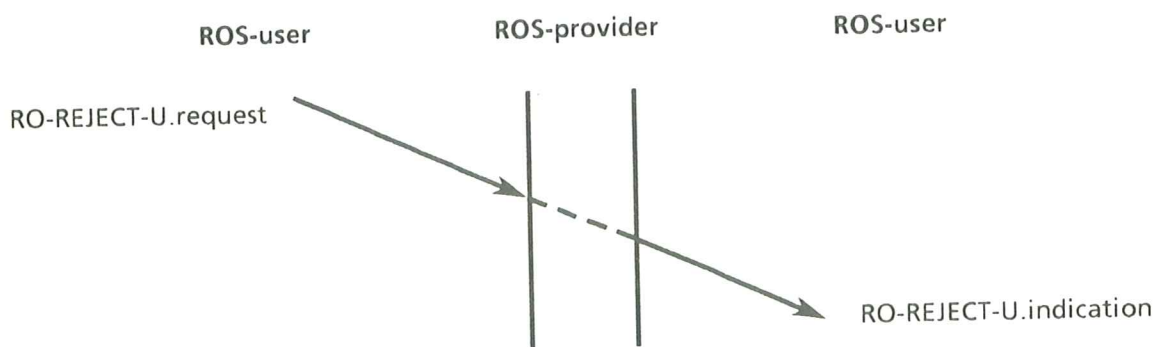


Figure 18. RO-REJECT-U Primitive Events

Parameter Name	Type	Parameter Description	Note
Invoke-ID	M	ID of the invocation, result or error	12
Reject-reason	M	Indicates the reason for rejection	
Priority	C	The priority assigned to the transmission of OPDUs, as defined by the Application protocol	

Table 12. Parameters of the RO-REJECT-U.request primitive

Note 12

For a list of reject reasons see clause 5..4. All except General Problem can occur.

Parameter Name	Type	Parameter Description	Notes
Invoke-ID	M	ID of the invocation, result or error which failed	12
Reject-reason	M	Indicates the reason for rejection	

Table 13. Parameters of the RO-REJECT-U.indication primitive

5.2.1.7 RO-REJECT-P

The RO-REJECT-P primitives are used to advise the ROS-user of a problem detected by the ROS-provider.

The service structure of RO-REJECT-P consists of a single event, as illustrated in Figure 19.

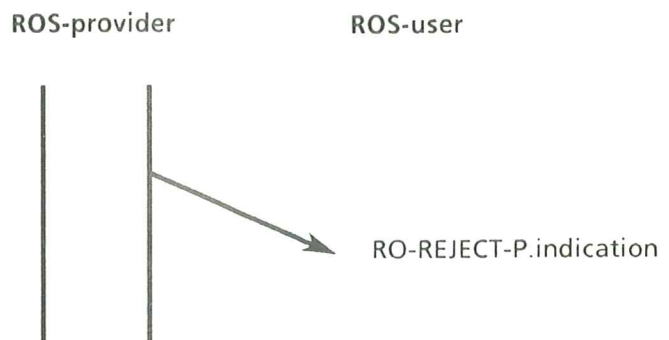


Figure 19. RO-REJECT-P Primitive Event

Parameter Name	Type	Parameter Description	Notes
Invoke-ID	C	ID of the invocation, result or error which failed	13
OSDU-not-transferred	C	The invocation, result, error or reject OSDU which could not be transferred within the allotted time	14, 16
Reject-reason	C	Indicates the reason for rejection	15, 16

Table 14. Parameters of the RO-REJECT-P.indication primitive

Note 13

May be omitted if an unrecognized OPDU.

Note 14

Present if the underlying layer is unable to transfer the OPDU. Local reject can occur.

Note 15

For a list of reject reasons see clause 5.4. General Problem can occur.

Note 16

The "OSDU-not-transferred" and the "Reject-reason" are mutually exclusive.

5.3 CO-ROS Service Schematic Representation

5.3.1 Introduction

The purpose of this clause is to give a schematic description of how the CO-ROS works. The diagrams in Figures 20, 21, 22 and 23 specify the relationship between the service primitives and the OPDUs.

5.3.2 Relation between ROS Service Primitives, OPDUs and Error and Reject Situations

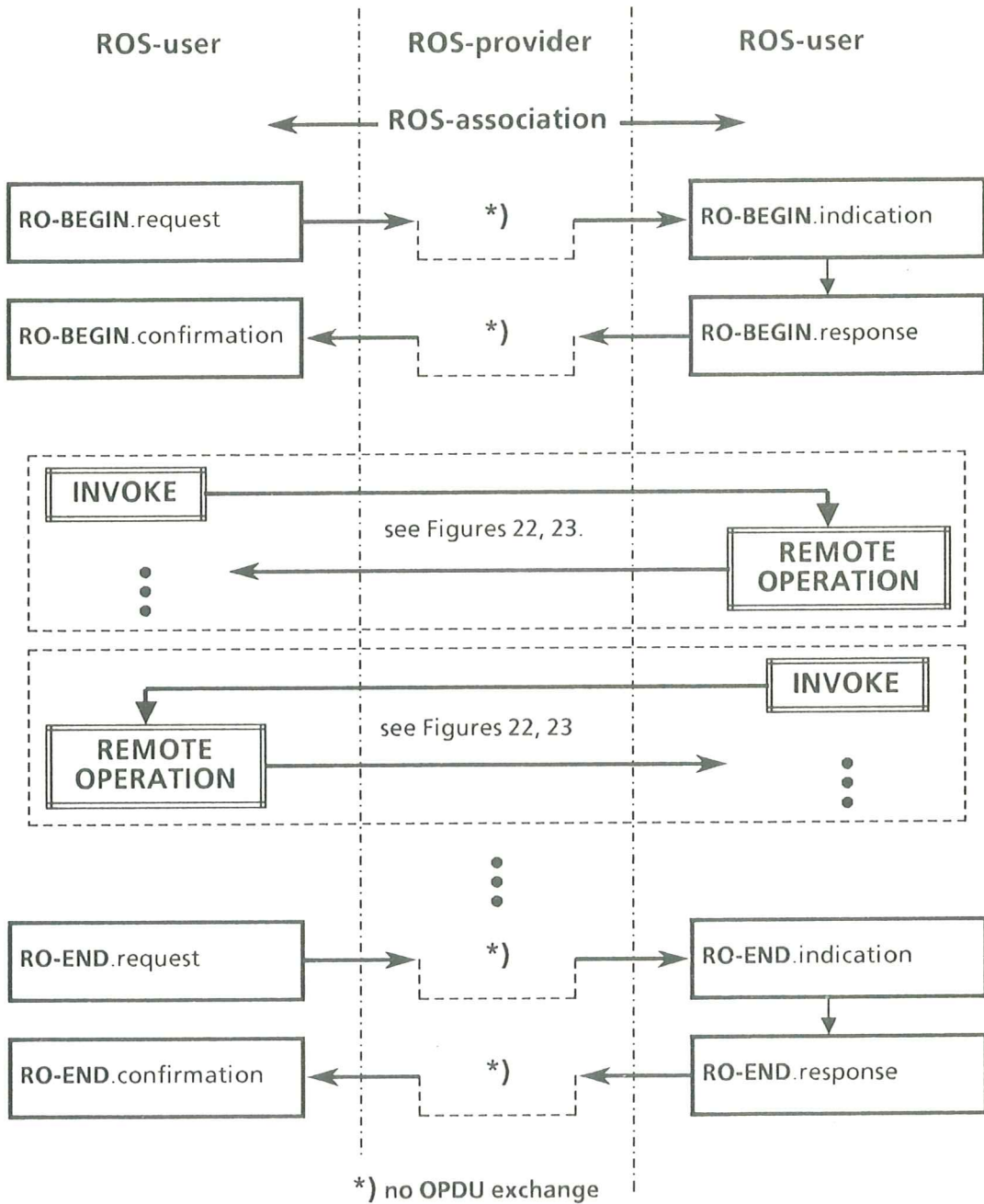


Figure 20. CO-ROS Operation Overview

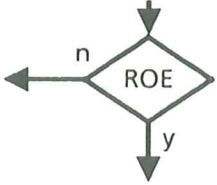
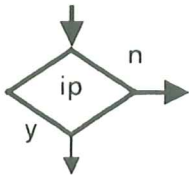
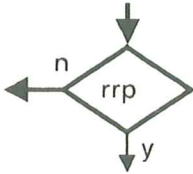
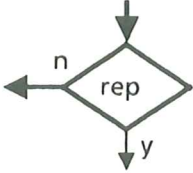


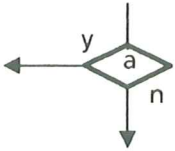
Symbols (used in Figures 22 and 23.)	Error conditions
	Remote Operation Errors: Remote Operation Error?
	ROS-user Errors: Invoke Problem? duplicate invocation unrecognised operation mistyped argument
	Return Result Problem? unrecognised invocation result response unexpected mistyped result
	Return Error Problem? unrecognised invocation error response unexpected unrecognised error unexpected error mistyped parameter
	ROS Errors: General Problem? unrecognised OPDU mistyped OPDU badly structured OPDU
	OPDU transfered in time?
	ROS Abort Condition (set to avoid loops) or OPDU is a REJECT-OPDU

Figure 21. Symbols used in Figures 22 and 23.

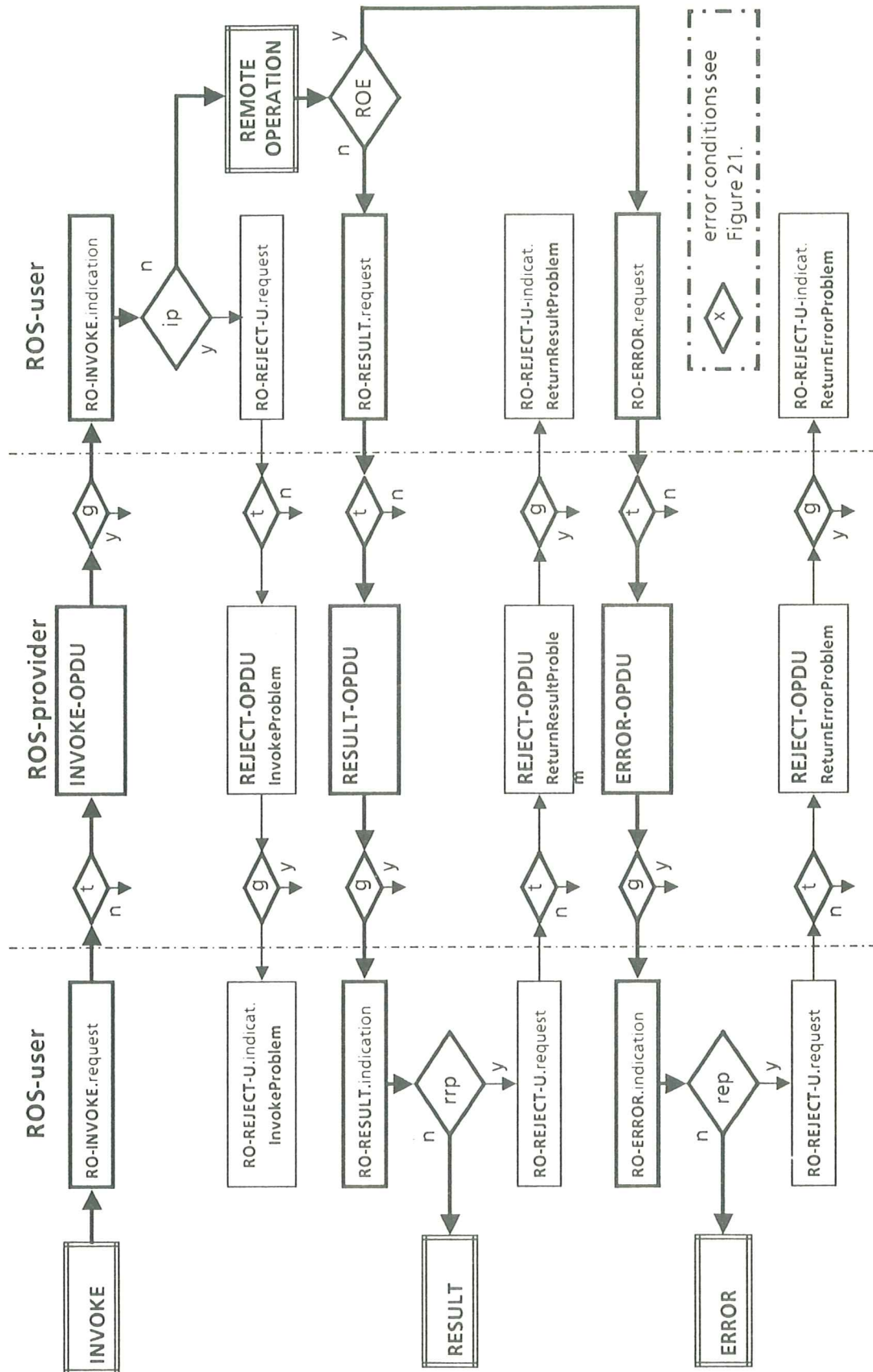


Figure 22. Invoke, Result, Error and Reject in ROS (continued in Figure 23.)

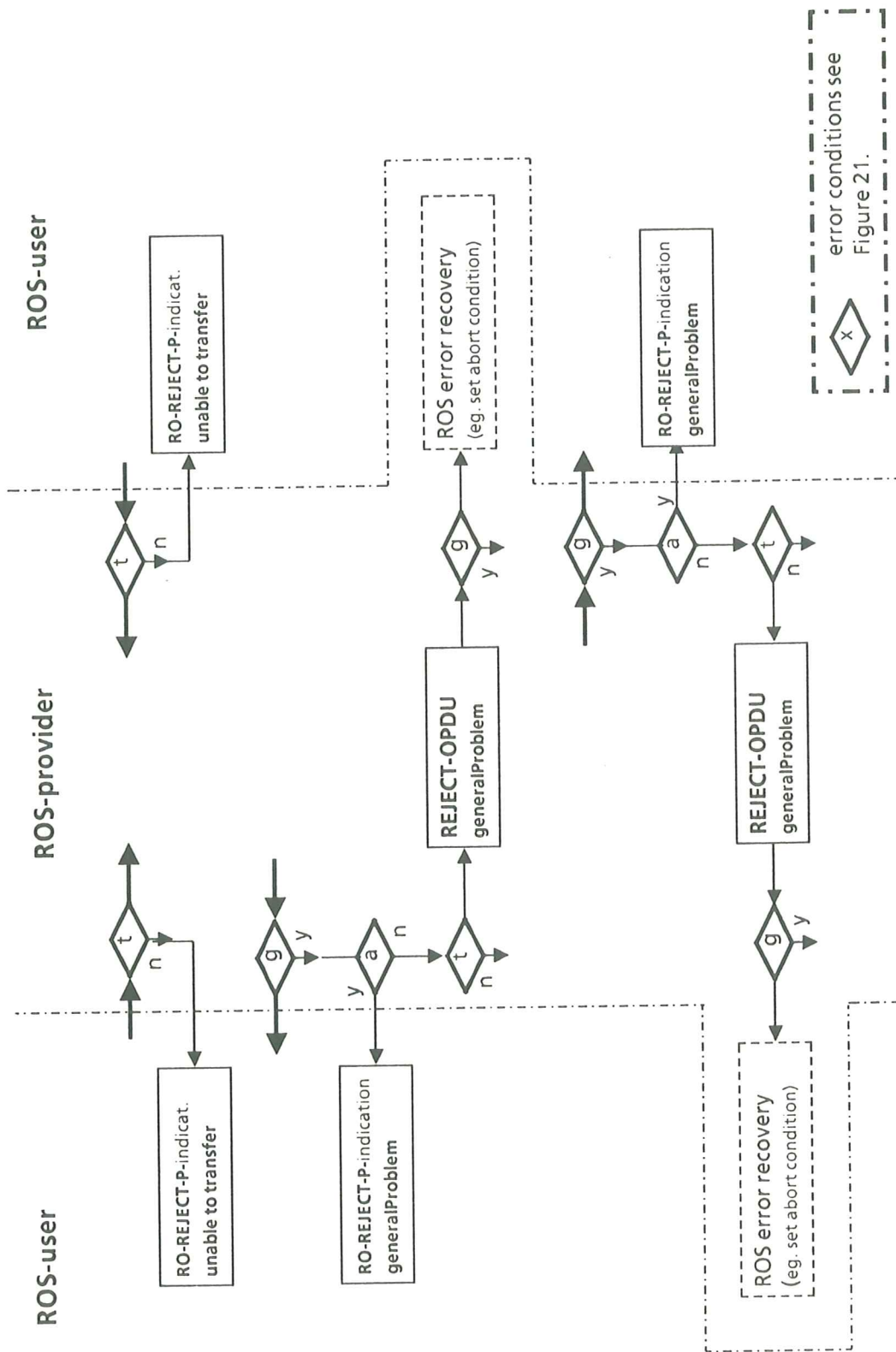


Figure 23. Reject in ROS (continued from Figure 22)

5.4 Remote Operation Protocol

For the Ordinary-Operations (OPERATION) and Ordinary-Operations Errors (ERRORS) macros, this Technical Report uses the same formal definition as CCITT Rec. X.410, section 2. This definition is sufficient to generate the Operation Protocol Data Units (OPDUs). This definition is reproduced in Figure 24.

```
ROS DEFINITIONS ::= =
BEGIN
--the following macros are used as defined in Figure 4
OPERATION          ::= RemoteOperations.OPERATION
ERROR              ::= RemoteOperations.ERROR
-- OPDU
OPDU               ::= CHOICE {[1] Invoke, [2] ReturnResult, [3] ReturnError, [4]
Reject}
-- OPDU types
Invoke            ::= SEQUENCE {invokeID INTEGER, OPERATION, argument ANY}
ReturnResult      ::= SEQUENCE {invokeID INTEGER, result ANY}
ReturnError       ::= SEQUENCE {invokeID INTEGER, ERROR, parameter ANY}
Reject            ::= SEQUENCE {
    invokeID CHOICE {INTEGER, NULL}
    problem CHOICE {
        [0] IMPLICIT GeneralProblem,
        [1] IMPLICIT InvokeProblem,
        [2] IMPLICIT ReturnResultProblem,
        [3] IMPLICIT ReturnErrorProblem}}
-- OPDU Rejection Reason
GeneralProblem    ::= INTEGER {
    -- ROS-provider detected
    unrecognisedOPDU (0),
    mistypedOPDU (1),
    badlyStructuredOPDU (2)}
InvokeProblem     ::= INTEGER {
    -- ROS-user detected
    duplicateInvokation (0),
    unrecognisedOperation (1),
    mistypedArgument (2),
    resourceLimitation (3)}
ReturnResultProblem ::= INTEGER {
    -- ROS-user detected
    unrecognisedInvokation (0),
    resultResponseUnexpected (1),
    mistypedResult (2)}
ReturnErrorProblem ::= INTEGER {
    -- ROS-user detected
    unrecognisedInvokation (0),
    errorResponseUnexpected (1),
    unrecognisedError (2),
    unexpectedError (3),
    mistypedParameter (4)}
END
```

Figure 24. Formal Definition of the ROS Protocol

5.5 Mapping of Macros onto CO-ROS

This clause describes how the macros described in clause 4.2 (Figure 4) are mapped onto the Service Primitives of the Remote Operation Service.

CO-ROS is a connection-oriented service, each connection being termed an "association". An association represents an object binding. This clause therefore describes rules for establishing and releasing associations, and the rules for performing remote operations, by use of Remote Operation Service Primitives.

5.5.1 Establishing and Releasing ROS-associations

An initiating AE establishes an ROS-associations by a Bind Operation. If the ROS-association is established, Ordinary Operations may be invoked. When the initiating AE wishes to terminate an ROS-association, it issues an Unbind Operation.

5.5.2 Mapping of Macros onto ROS Primitives

This clause describes the rules for mapping of macros onto the ROS primitives.

5.5.2.1 Mapping of the BIND Macro

This Macro maps onto the RO-BEGIN primitives as follows.

A BIND of an initiating AE is mapped onto an RO-BEGIN.request with the following parameters:

Parameter Name	Parameter Value
Destination reference	The ROS-address of the responding AE
Application protocol	Application protocol in use
User-data (initiating)	See ARGUMENT of the macro

The peer (responding) AE receives an RO-BEGIN.indication as result of the above request with the following parameters:

Parameter Name	Parameter Value
Initiator reference	The ROS-address of the initiating AE
Application protocol	Application protocol in use
User-data (initiating)	See ARGUMENT of the macro

When the responding AE wants to respond to an RO-BEGIN.indication, it issues an RO-BEGIN.response with the following parameters:

Parameter Name	Parameter Value
Disposition	Accepted or refused
User-data (accepting)	See RESULT of the macro (only if the ROS-association is accepted)
Refusal reason	See BIND-ERROR of the macro (only if the ROS-association is refused)

The initiating AE receives an RO-BEGIN.confirmation as result of this response with the following parameters:

Parameter Name	Parameter Value
Disposition	Accepted or refused
User-data (accepting)	See RESULT of the macro (only if the ROS-association is accepted)
Refusal reason	See BIND-ERROR of the macro (only if the ROS-association is refused)

5.5.2.2 Mapping of the OPERATION Macro

All OPERATION macros are mapped onto the RO-INVOKE, RO-RESULT and RO-ERROR primitives, according to the following general rules.

When an invoking AE wishes to invoke an operation, it issues an RO-INVOKE.request with the argument and operation number described in the macro. The priority and class of operation is as defined in a table specific to the particular Application protocol. The invoke ID is created locally by the invoker.

Parameter Name	Parameter Value
Operation	Identifies the operation that has to be performed. See Value of the macro
OP-class	See entry in the table specifying the operations
Arguments	ARGUMENT as defined in the macro
Invoke ID	Created locally by the invoker
Priority	See entry in the table specifying the operations

The invocation is indicated to the performing AE by an RO-INVOKE.indication with the following parameters:

Parameter Name	Parameter Value
Operation	Identifies the operation that has to be performed. See Value of the macro
Arguments	ARGUMENT as defined in the macro
Invoke ID	Created locally by the invoker

When the performing AE wishes to return the result of an operation, it issues an RO-RESULT.request with the following parameters:

Parameter Name	Parameter Value
Invoke ID	Identical to the invoke ID parameter of the corresponding RO-INVOKE.indication
Result	RESULT as defined in the macro
Priority	See entry return result in the table specifying the operations

The invoking AE receives an RO-RESULT.indication with the following parameters:

Parameter Name	Parameter Value
Invoke ID	Identical to the invoke ID parameter of the corresponding RO-INVOKE.indication
Result	RESULT as defined in the macro

When an AE wishes to indicate an error in an operation, it issues an RO-ERROR.request with the following parameters:

Parameter Name	Parameter Value
Invoke ID	Identical to the invoke ID parameter of the corresponding operation
Error	See value of ERROR macro
Error parameters	As defined in the ERROR macro
Priority	See entry return error in the table specifying the operations

The invoking AE receives an RO-ERROR.indication with the following parameters:

Parameter Name	Parameter Value
Invoke ID	Identical to the invoke ID parameter of the corresponding operation
Error	See value of ERROR macro
Error parameters	As defined in the ERROR macro

When an AE wishes to reject an RO-INVOKE, RO-RESULT or RO-ERROR indication, it issues an RO-REJECT-U.request with the following parameters:

Parameter Name	Parameter Value
Invoke ID	Identical to the invoke ID parameter of the corresponding operation
Reject reason	See clause 5.4
Priority	See entry reject in the table specifying the operation

The invoking AE receives an RO-REJECT-U.indication with the following parameters:

Parameter Name	Parameter Value
Invoke ID	Identical to the invoke ID parameter of the corresponding operation
Reject reason	See clause 5.4

5.5.2.3 Mapping of the UNBIND Macro

This macro maps onto the RO-END primitives as follows:

When the initiating AE wants to release an ROS-association, it issues an RO-END.request. An entity may only release an ROS-association which it has initiated.

The peer AE will receive an RO-END.indication as result of this request.

When the peer AE wants to respond to an RO-END.indication, it issues an RO-END.response. The initiating AE will receive an RO-END.confirmation as result of this request.

In the current ROS version **RESULT** and **UNBIND-ERROR** are ignored.

6. MAPPINGS ONTO UNDERLYING SERVICES

6.1 General

The following mappings of Remote Operations onto underlying services are defined in this Technical Report:

- Mapping of CO-ROS onto the Reliable Transfer Server (RTS) specified in CCITT Rec. X.410. RTS maps onto the BAS Session Service of ISO 8326. See clause 6.2
- Mapping onto the BCS Session Service of ISO 8326. See clause 6.3.

Mappings onto other services are possible and are not precluded. See clause 1.2.4.

6.2 Mapping CO-ROS onto Reliable Transfer Server

6.2.1 Introduction

This clause describes how the Remote Operation Service (ROS) is supported by the Reliable Transfer Server (RTS) as described in Standard ECMA-93 and CCITT Rec. X.410.

6.2.2 Creating and Releasing ROS-associations

An ROS-association is created by the ROS at the specific request of the ROS-user by establishing an RTS-association. There is a one-to-one mapping between ROS-associations and RTS-associations.

An RTS-association is established by issuing the RT-OPEN.request primitive with the following parameters:

Parameter Name	Parameter Value
Responder-address	The address of the RTS of the corresponding AE. This is derived from the RO-BEGIN.request Destination reference by a local mapping rule
Dialogue-mode	Two-way alternate
Initial-turn	The initiating ROS-user
Application-protocol	As specified in the RO-BEGIN.request
User-data	Provided by ROS-user as ROS User-data

The remote ROS entity issues an RT-OPEN.response with the following parameters:

Parameter Name	Parameter Value
Disposition	Accepted or refused. (As specified by the remote ROS-user)
User-data	If the association was accepted, provided by the ROS-user as ROS User Data
Refusal-reason	Unacceptable Dialogue Mode, Busy, or Validation failure

RTS-associations are released by invoking the RT-CLOSE.request primitive, which has no parameters.

6.2.3 Transferring OPDUs

Each OPDU is transferred by invoking the RT-TRANSFER.request primitive with the following parameters:

Parameter Name	Parameter Value
RSDU	The OPDU to be transferred
Transfer-time	As specified by a local rule of the ROS-provider, which may relate to the priority of the OPDU

The following are the rules for the sending of OPDUs on an association:

- I) Each OPDU is assigned a priority as defined in an Application Protocol Standard.
- II) On an association, higher priority OPDUs are sent first.
- III) On an association, OPDUs of the same priority are sent first-in-first-out (FIFO).

6.2.4 Managing the Turn

The ROS entity without the turn may issue an RT-TURN-PLEASE.request primitive, the priority parameter of which reflects the highest priority OPDU awaiting transfer.

When the initiating ROS-user requests the turn to issue an RT-CLOSE.request primitive to release the association, the priority parameter should reflect the lowest priority OPDU it is prepared to receive before releasing the RTS-association. This requires further study.

The ROS entity which has the turn will issue an RT-TURN-GIVE.request primitive in response to an RT-TURN-PLEASE.indication when it has no further OPDUs to transfer of priority equal to or higher than that indicated in the RT-TURN-PLEASE.indication. If it has OPDUs of lower priority still to transfer, it may either issue immediately an RT-TURN-PLEASE.request whose priority reflects the highest priority OPDU remaining to be transferred or it may refuse the RT-TURN-PLEASE and continue to transfer OPDUs.

6.2.5 Action on RT-EXCEPTION.indication

The RTS will issue an RT-EXCEPTION.indication if it is unable to transfer the OPDU within the specified time. This situation will result in an RO-REJECT-P.indication being issued, with the "OSDU-not-transferred" parameter being set.

6.3 Mapping CO-ROS onto ISO 8326 BCS Session Services

6.3.1 Introduction and Scope

This clause describes how the Remote Operation Service (ROS) is supported by the OSI Presentation and BCS Session Services. The "Kernel" functional unit is used, and optionally either the "Half-duplex" functional unit or the "Full-duplex" functional unit..

The ROS has minimal requirements on the Presentation Layer itself. It makes use of the Session Layer Services of ISO 8326, which are made directly available to the Application Layer entities by the Presentation Layer. A minimal Presentation Layer protocol, which meets the needs of the ROS, is implicitly defined in 6.3.2.1 below, and the remainder of this clause defines the use of the Session Layer Services by the ROS. The encoding of various parameters is defined using the notation of CCITT Rec. X.409.

The subset of the Session Service used is composed of the following functional units. The corresponding session services that are used are also listed:

Kernel	Session Connection Normal Data Transfer Orderly Release U-Abort P-Abort
Duplex	-
Half-duplex	Give Tokens Please Tokens

Note 17

Bit Alignment. Where referenced in this Technical Report, bit positions in octets are as defined in CCITT Rec. X.409. Bit 8 as defined there corresponds to the high-order bit of the Session Service, and bit 1 corresponds to the low-order bit.

Note 18

The details of this mapping are derived from the CCITT Rec. X.410 mapping of RTS onto BAS session, where applicable.

6.3.2 Session Connection Establishment Phase

The ROS attempts Session Connection Establishment in response to:

RO-BEGIN: An RO-BEGIN.request issued by the ROS-user.

An ROS Entity may reject an incoming connect indication. Specific reasons are given in the session connection response, see clause 6.3.2.1.

6.3.2.1 S-CONNECT

Parameter	Req/Ind	Resp/Conf
Session Connection Identifier	1	1
Calling SSAP Address	2	-
Called SSAP Address	2	2
Result	-	3
Quality of Service (QOS)	4	4
Session Requirements	5	5
Initial Data Token Assignment	6	6
SS-User Data	7	7

1. The initiating ROS-user will supply a Session Connection Identifier, which may be used to uniquely identify the connection. This identifier is formed of the following components: Calling SS-User Reference, Common Reference, and, optionally, Additional Reference Information. The identifier is returned unchanged by the responding ROS-user, except that the Calling SS-User Reference supplied by the initiator is conveyed as the Called SS-User reference.

Each component, when present, will contain a data element of the appropriately named type from the following definitions:

- CallingSSUserReference ::= SSAPAddress --of the initiator
- CommonReference ::= UTCTime
- AdditionalReferenceInformation ::= T61String
- SSAPAddress ::= T61String

The UTCTime field is used to differentiate between session connections initiated by the same ROS-user. It should not be used as a definitive statement of the connect request time.

The encoding of this field is not defined in this Technical Report; it requires clarification as explained in clause 4.3.

2. ROS may optionally use Session Layer addressing, that is, a session address passed in the Connect SPDU of the Session Layer. If session addressing is not used, the SSAP Addresses are passed down to (and then passed up from) the Transport Layer. They then have a one-to-one relationship with transport addresses.
3. The receiving ROS Entity may accept or reject the Session connection. The ROS-user may cause this rejection by the RO-BEGIN.response primitive. The Result parameter indicates which has occurred. Some qualification of the reason for the rejection may be present in the User Data parameter, see point 7 below. The Session provider may also reject the connection request under some circumstances, see ISO 8326.
4. The parameters Extended Control and Optimized Dialogue Transfer are set to not required. The remaining parameters are not used.
5. This parameter specifies the functional units to be used as listed in clause 6.3.1. Use of the "Half-duplex" functional unit is negotiated if two-way alternate Session dialogue is required. Otherwise two-way simultaneous Session dialogue is negotiated.

6. If use of the "Half-duplex" functional unit is requested the initiating ROS Entity will always request that it is the initial holder of the data token.
7. In the S-CONNECT.request, this parameter contains a (Presentation Layer) PConnect data element. In the S-CONNECT.response, it contains a PAccept or PRefuse data element depending upon the Result parameter. These data elements are defined as follows:

```
PConnect ::= SET {
  [0] IMPLICIT DataTransferSyntax,
  [1] IMPLICIT pUserData SET {
    [3] ConnectionData,
    applicationProtocol [4] IMPLICIT INTEGER --as defined in the Application--
                                     --Protocol--
    protocolVersion [5] IMPLICIT INTEGER OPTIONAL}}
```

```
PAccept ::= SET {
  [0] IMPLICIT DataTransferSyntax,
  [1] IMPLICIT pUserData SET {
    [2] ConnectionData }}
```

```
PRefuse ::= SET { [0] IMPLICIT RefuseReason }
```

```
DataTransferSyntax ::= SET { [0] IMPLICIT INTEGER {x.409(0) }}
```

The only value defined for the Data Transfer Syntax parameter refers to the Presentation Transfer Syntax specified in CCITT Rec. X.409.

```
ConnectionData ::= CHOICE { open [0] ANY } -- ANY is the User Data of the--
                                     --RO-BEGIN service primitives---
```

```
RefuseReason ::= INTEGER {
  busy (0), validationFailure (2), unacceptableDialogueMode (3) }
```

6.3.3 Data Transfer Phase - Action of Sending ROS Entity

Each OPDU is conveyed in the S-DATA.request. An OPDU is transferred as a single SSDU.

If the "Half-duplex" functional unit is used, only the holder of the token may issue this request.

6.3.3.1 S-DATA

Parameter	Req/Ind
SS-User Data	1

1. Maximum size is not restricted in this mapping.

6.3.4 Data Transfer Phase - Actions of Receiving ROS Entity

6.3.4.1 Responding to Problems

If the receiving ROS Entity detects a problem during receipt of an OPDU, it may issue an S-U-ABORT.request. Aborting the Session connection is the most severe action; this is described in clause 6.3.5.2.

6.3.4.2 Use of Half-duplex Session

If the use of the "Half-duplex" functional unit has been negotiated, the following applies:

- The ROS Entity that does not have the token can request the token if it has OPDUs to send, see clause 6.3.4.2.1.
- The ROS Entity that has the token may give the token, see clause 6.3.4.2.2.

Each OPDU is assigned a priority as defined in an Application Protocol Standard. The priority is specified as a parameter in the S-TOKEN.PLEASE.request.

6.3.4.2.1 S-TOKEN-PLEASE

Parameter	Req/Ind
Tokens	1
SS-User Data	2

1. The receiving ROS Entity will only request the data token.
2. The priority parameter is defined as follows:

Priority ::= INTEGER

The encoding of this field is not defined in this Technical Report; it requires clarification as explained in clause 4.3.

6.3.4.2.2 S-TOKEN-GIVE

Parameter	Req/Ind
Tokens	1

1. Only the data token is given.

6.3.5 Session Connection Termination Phase

6.3.5.1 Orderly Release

The ROS Entity issues an S-RELEASE.request in response to an RO-END.request, see clause 5.2.1.2. Upon receiving an S-RELEASE.indication, the ROS Entity issues an RO-END.indication. Upon receiving the RO-END.response, the ROS Entity issues an S-RELEASE.response. Upon receiving the S-RELEASE.confirmation, the ROS Entity issues an RO-END.confirmation.

If the "Half-duplex" functional unit has been negotiated, the ROS Entity may only issue the S-RELEASE.request if it holds the data token. If it does not hold the token, this is obtained by issuing an S-TOKEN-PLEASE.request, see clause 6.3.4.2.1, with the priority parameter value being that from the RO-END.request.

6.3.5.1.1 S-RELEASE

Parameter	Req/Ind	Resp/Conf
Result	-	1
SS-User Data	-	-
Re-use of Transport	2	-

1. The responding ROS Entity will always accept the release.
2. The transport connection need not be released with the session connection and may be re-used.

6.3.5.2 Abort Initiated by the ROS Entity

If the ROS Entity detects a serious problem it may issue an S-U-ABORT.request. It should provide a reason parameter for diagnostic purposes. The ROS-user is informed about this by the RO-REJECT-P primitive, see clause 5.5.2.7.

6.3.5.2.1 S-U-ABORT

Parameter	Req/Ind
SS-User Data	1

1. The SS-User Data parameter contains an Abort Information data element, defined as follows:

AbortInformation ::= SET {
[0] IMPLICIT AbortReason OPTIONAL }

AbortReason ::= INTEGER {
localSystemProblem (0),
invalidParameter (1), --reflectedParameter supplied--
temporaryProblem (3), -- the ROS cannot accept a session for a period of time--
protocolError (4) --ROS level protocol error-- }

6.3.5.3 Abort Initiated by Session Provider

The Session-provider may abort a session connection for any of a variety of reasons (for example, transport connection failure or local or remote provider problem), indicated by the reason parameter. The ROS-user is informed about this by the RO-REJECT-P primitive, see clause 5.5.2.7.

6.3.5.3.1 S-P-ABORT

Parameter	Ind
Reason	1

1. The following reason codes may be supplied:
 - Transport disconnect;
 - Protocol error.

(Blank page)

7. GUIDELINES FOR REFERENCING THIS TECHNICAL REPORT

Standards documents which reference this Technical Report and use its notation or concepts or mappings should include a statement which makes clear how these are to be used.

Some suggested text is as follows:

Either " ... uses the notation and concepts and allows any choice of the mappings defined in section 6."

Or " ... uses the notation and concepts and requires use of the particular mapping defined in clause 6.x of section 6."

(Blank page)

Appendix A

Bibliography

The following sources of background information which are not Standards, Recommendations or Technical Reports are referenced by this Technical Report.

- CCITT X-400 Series Implementor's Guide (Working Paper from CCITT Com. VII).
- CRISTIAN 82: F. Cristian: Robust Data Types. Acta Informatica 17, pp 365-397 (1982).
- CRISTIAN 84: F. Cristian: Correct and Robust Programs. IEEE Transactions on Software Engineering. Vol. SE-10, No 2, March 1984.
- BIRRELL 84: AD Birrell and BJ Nelson: Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, Vol. 2, No 1, February 1984.
- LISKOV 82: B. Liskov, R. Scheifler: Guardians and Actions: Linguistic support for robust distributable programs. Proceedings of Principles of Programming Conference, ACM - SIGPLAN 1982.
- GOLDBERG A. Goldberg and D. Robson: Smalltalk-80 - The Language and its Implementation. Addison Wesley 1983.

