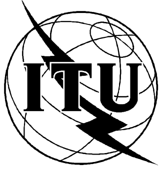


INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

X.880

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(07/94)

DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS

OSI APPLICATIONS – REMOTE OPERATIONS

INFORMATION TECHNOLOGY – REMOTE OPERATIONS: CONCEPTS, MODEL AND NOTATION

ITU-T Recommendation X.880

(Previously "CCITT Recommendation")

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. Some 179 member countries, 84 telecom operating entities, 145 scientific and industrial organizations and 38 international organizations participate in ITU-T which is the body which sets world telecommunications standards (Recommendations).

The approval of Recommendations by the Members of ITU-T is covered by the procedure laid down in WTSC Resolution No. 1 (Helsinki, 1993). In addition, the World Telecommunication Standardization Conference (WTSC), which meets every four years, approves Recommendations submitted to it and establishes the study programme for the following period.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC. The text of ITU-T Recommendation X.880 was approved on 1st of July 1994. The identical text is also published as ISO/IEC International Standard 13712-1.

NOTE

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

© ITU 1995

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the ITU.

ITU-T X-SERIES RECOMMENDATIONS
DATA NETWORKS AND OPEN SYSTEM COMMUNICATIONS
(February 1994)

ORGANIZATION OF X-SERIES RECOMMENDATIONS

Subject area	Recommendation series
PUBLIC DATA NETWORKS	
Services and Facilities	X.1-X.19
Interfaces	X.20-X.49
Transmission, Signalling and Switching	X.50-X.89
Network Aspects	X.90-X.149
Maintenance	X.150-X.179
Administrative Arrangements	X.180-X.199
OPEN SYSTEMS INTERCONNECTION	
Model and Notation	X.200-X.209
Service Definitions	X.210-X.219
Connection-mode Protocol Specifications	X.220-X.229
Connectionless-mode Protocol Specifications	X.230-X.239
PICS Proformas	X.240-X.259
Protocol Identification	X.260-X.269
Security Protocols	X.270-X.279
Layer Managed Objects	X.280-X.289
Conformance Testing	X.290-X.299
INTERWORKING BETWEEN NETWORKS	
General	X.300-X.349
Mobile Data Transmission Systems	X.350-X.369
Management	X.370-X.399
MESSAGE HANDLING SYSTEMS	X.400-X.499
DIRECTORY	X.500-X.599
OSI NETWORKING AND SYSTEM ASPECTS	
Networking	X.600-X.649
Naming, Addressing and Registration	X.650-X.679
Abstract Syntax Notation One (ASN.1)	X.680-X.699
OSI MANAGEMENT	X.700-X.799
SECURITY	X.800-X.849
OSI APPLICATIONS	
Commitment, Concurrency and Recovery	X.850-X.859
Transaction Processing	X.860-X.879
Remote Operations	X.880-X.899
OPEN DISTRIBUTED PROCESSING	X.900-X.999

CONTENTS

Page

1 Scope 1

2 Normative references 1
- 2.1 Identical Recommendations | International Standards 1
- 2.2 Paired Recommendations | International Standards equivalent in technical content 1
- 2.3 Additional references 2

3 Definitions 2
- 3.1 OSI reference model definitions 2
- 3.2 ASN.1 definitions 2
- 3.3 ROS definitions 2

4 Abbreviations 3

5 Conventions 3

6 ROS model 3

7 Realization of ROS 5

8 ROS concepts 6
- 8.1 Introduction 6
- 8.2 Operation 6
- 8.3 Error 7
- 8.4 Operation package 8
- 8.5 Connection package 8
- 8.6 Association contract 9
- 8.7 ROS-object class 10
- 8.8 Code 11
- 8.9 Priority 11

9 Generic ROS protocol 11
- 9.1 Introduction 11
- 9.2 ROS 11
- 9.3 Invoke 12
- 9.4 Return result 13
- 9.5 Return error 14
- 9.6 Reject 15
- 9.7 Reject Problem 17
- 9.8 Invoke id 17
- 9.9 No invoke id 17
- 9.10 Errors 17
- 9.11 Bind 18
- 9.12 Unbind 18

10 Useful definitions 18
- 10.1 Introduction 18
- 10.2 Empty bind 18
- 10.3 Empty unbind 18
- 10.4 Refuse 19
- 10.5 No-op 19
- 10.6 Forward 19
- 10.7 Reverse 19
- 10.8 Consumer performs 19
- 10.9 Supplier performs 20
- 10.10 All operations 20

Page

10.11 recode 20
10.12 switch 20
10.13 combine 21
10.14 ROS single abstract syntax 21
10.15 ROS consumer abstract syntax 21
10.16 ROS supplier abstract syntax 21

Annex A – ASN.1 modules 22

Annex B – Guidelines for the use of the notation 29
B.1 Examples of Operations and their Errors 29
B.2 Examples of Operation Packages and the use of switch{} 30
B.3 Examples of Bind and Unbind operations 31
B.4 Examples of Connection Packages 31
B.5 Example of an Association Contract 32
B.6 Examples of ROS-objects 32
B.7 Example of the use of Forward{} and Reverse{} 32
B.8 Examples of ConsumerPerforms{}, SupplierPerforms{} and AllOperations{} 33

Annex C – Migrating from the ROS macros 35
C.1 Introduction 35
C.2 Operation 35
C.3 Error 36
C.4 Bind 36
C.5 Unbind 36

Annex D – Assignment of object identifier values 37

Summary

This Recommendation | International Standard uses the Abstract Syntax Notation One (ASN.1) to define information object classes corresponding to the fundamental concepts of the Remote Operations Service (ROS). This, in turn, provides the notation that will allow application designers to specify instances of these classes. This Recommendation | International Standard also provides a collection of definitions for specifying the generic protocol between objects that communicate using remote operations. A number of definitions of general utility to designers of ROS-based applications is also provided.

Introduction

Remote operations (ROS) is a paradigm for interactive communication between objects. As such it can be used in the design and specification of distributed applications. The basic interaction involved is the invocation of an operation by one object (the invoker), its performance by another (the performer), possibly followed by a report of the outcome of the operation being returned to the invoker.

The concepts of ROS are abstract, and may be realized in many ways. For example, objects whose interactions employ ROS concepts may be separated by a software interface or by an OSI network.

This Recommendation | International Standard describes the concepts and model of ROS. It uses ASN.1 to specify information object classes corresponding to the fundamental concepts of ROS, such as operation and error. This in turn provides a notation so that designers can specify particular instances of those classes, e.g. particular operations and errors.

This Recommendation | International Standard provides a generic set of PDUs which can be used in realizing the ROS concepts between objects remote from one another. These PDUs are used in the OSI realization of ROS, which are specified in the companion Recommendations | International Standards to this one.

This Recommendation | International Standard also provides a number of definitions of general utility to designers of ROS-based applications.

Annex A forms an integral part of this Recommendation | International Standard.

Annexes B, C and D do not form an integral part of this Recommendation | International Standard.

INTERNATIONAL STANDARD

ITU-T RECOMMENDATION

INFORMATION TECHNOLOGY – REMOTE OPERATIONS: CONCEPTS, MODEL AND NOTATION

1 Scope

This Recommendation | International Standard specifies the Remote Operations Service (ROS) using the Abstract Syntax Notation (ASN.1) to define information object classes corresponding to the fundamental concepts of ROS. This, in turn, provides the notation that will allow application designers to specify particular instances of these classes.

This Recommendation | International Standard also provides a collection of definitions for specifying the generic protocol between objects that communicate using ROS concepts. These definitions are used in the companion Recommendations | International Standards to this one to provide the protocol data units, the service primitives and the application context definitions used in the OSI realization of ROS.

A number of definitions of general utility to designers of ROS-based applications is also provided.

No requirement is made for conformance to this Recommendation | International Standard.

2 Normative references

The following ITU-T Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Specification. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Specification are encouraged to investigate the possibility of applying the most recent editions of the Recommendations and Standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunications Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

- ITU-T Recommendation X.680 (1994) | ISO/IEC 8824-1:1994, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation.*
- ITU-T Recommendation X.681 (1994) | ISO/IEC 8824-2:1994, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification.*
- ITU-T Recommendation X.682 (1994) | ISO/IEC 8824-3:1994, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification.*
- ITU-T Recommendation X.683 (1994) | ISO/IEC 8824-4:1994, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications.*
- ITU-T Recommendation X.200 (1994) | ISO/IEC 7498-1:1994, *Information technology – Open Systems Interconnection – Basic Reference Model: The basic model.*
- ITU-T Recommendation X.881 (1994) | ISO/IEC 13712-2:1994, *Information technology – Remote Operations: OSI realizations – Remote Operations Service Element (ROSE) service definition.*
- ITU-T Recommendation X.882 (1994) | ISO/IEC 13712-3:1994, *Information technology – Remote Operations: OSI realizations – Remote Operations Service Element (ROSE) protocol specification.*

2.2 Paired Recommendations | International Standards equivalent in technical content

- CCITT Recommendation X.219 (1988), *Remote Operations: Model, notation and service definition.*

  ISO/IEC 9072-1:1989, *Information processing systems – Text communication – Remote Operations – Part 1: Model, notation and service definition.*
- CCITT Recommendation X.229 (1988), *Remote Operations: Protocol specification.*

  ISO/IEC 9072-2:1989, *Information processing systems – Text communication – Remote Operations – Part 2: Protocol specification.*

2.3 Additional references

– CCITT Recommendation X.407 (1988), *Message handling systems: Abstract service definition conventions.*

3 Definitions

3.1 OSI reference model definitions

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.200 | ISO/IEC 7498-1:

a) abstract syntax;

b) protocol data unit;

c) quality of service.

3.2 ASN.1 definitions

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.680 | ISO/IEC 8824-1:

a) (data) type;

b) (data) value.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.681 | ISO/IEC 8824-2:

a) field;

b) (information) object;

c) (information) object class;

d) (information) object set.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.682 | ISO/IEC 8824-3:

a) constraint;

b) exception value.

This Recommendation | International Standard makes use of the following terms defined in ITU-T Rec. X.683 | ISO/IEC 8824-4:

– parameterized.

3.3 ROS definitions

This Recommendation | International Standard defines the following terms:

3.3.1 **argument**: A data value accompanying the invocation of an operation.

3.3.2 **association**: A relationship between a pair of objects, serving as the context for the invocation and perfomance of operations.

3.3.3 **association contract**: A specification of the roles of a pair of communicating objects who may have an association with each other.

3.3.4 **asymmetrical**: Describing an operation package (or association contract), where the sets of operations which the two parties are capable of performing differ.

3.3.5 **connection package**: A specification of the roles of a pair of communicating objects in the dynamic establishment and release of associations between them.

3.3.6 **contract**: A set of requirements on one or more objects prescribing a collective behaviour.

3.3.7 **error**: A report of the unsuccessful performance of an operation.

3.3.8 linked operation: An operation invoked during the performance of another operation by the (latter's) performer and intended to be performed by the (latter's) invoker.

3.3.9 object: A model of (possibly a self-contained part of) a system, characterized by its initial state and its behaviour arising from external interactions over well-defined interfaces.

3.3.10 operation: A function that one object (the invoker) can request of another (the performer).

3.3.11 operation package: A collection of related operations used to specify the roles of a pair of communicating objects, each operation being invokable by one or both objects of the pair and performable by the partner.

3.3.12 parameter (of an error): A data value which may accompany the report of an error.

3.3.13 result: A data value which may accompany the report of the successful performance of an operation.

3.3.14 ROS-object: An object whose interactions with other objects are described using ROS concepts.

3.3.15 symmetrical: Describing an operation package (or assocation contract) in which both parties are capable of performing the same set of operations.

3.3.16 synchronous: A characteristic of an operation that, once invoked, its invoker cannot invoke another synchronous operation (with the same intended performer) until the outcome has been reported.

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
PDU	Protocol Data Unit
QOS	Quality of Service
RO (or ROS)	Remote Operations

5 Conventions

This Recommendation | International Standard employs ASN.1 to define:

a) Information object classes corresponding to the ROS concepts – These also provide notation with which designers of ROS applications can specify particular instances of those classes.

b) Particular information objects of those classes.

c) The PDUs of the generic ROS protocol.

d) Data types needed in these definitions.

Many of these definitions are parameterized, so that their users must supply actual parameters in order to complete them.

6 ROS model

Remote operations (ROS) is a paradigm for interactive communication between objects. Objects whose interactions are described and specified using ROS are **ROS-objects**. The basic interaction involved is the invocation of an operation by one ROS-object (the invoker) and its performance by another (the performer).

Completion of the performance of the operation (whether successfully or unsuccessfully) may lead to the return, by the performer to the invoker, of a report of its outcome. This is illustrated in Figure 1.

A report of the successful completion of an operation is a **result**; a report of unsuccessful completion an **error**.

During the performance of an operation, the performer can invoke **linked operations**, intended to be performed by the invoker of the original operation.

For correct interworking, certain properties of the operation must be known by both invoker and performer. The properties include:

whether reports are to be returned, and if so, which ones;

the types of the values, if any, to be conveyed with invocations of the operation or returns from it;

which operations, if any, can be linked to it;

the code value to be used to distinguish this operation from the others that might be invoked.

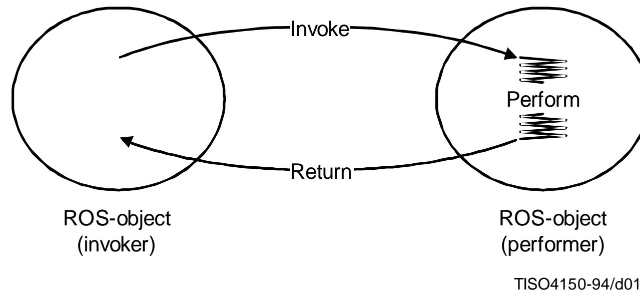


Figure 1 – Invocation, performance, and return of an operation

The interworking capabilities of (pairs of) ROS-objects of some **ROS-object class** are defined in terms of sets of related operations called **operation package**s. A package may be **symmetrical**, in which case it is defined by a single set of operations which each ROS-object in the pair can invoke (of the other). Alternatively, a package may be **asymmetrical**, in which case it is defined by two sets of operations, those which one ROS-object of the pair can invoke and those which the other can invoke. For the purpose of defining an asymmetrical package, the two ROS-objects are (arbitrarily) labelled the **consumer** and **supplier** respectively.

NOTE 1 – While these labels are, in general, arbitrary, it will often be the case that an intuitive assignment can be made, given that frequently in such a pair of objects one will be supplying a service which the other consumes.

A pair of ROS-objects must have an association between them to serve as a context for the invocation and performance of operations. Each such association is governed by an **association contract**. A contract is specified in terms of the set of packages which (collectively) determine the operations which can be invoked during the association. If the contract specification includes one or more asymmetrical packages then the contract is itself asymmetrical. For the purpose of specifying an asymmetrical association contract, the two ROS-objects which establish an association with each other are labelled the **initiator** and **responder**.

An association may be brought into and out of existence by “off-line” means. Alternatively, an association may be established and released dynamically. One option, described in this Recommendation | International Standard, by which an association may be established and released dynamically is, respectively, through the invocation and performance of special **bind** and **unbind** operations. The contract for associations of the latter variety includes a **connection package**, which includes the particular bind and unbind operations to be used.

NOTE 2 – The mechanism for the establishment and release of associations may also be done by other means described in other Recommendations | International Standards.

An association requires a relationship between the objects, the existence of which corresponds to the mutual agreement of the objects to the terms of some association contract.

NOTE 3 – This specification does not address the means by which these relationships are established or terminated.

In the foregoing, the only objects seen to be involved in an operation have been the invoker and performer. However, in general, the invoker and performer of an operation are not directly attached to one another, but are connected by some medium through which invocations and returns can be conveyed. This expanded view is illustrated in Figure 2.

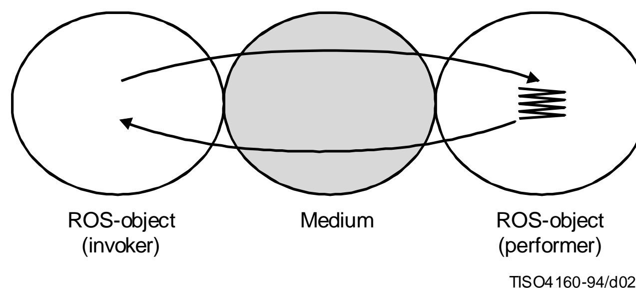


Figure 2 – Expanded view

The medium may introduce delay and the possibility of failure or inaccuracy both in the conveyance of invocations and returns, and in the establishment, release, and maintenance of associations. It may also introduce the possibility of the security of the association and its operations being threatened. The extent of these (together with other factors) are described as the quality-of-service (QOS).

Association contracts can now be seen as having three parties, the third party being the medium. The medium's obligation under the contract is to satisfy the QOS requirements.

NOTE 4 – In the future, target and minimum acceptable QOS requirements may form part of the specification of operations, operation packages and of the association contract itself directly. Different aspects of QOS are applicable to these different levels of specification.

7 Realization of ROS

A **realization** of ROS involves the definition of a suitable medium to convey invocations and returns between ROS-objects. Such a medium may, for example, comprise:

a) a message-passing or procedure calling capability allowing the invoker and performer of an operation to be implemented in separately developed software modules within a single computer system;

b) a communications capability, allowing the invoker and performer of an operation to be implemented in separate computer systems.

A realization may be general-purpose, in which case it can be employed to support any association contract. Others are special-purpose, accommodating only some particular contract(s).

Figure 3 depicts an approach to realizing ROS by communications means which is likely to be widely used.

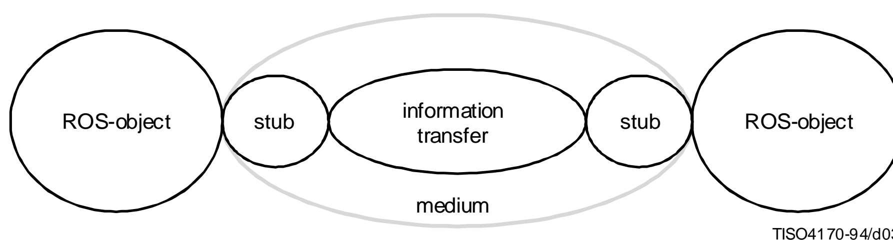


Figure 3 – Approach to communications realization of ROS

In this approach, the medium is composed of **stub** objects, one for each ROS-object, and an information transfer object. The stub object associated with each ROS-object appears to play the role of the partner ROS-object. However, it does not actually invoke or perform any operations, merely transforming invocations and returns into PDUs and vice versa, as appropriate. These PDUs are exchanged between the stubs by means of the information transfer object.

Thus, to invoke an operation, the invoker invokes the operation of the associated stub, which forms a PDU describing the invocation. The stub uses the information transfer capability to transfer the PDU to the other stub. The latter stub interprets the PDU and then invokes the appropriate operation of the ROS-object with which it is associated, the performer. When the operation has been performed, the performer conveys any return to its associated stub, which forms a PDU describing it. The stub then, uses the information transfer capability to transfer the PDU to the other stub. The latter stub interprets the PDU and then reports the return to the invoker.

Clause 9 defines a collection of suitable PDUs.

Various information transfer capabilities can be used in a ROS realization of this kind. Of particular importance are the information transfer capabilities of OSI. The pair of companion Recommendations | International Standards, ITU-T Rec. X.881 | ISO/IEC 13712-2 and ITU-T Rec. X.882 | ISO/IEC 13712 -3, describe a number of such realizations.

8 ROS concepts

8.1 Introduction

8.1.1 This clause defines the information object classes corresponding to the basic concepts of ROS, specifying the characteristics that objects of such classes have. The following information object classes are defined:

– `OPERATION` (describing operations);

– `ERROR` (describing errors);

– `OPERATION-PACKAGE` (describing operation packages);

– `CONNECTION-PACKAGE` (describing connection packages);

– `CONTRACT` (describing association contracts);

– `ROS-OBJECT-CLASS` (describing ROS-object classes).

8.1.2 The information object classes are defined using ASN.1. This provides notation which is available to designers of ROS applications for specifying particular instances of those classes. Designers are encouraged, but not obliged, to use this specification approach. If some other approach is employed, the resulting specification shall include or reference a description of how a valid use of the notation provided could be derived.

NOTE – A number of existing specifications employ the ASN.1 MACRO notation (which were defined in previous versions of this Recommendation | International Standard: see CCITT Rec. X.219 | ISO/IEC 9072-1) to specify operations, errors, and other classes of information object relevant to ROS. Annex C describes how the use of these macros can be transformed onto the notation provided. These macros should not be used for new applications.

8.2 Operation

8.2.1 An operation is a function that one object (the invoker) can request of another (the performer). The information object class `OPERATION`, to which all operations belong, is specified as follows, the various fields being described in 8.2.2 to 8.2.13:

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

8.2.2 The `&ArgumentType` field specifies the data type of the argument of the operation. If in some operation the field is omitted, then that operation takes no argument value.

8.2.3 The &argumentTypeOptional field, which can be present only if the &ArgumentType field is present, specifies if the data type of the operation argument may optionally be omitted. If this field is absent or takes the value FALSE, the value of the &ArgumentType cannot be omitted from the Invoke{ } PDU (see 9.3).

8.2.4 The &returnResult field specifies whether a result is returned in the event that the operation is performed successfully, taking the value TRUE if it is, and FALSE otherwise.

8.2.5 The &ResultType field specifies the data type of the value returned with the result of the operation. If it is omitted, then the operation returns no result value. It shall be omitted if the &returnResult field is FALSE.

8.2.6 The &resultTypeOptional field, which can be present only if the &ResultType field is present, specifies if the data type of the value returned as the result of performing the operation may optionally be omitted. If this field is absent or takes the value FALSE, the value of the &ResultType cannot be omitted from the ReturnResult{ } PDU (see 9.4).

8.2.7 The &Errors field specifies a set of errors, any one of which may be returned to report an unsuccessful performance of the operation. If this field is omitted, then unsuccessful performance of the operation is either not possible or not reported.

8.2.8 The &alwaysReturns field specifies whether the outcome of performing the operation is always returned, taking the value TRUE if it is and FALSE otherwise. If this field is set to TRUE, at least one of the &returnResult or &Errors field must be present.

8.2.9 The &Linked field, if present, specifies a set of operations, any of which may be invoked as linked operations during the performance of the operation. If this field is omitted, then no operations may be linked to an invocation of this one.

8.2.10 The &synchronous field specifies whether or not the operation is synchronous, taking the value TRUE if it is, and FALSE otherwise. If the &returnResult field is FALSE, this field shall also take the value FALSE. When the &synchronous field is set to TRUE, it implies that no other *synchronous* operation may be invoked by this side until this operation has returned.

NOTE – The combination of the &alwaysReturns and the &synchronous fields replaces the earlier concept of "operation classes" defined in CCITT Rec. X.219 | ISO/IEC 9072-1.

8.2.11 The &InvokePriority field specifies the permitted Priority (see 8.9) levels at which this operation can be invoked.

8.2.12 The &ResultPriority field specifies the permitted Priority (see 8.9) levels at which the result of this operation can be returned. If the &returnResult field is FALSE, this field shall be omitted.

8.2.13 The &operationCode field, if present, specifies the Code value (see 8.8) which is used to identify this operation, e.g. when it is to be invoked.

NOTE – An operation which does not have an &operationCode cannot be invoked using the Invoke{} PDU (see 9.3). In practice, therefore, all operations should have &operationCodes assigned except when intended for use in some special circumstances, e.g. as a bind operation.

8.3 Error

8.3.1 An error is a report of the unsuccessful performance of an operation. The information object class ERROR, to which all errors belong, is specified as follows, the various fields being described in 8.3.2 to 8.3.5:

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

8.3.2 The &ParameterType field specifies the data type of the parameter of the error. If in some error the field is omitted, then that error takes no parameter value.

8.3.3 The ¶meterTypeOptional field, which can be.present only if the &ParameterType field is present, specifies if the data type of the value returned as the parameter qualifying the error may optionally be present. If this field is absent or takes the value FALSE, the value of the &ParameterType cannot be omitted from the ReturnError{} PDU (see 9.5).

8.3.4 The &ErrorPriority field specifies the permitted Priority (see 8.9) levels at which the error can be returned.

8.3.5 The &errorCode field, if present, specifies the Code value (see 8.8) which is used to identify this error, e.g. when it is returned.

NOTE – An error which does not have an &errorCode cannot be returned using the ReturnError{} PDU defined below. In practice, therefore, all errors should have their &errorCode fields assigned except when intended for use in some special circumstances, e.g. as a bind error.

8.4 Operation package

8.4.1 An operation package is a specification of the roles of a pair of communicating objects, in terms of the operations which they can invoke of each other. Where the package is asymmetrical, the terms "consumer" and "supplier" are employed for the two objects involved. The information object class OPERATION-PACKAGE, to which all such packages belong, is specified as follows, the various fields being described in 8.4.2 to 8.4.7:

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

8.4.2 The &Both field specifies a set of OPERATIONs which both objects shall be capable of performing. This field may be omitted.

8.4.3 The &Consumer field specifies a set of OPERATIONs which one of the objects, known as the consumer, shall be capable of performing. This field may be omitted, in which case the consumer shall only be capable of performing the operations specified by the &Both field.

8.4.4 The &Supplier field specifies a set of OPERATIONs which one of the objects, known as the supplier, shall be capable of performing. This field may be omitted, in which case the supplier shall only be capable of performing the operations specified by the &Both field.

NOTE – A parameterized operation package, switch{}, is provided (see 10.12) to allow one operation package to be derived by switching the roles of some other.

8.4.5 The &id field, if present, specifies the OBJECT IDENTIFIER value which is used to identify this package e.g. if it is being announced or negotiated.

NOTE – A package which does not have an &id cannot be announced or negotiated.

8.4.6 All operations included (directly or indirectly) by the &Both, &Consumer, and &Supplier fields shall have distinct &operationCode values.

8.4.7 All errors included in the &Errors field of all of the operations included (see 8.4.6), shall have distinct &errorCode values.

8.5 Connection package

8.5.1 A connection package is a specification of the operations and QOS involved in dynamic establishment and release of an association. The connection package shall be specified only if the bind and the unbind operations are used to, respectively, dynamically establish and release an association. The information object class CONNECTION-

PACKAGE, to which all such connection packages belong, is specified as follows, the various fields being described in 8.5.2 to 8.5.6:

```
CONNECTION-PACKAGE ::= CLASS
{
        &bind                   OPERATION DEFAULT emptyBind,
        &unbind                 OPERATION DEFAULT emptyUnbind,
        &responderCanUnbind     BOOLEAN DEFAULT FALSE,
        &unbindCanFail          BOOLEAN DEFAULT FALSE,
        &id                     OBJECT IDENTIFIER UNIQUE OPTIONAL
}
WITH SYNTAX
{
        [BIND                   &bind]
        [UNBIND                 &unbind]
        [RESPONDER UNBIND       &responderCanUnbind]
        [FAILURE TO UNBIND      &unbindCanFail]
        [ID                     &id]
}
```

8.5.2 The &bind field specifies an OPERATION which is to be performed as part of the establishment of an association. Such an operation must have its &returnResult and &alwaysReturns fields set to TRUE, and have a single error, present in the &Errors field. If the association is successfully established, the bind operation will return a result. If the association is not successfully established, the bind operation will report its error. If the &bind field is not explicitly included, the connection package will include the emptyBind operation (see 10.2).

8.5.3 The &unbind field specifies an OPERATION which is to be performed as part of the release of an association. Such an operation must have its &returnResult and &alwaysReturns fields set to TRUE, and must have a single error, defined by the &Errors field. If the association is successfully released, the unbind operation will return a result. If the association is not successfully released, the unbind operation will not return a result, and will report its error. If the &unbind field is not explicitly included, the connection package will include the emptyUnbind operation (see 10.3).

8.5.4 The &responderCanUnbind field indicates whether or not the association responder (as well as the initiator) can invoke the &unbind operation.

8.5.5 The &unbindCanFail field indicates whether or not it is possible for the association to still exist after the &unbind operation has signalled an error.

8.5.6 The &id field, if present, specifies the OBJECT IDENTIFIER value which is used to identify this connection package e.g. if it is being announced or negotiated.

NOTE – A connection package which does not have an &id cannot be announced or negotiated.

8.6 Association contract

8.6.1 An association contract is a specification of the roles of a pair of communicating objects who may establish an association with each other. The information object class CONTRACT, to which all such contracts belong, is specified as follows, the various fields being described in 8.6.2 to 8.6.6:

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

8.6.2 The presence of the `&connection` field indicates that the associations governed by this association contract are dynamically established and released by means of, respectively, the bind and the unbind operations specified as a part of a connection package, and the contents of the field specify the connection package involved.

8.6.3 The `&OperationsOf` field, if present, specifies one or more operation packages applicable while the association exists, and which are either symmetrical or in which the initiator of the association can play both consumer and supplier roles.

8.6.4 The `&InitiatorConsumerOf` field, if present, specifies one or more operation packages applicable while the association exists and in which the initiator of the association shall be considered to play the role of consumer.

NOTE – A parameterized operation package, `switch{}`, is provided (see 10.12) to allow one operation package to be derived by switching the roles of some other.

8.6.5 The `&InitiatorSupplierOf` field, if present, specifies one or more operation packages applicable while the association exists and in which the initiator of the association shall be considered to play the role of supplier.

8.6.6 The `&id` field, if present, specifies the `OBJECT IDENTIFIER` value which is used to identify this association contract, e.g. if it is being announced or negotiated.

NOTE – An association contract which does not have an `&id` cannot be announced or negotiated.

8.7 ROS-object class

8.7.1 A ROS-object class defines the capabilities of a set of ROS-objects which have in common their ability to interact with other ROS-objects using a particular set of contracts. The information object class `ROS-OBJECT-CLASS` is specified as follows, the various fields being described in 8.7.2 to 8.7.6:

```
ROS-OBJECT-CLASS ::= CLASS
{
        &Is                         ROS-OBJECT-CLASS OPTIONAL,
        &Initiates                  CONTRACT OPTIONAL,
        &Responds                   CONTRACT OPTIONAL,
        &InitiatesAndResponds       CONTRACT OPTIONAL,
        &id                         OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX
{
        [IS                         &Is]
        [BOTH                       &InitiatesAndResponds]
        [INITIATES                  &Initiates]
        [RESPONDS                   &Responds]]
        ID                          &id
}
```

8.7.2 The `&Is` field, if present, specifies a set of ROS-object classes which are superclasses of the class being defined. ROS-objects of the latter class are capable of supporting all of the contracts implied by their membership of each of the specified superclasses, as well as those explicitly present in the fields defined in 8.7.3 to 8.7.5 below.

8.7.3 The `&InitiatesAndResponds` field specifies a set of `CONTRACT`s for which ROS-objects of the class shall be capable of acting as initiator and responder. This field may be omitted.

8.7.4 The `&Initiates` field specifies a set of `CONTRACT`s for which ROS-objects of the class shall be capable of playing the initiator. This field may be omitted, in which case the ROS-objects shall only be capable of initiating the association contracts specified by the `&InitiatesAndResponds` field.

8.7.5 The `&Responds` field specifies a set of `CONTRACT`s for which object of the class shall be capable of playing the responder. This field may be omitted, in which case the object shall only be capable of responding to the association contracts specified by the `&InitiatesAndResponds` field.

8.7.6 The `&id` field specifies the `OBJECT IDENTIFIER` value which is used to identify this object class, e.g. if it is being announced or negotiated.

8.8 Code

The `Code` type provides the values for the `&operationCode` fields of operations and the `&errorCode` fields of errors. It is specified as follows:

```
Code ::= CHOICE
{
        local       INTEGER,
        global      OBJECT IDENTIFIER
}
```

8.9 Priority

The `Priority` type is specified as follows:

```
Priority ::= INTEGER (0..MAX)
```

This parameter defines the priority assigned to the transfer of the corresponding invocation (of an operation) or its return with respect to the other invocations (and their returns) to be exchanged between the two ROS-objects.

The lower the value, the higher the priority.

9 Generic ROS protocol

9.1 Introduction

This clause provides a collection of definitions which can be used to specify protocols realizing the ROS concepts. The primary definitions are:

a) a parameterized set of PDUs for invocations and returns of operations (`ROS{ }`);

b) a parameterized PDU for the invocation and return of a bind operation (`Bind{ }`);

c) a parameterized PDU for the invocation and return of an unbind operation (`Unbind{ }`).

In addition, there are a number of secondary definitions which these rely on.

9.2 ROS

9.2.1 The parameterized type `ROS{ }` provides a basis for the definition of an abstract syntax containing PDUs for the invocation of operations, for the returning of results and errors, and for the rejection of invalid PDUs. It is specified as follows:

```
ROS {InvokeId:InvokeIdSet, OPERATION:Invokable, OPERATION:Returnable} ::= CHOICE
{
        invoke          [1]     Invoke {{InvokeIdSet}, {Invokable}},
        returnResult    [2]     ReturnResult {{Returnable}},
        returnError     [3]     ReturnError {{Errors{{Returnable}}}},
        reject          [4]     Reject
}
(CONSTRAINED BY { -- must conform to the above definition -- }
! RejectProblem : general-unrecognizedPDU)
```

NOTE – In an actual realization, the use of encoding rules which are not self-identifying and self-delimiting may not permit the distinction between the outer-level and the inner-level constraint violations.

9.2.2 The ASN.1 parameters which must be supplied to produce a fully-defined type are as follows:

a) `InvokeIdSet` – This value set of type `InvokeId` defines the available values for identification of invocations and thus for correlation of later reports. If in some realization the correlation function can be carried out by other means, then it can be set to the value set `NoInvokeId` (see 9.9).

NOTES

1 The application designer may choose to leave this parameter as an unconstrained `INTEGER`, or provide the exact range, or propogate it as one of the parameters of the abstract syntax (see 10.14, 10.15 and 10.16).

2 An OSI realization does not support the value `noInvokeId` (see 9.9.1) for the `Invoke`, `ReturnResult`, and `ReturnError` PDUs.

b) The `Invokable` parameter specifies an object set that describes those operations which may be invoked;

c) The `Returnable` parameter specifies an object set that describes those operations for which responses may need to be generated.

NOTE – This parameter is provided to highlight the asymmetry of the abstract syntax (see 10.15 and 10.16). It can be derived from the `Invokable` information object set. Operations in the `Invokable` information object set whose `&alwaysReturns` field is not `FALSE` shall be included in this parameter.

9.2.3 If the receiver of a PDU does not recognize it as one of those defined, a `Reject{}` PDU is generated whose `problem` component takes the exception value: `general-unrecognizedPDU`.

9.3 Invoke

9.3.1 The parameterized type `Invoke{}` provides a PDU for the invocation of operations. It is specified as follows:

```
Invoke {InvokeId:InvokeIdSet, OPERATION:Operations} ::= SEQUENCE
{
    invokeId      InvokeId     (InvokeIdSet)
                               (CONSTRAINED BY  {-- must be unambiguous --}
                               ! RejectProblem          : invoke-duplicateInvocation),
    linkedId      CHOICE       {
                                   present   [0]      IMPLICIT present < InvokeId,
                                   absent    [1]      IMPLICIT NULL
                               }
                               (CONSTRAINED BY  {-- must identify an outstanding operation  --}
                               ! RejectProblem          : invoke-unrecognizedLinkedId)
                               (CONSTRAINED BY  {-- which has one or more linked operations--}
                               ! RejectProblem          : invoke-linkedResponseUnexpected)
                                   OPTIONAL,
    opcode        OPERATION.&operationCode
                               ({Operations}
                               !  RejectProblem         : invoke-unrecognizedOperation)
    argument      OPERATION.&ArgumentType
                               ({Operations}   {@opcode}
                               ! RejectProblem          : invoke-mistypedArgument)
                                   OPTIONAL
}
(CONSTRAINED BY  {-- must conform to the above definition --}
! RejectProblem          : general-mistypedPDU)
(
    WITH COMPONENTS
    {...,
        linkedId  ABSENT
    }
|   WITH COMPONENTS
    {...,
        linkedId  PRESENT,
        opcode
        (CONSTRAINED BY {-- must be in the &Linked  field of the associated operation --}
        ! RejectProblem          : invoke-unexpectedLinkedOperation)
    }
)
```

NOTE – In an actual realization, the use of encoding rules which are not self-identifying and self-delimiting may not permit the distinction between the outer-level and the inner-level constraint violations.

9.3.2 The ASN.1 parameters which must be supplied to produce a fully-defined type are as follows:

a) `InvokeIdSet` – This value set of type `InvokeId` defines the available values for identification of invocations and thus for correlation of later reports [see 9.2.2 a)].

b) `Operations` – Those which may be invoked.

9.3.3 The resulting PDU has up to four components, as follows:

a) `invokeId` – This component identifies the particular invocation. It shall be one of those allowed by the `InvokeIdSet` parameter. The `invokeId` shall not be one which is already in use (the rule for determining this is a characteristic of the specific mapping); otherwise, a `Reject{}` PDU (see 9.6) shall be generated, whose `problem` component takes the exception value: `invoke-duplicateInvocation`.

b) `linkedId` – This component, if present, indicates that this invocation is linked to a previous invocation in the opposite direction. It shall be the `InvokeId` of a previous invocation for which no outcome has been reported; if it is not, then a `Reject{}` PDU shall be generated, whose `problem` component takes the exception value:`invoke-unrecognizedLinkedID`. The previous invocation shall be of an operation which allows linked operations; if not, a `Reject{}`PDU shall be generated, whose `problem` component will take the exception value: `invoke-linkedResponseUnexpected.`

NOTE – For an OSI realization, the value `absent:NULL` for the `linkedID` is not used.

c) `opcode` – This component shall identify, by means of its `&operationCode,` one of the `Operations`; if not, then a `Reject{}` PDU shall be generated, whose `problem` component takes the exception value: `invoke-unrecognizedOperation.` If the `linkedId` component is present, then the previous invocation shall be of an operation which allowed this particular operation to be linked to it; if not, a `Reject{}` PDU shall be generated, whose `problem` component takes the exception value: `invoke-unexpectedLinkedOperation.`

d) `argument` – This component shall be present and of the type forming the `&ArgumentType` field of the operation identified by the `opcode` component, unless that field is omitted or the `&argumentTypeOptional` field omitted. If this condition is not met, then a `Reject {}` PDU shall be generated, whose `problem` component takes the exception value: `invoke-mistypedArgument.`

9.3.4 If the receiver of such a PDU finds it to be mistyped, a `Reject{}` PDU is generated whose `problem` component takes the exception value: `general-mistypedPDU`.

9.4 Return result

9.4.1 The parameterized type `ReturnResult{}` provides a PDU for reporting the successful performance of operations. It is specified as follows:

```
ReturnResult {OPERATION:Operations}::= SEQUENCE
{
        invokeId        InvokeId
                                (CONSTRAINED BY   {-- must be that for an outstanding operation  --}
                                ! RejectProblem         : returnResult-unrecognizedInvocation)
                                (CONSTRAINED BY   {-- which returns a result --}
                                ! RejectProblem         : returnResult-resultResponseUnexpected),
        result          SEQUENCE
        {
        opcode          OPERATION.&operationCode
                                ({Operations})(CONSTRAINED BY  {-- identified by invokeId --}
                                ! RejectProblem         : returnResult-unrecognizedInvocation),
        result          OPERATION.&ResultType ({Operations} {@opcode}
                                ! RejectProblem         : returnResult-mistypedResult)
        }
                        OPTIONAL
}
(CONSTRAINED BY  {-- must conform to the above definition --}
! RejectProblem         : general-mistypedPDU)
```

NOTE – In an actual realization, the use of encoding rules which are not self-identifying and self-delimiting may not permit the distinction between the outer-level and the inner-level constraint violations.

9.4.2 A single ASN.1 parameter must be supplied to produce a fully-defined type, namely `Operations`, those whose result may be returned.

9.4.3 The resulting PDU has up to three components, as follows:

a) `invokeId` – This component identifies the particular invocation whose successful performance is being reported. It shall be the `InvokeId` of a previous invocation for which no outcome has yet been reported; if it is not, then a `Reject{} PDU` shall be generated, whose `problem` component takes the exception value: `returnResult-unrecognizedInvocation.` The previous invocation shall be of an operation which returns a result; if not, then a `Reject{}` PDU shall be generated, whose `problem` component takes the exception value: `returnResult-resultResponseUnexpected`.

b) opcode – This component, present if and only if the result component is present, shall identify, by means of its &operationCode, one of the Operations, specifically, that indicated by the invokeId; if not, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnResult-unrecognizedInvocation.

c) result – This component shall be present and of the type forming the &ResultType field of the operation identified by the opcode component, unless that field is omitted, in which case this component shall itself be omitted. If this condition is not met, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnResult-mistypedResult.

9.4.4 If the receiver of such a PDU finds it to be mistyped, a Reject{} PDU is generated whose problem component takes the exception value: general-mistypedPDU.

9.5 Return error

9.5.1 The parameterized type ReturnError{} provides a PDU for reporting the unsuccessful performance of operations. It is specified as follows:

```
ReturnError {ERROR:Errors} ::= SEQUENCE
{
        invokeId        InvokeId
                                (CONSTRAINED BY   {-- must be that for an outstanding operation --}
                                ! RejectProblem   : returnError-unrecognizedInvocation)
                                (CONSTRAINED BY   {-- which returns an error --}
                                ! RejectProblem   : returnError-errorResponseUnexpected),
        errcode         ERROR.&errorCode
                                ({Errors}
                                ! RejectProblem   : returnError-unrecognizedError)
                                (CONSTRAINED BY   {-- must be in the &Errors field of the associated operation --}
                                ! RejectProblem   : returnError-unexpectedError),
        parameter       ERROR.&ParameterType
                                ({Errors}{@errcode}
                                ! RejectProblem   : returnError-mistypedParameter)
                        OPTIONAL
}
(CONSTRAINED BY  { -- must conform to the above definition -- }
! RejectProblem         : general-mistypedPDU)
```

NOTE – In an actual realization, the use of encoding rules which are not self-identifying and self-delimiting may not permit the distinction between the outer-level and the inner-level constraint violations.

9.5.2 A single ASN.1 parameter must be supplied to produce a fully-defined type, namely Errors, which is the set of errors for those operations whose errors may be returned (see 9.10).

9.5.3 The resulting PDU has up to three components, as follows:

a) invokeId – This component identifies the particular invocation whose unsuccessful performance is being reported. It shall be the InvokeId of a previous invocation for which no outcome has yet been reported; if it is not, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnError-unrecognizedInvocation. The previous invocation shall be of an operation which returns an error; if not, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnError-errorResponseUnexpected.

b) errcode – This component shall identify, by means of its &errorCode, one of the errors of the Errors; if not, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnError-unrecognizedError. The error shall be one of those appearing in the &Errors field of the operation identified by the invokeId; if not, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnError-unexpectedError.

c) parameter – This component shall be present and of the type forming the &ParameterType field of the error identified by the errcode component, unless that field is omitted, in which case this component shall itself be omitted. If this condition is not met, then a Reject{} PDU shall be generated, whose problem component takes the exception value: returnError-mistypedParameter.

9.5.4 If the receiver of such a PDU finds it to be mistyped a Reject{} PDU is generated whose problem component takes the exception value: general-mistypedPDU.

9.6 Reject

9.6.1 The type `Reject` provides a PDU for reporting erroneous use of the other `ROS{}` PDUs. It is specified as follows:

```
Reject ::= SEQUENCE
{
        invokeId        InvokeId,
        problem         CHOICE
            {
                        general         [0]     GeneralProblem,
                        invoke          [1]     InvokeProblem,
                        returnResult    [2]     ReturnResultProblem,
                        returnError     [3]     ReturnErrorProblem
            }
}
(CONSTRAINED BY  {-- must conform to the above definition --}
! RejectProblem         : general-mistypedPDU)
```

NOTE – In an actual realization, the use of encoding rules which are not self-identifying and self-delimiting may not permit the distinction between the outer-level and the inner-level constraint violations.

9.6.2 The resulting PDU has two components, as follows:

a) `invokeId` – This component is the `InvokeId` appearing in the PDU being rejected, except that if the `invokeId` cannot be determined, then the value `noInvokeId` (see 9.9) is sent instead.

b) `problem` – This component identifies the particular `problem` with the rejected PDU. There are four possible categories, as specified in 9.6.3 to 9.6.6.

9.6.3 A `GeneralProblem` is a fundamental problem with the form or structure of a `ROS{}` PDU. The possibilities are specified as follows:

```
GeneralProblem ::=  INTEGER
{
        unrecognizedPDU (0),
        mistypedPDU (1),
        badlyStructuredPDU (2)
}
```

and indicate the following:

a) `unrecognizedPDU` – The tag of the PDU indicates that it is not one of the alternatives allowed by 9.2;

b) `mistypedPDU` – The structure of the PDU does conform to the appropriate definition;

c) `badlyStructuredPDU` – The structure of the PDU cannot be determined based on the expected abstract syntax.

NOTE – In some mappings, these problems will be identified and handled within the communications infrastructure.

9.6.4 An `InvokeProblem` indicates that some component of an `Invoke{}` PDU was erroneous. The possibilities are specified as follows:

```
InvokeProblem ::=  INTEGER
{
        duplicateInvocation (0),
        unrecognizedOperation (1),
        mistypedArgument (2),
        resourceLimitation (3),
        releaseInProgress (4),
        unrecognizedLinkedId (5),
        linkedResponseUnexpected (6),
        unexpectedLinkedOperation (7)
}
```

and indicate the following:

a) `duplicateInvocation` [see 9.3.3 a)];

b) `unrecognizedOperation` [see 9.3.3 c)];

c) `mistypedArgument` [see 9.3.3 d)];

d) `resourceLimitation` – The intended performer is not willing to perform the operation due to a resource limitation;

e) `releaseInProgress` – The intended performer is not willing to perform the operation because it is about to release the association;

f) `unrecognizedLinkedId` [see 9.3.3 b)];

g) `linkedResponseUnexpected` [see 9.3.3 b)];

h) `unexpectedLinkedOperation` [see 9.3.3 c)].

9.6.5 A `ReturnResultProblem` indicates that some component of a `ReturnResult{}` PDU was erroneous. The possibilities are specified as follows:

```
ReturnResultProblem ::= INTEGER
{
        unrecognizedInvocation (0),
        resultResponseUnexpected (1),
        mistypedResult (2)
}
```

and indicate the following:

a) `unrecognizedInvocation` [see 9.4.3 a)];

b) `resultResponseUnexpected` [see 9.4.3 a)];

c) `mistypedResult` [see 9.4.3 b) and 9.4.3 c)].

9.6.6 A `ReturnErrorProblem` indicates that some component of a `ReturnError{}` PDU was erroneous. The possibilities are specified as follows:

```
ReturnErrorProblem ::= INTEGER
{
        unrecognizedInvocation (0),
        errorResponseUnexpected (1),
        unrecognizedError (2),
        unexpectedError (3),
        mistypedParameter (4)
}
```

and indicate the following:

a) `unrecognizedInvocation` [see 9.5.3 a)];

b) `errorResponseUnexpected` [see 9.5.3 a)];

c) `unrecognizedError` [see 9.5.3 b)];

d) `unexpectedError` [see 9.5.3 b)];

c) `mistypedParameter` [see 9.5.3 c)].

9.6.7 If the receiver of such a PDU finds it to be mistyped, no new `Reject{}` PDU shall be generated.

9.7 Reject Problem

The integer `RejectProblem` describes the error code value generated in the event of some type or constraint in a PDU definition being violated.

```
RejectProblem ::= INTEGER
{
        general-unrecognizedPDU (0),
        general-mistypedPDU (1),
        general-badlyStructuredPDU (2),
        invoke-duplicateInvocation (10),
        invoke-unrecognizedOperation (11),
        invoke-mistypedArgument (12),
        invoke-resourceLimitation (13),
        invoke-releaseInProgress (14),
        invoke-unrecognizedLinkedId (15),
        invoke-linkedResponseUnexpected (16),
        invoke-unexpectedLinkedOperation (17),
        returnResult-unrecognizedInvocation (20),
        returnResult-resultResponseUnexpected (21),
        returnResult-mistypedResult (22),
        returnError-unrecognizedInvocation (30),
        returnError-errorResponseUnexpected (31),
        returnError-unrecognizedError (32),
        returnError-unexpectedError (33),
        returnError-mistypedParameter (34)
}
```

9.7.1 The reaction to the signalling of an error is the sending of a `Reject{}` PDU. Where the error signalled is denoted by the `RejectProblem` identifier "α-β" for some strings α and β, the `problem` component of the `Reject{}` PDU shall take the value "α:β".

9.8 Invoke id

9.8.1 The `InvokeId` type defines the values that may be used to identify a particular invocation of an operation. It is specified as follows:

```
InvokeId ::= CHOICE
{
        present   INTEGER,
        absent    NULL
}
```

9.8.2 Where the `InvokeId` is `present`, it is a value of type `INTEGER`. When it is `absent`, a `NULL` value is used in its place.

9.9 No invoke id

9.9.1 `noInvokeId` is the value of `InvokeId` used when an `INTEGER` value is either not needed or not available. It is specified as follows:

```
noInvokeId InvokeId ::= absent:NULL
```

9.9.2 `NoInvokeId` is a value set of type `InvokeId` consisting only of the value `noInvokeId`. It is specified as follows:

```
NoInvokeId InvokeId ::= {noInvokeId}
```

9.10 Errors

The parameterized error set `Errors{}`, given a set of `Operations` as its ASN.1 parameter, is the set of all errors in the `&Errors` fields of those `Operations`. It is specified as follows:

```
Errors {OPERATION:Operations} ERROR ::= {Operations.&Errors}
```

9.11 Bind

The parameterized type `Bind{}`, given a single bind `operation` as its ASN.1 parameter, provides PDUs for the invocation of that operation, for the returning of the result of that operation, and for the reporting of an error. It is specified as follows:

```
Bind {OPERATION:operation} ::= CHOICE
{
        bind-invoke   [16]   OPERATION.&ArgumentType({operation}),
        bind-result   [17]   OPERATION.&ResultType({operation}),
        bind-error    [18]   OPERATION.&Errors.&ParameterType({operation})
}
```

9.12 Unbind

The parameterized type `Unbind{}`, given a single unbind `operation` as its ASN.1 parameter, provides PDUs for the invocation of that operation for the returning of the result of that operation, and for the reporting of an error. It is specified as follows:

```
Unbind {OPERATION:operation} ::= CHOICE
{
        unbind-invoke   [19]   OPERATION.&ArgumentType({operation}),
        unbind-result   [20]   OPERATION.&ResultType({operation}),
        unbind-error    [21]   OPERATION.&Errors.&ParameterType({operation})
}
```

10 Useful definitions

10.1 Introduction

This clause provides a collection of definitions which may prove useful to designers of ROS-based applications. They include:

a) generally useful operations, (`emptyBind`, `emptyUnbind`, `no-op`), and their associated errors;

b) parameterized definitions which deliver the sets of operations involved in some operation package (`ConsumerPerforms{}`, `SupplierPerforms{}`, `AllOperations{}`) and some auxiliary definitions;

c) a parameterized definition which allows one operation to be derived from another by changing the operation code (`recode{}`);

d) parameterized definitions which allow operation packages to be defined either by switching the roles of another, or by combining several others (`switch{}`, `combine{}`);

e) parameterized types which can be used to define abstract syntaxes corresponding to an operation package (`ROS-SingleAS{}`, `ROS-ConsumerAS{}`, `ROS-SupplierAS{}`).

10.2 Empty bind

The `emptyBind` operation provides the simplest bind operation, which stands as the default should a bind operation not be explicitly specified for some connection package. This operation has no argument or result values, and has a single possible error, `refuse` (see 10.4), corresponding to refusal of the association. The operation is synchronous, meaning that no synchronous operations can be invoked until the result of the bind operation has been returned. The `emptyBind` operation is specified as follows:

```
emptyBind OPERATION ::= {ERRORS {refuse} SYNCHRONOUS TRUE}
```

10.3 Empty unbind

The `emptyUnbind` operation provides the simplest unbind operation, which stands as the default should an unbind operation not be explicitly specified for some connection package. This operation has no argument, result or error values. The operation is synchronous, meaning that the unbind operation can not be invoked until the result of any outstanding synchronous operation has been returned.The `emptyUnbind` operation is specified as follows:

```
emptyUnbind OPERATION ::= {SYNCHRONOUS TRUE}
```

10.4 Refuse

The `refuse` error provides for a refusal of some request, without any reason being provided. It is specified as follows:

```
refuse ERROR ::= {CODE  local:-1}
```

10.5 No-op

10.5.1 The `no-op` operation does nothing. It is specified as follows:

```
no-op OPERATION ::=
{
        ALWAYS RESPONDS FALSE
        CODE  local:-1
}
```

10.5.2 The operation does not return.

10.6 Forward

The parameterized operation set `Forward{}`, given an `OperationSet` as its ASN.1 parameter, is the expanded operation set formed by adding any operations indirectly linked to operations in the set, and with the same directionality. It is specified as follows:

```
Forward {OPERATION:OperationSet} OPERATION ::=
{
        OperationSet |
        OperationSet.&Linked.&Linked |
        OperationSet.&Linked.&Linked.&Linked.&Linked
}
```

It is assumed that there are no operations in the fifth or later level of linkage which are not also present at earlier levels. If for some `OperationSet` this assumption is not valid, then some other method of identifying the full set of implied operations would be required.

10.7 Reverse

The parameterized operation set `Reverse{}`, given an `OperationSet` as its ASN.1 parameter is that consisting of all operations directly or indirectly linked to operations in the set, and with opposite directionality. It is specified as follows:

```
Reverse {OPERATION:OperationSet} OPERATION ::= {Forward{{OperationSet.&Linked}}}
```

It is assumed that there are no operations in the sixth or later level of linkage which are not also present at earlier levels. If for some `OperationSet` this assumption is not valid, then some other method of identifying the full set of implied operations would be required.

10.8 Consumer performs

The parameterized operation set `ConsumerPerforms{}`, given an operation package `package` as its ASN.1 parameter, is that which contains all operations which the consumer needs to be capable of performing. It is specified as follows:

```
ConsumerPerforms {OPERATION-PACKAGE:package} OPERATION ::=
{
        Forward{{package.&Consumer}} |
        Forward{{package.&Both}} |
        Reverse{{package.&Supplier}} |
        Reverse{{package.&Both}}
}
```

This can only be used if the assumptions associated with `Forward{}` and `Reverse{}` concerning the linked operations are valid.

10.9 Supplier performs

The parameterized operation set `SupplierPerforms{}`, given an operation package `package` as its ASN.1 parameter, is that which contains all operations which the supplier needs to be capable of performing. It is specified as follows:

```
SupplierPerforms {OPERATION-PACKAGE:package} OPERATION ::=
{
        Forward{{package.&Supplier}} |
        Forward{{package.&Both}} |
        Reverse{{package.&Consumer}} |
        Reverse{{package.&Both}}
}
```

This can only be used if the assumptions associated with `Forward{}` and `Reverse{}` concerning the linked operations are valid.

10.10 All operations

The parameterized operation set `AllOperations{}`, given an operation package `package` as its ASN.1 parameter, is that which contains all operations involved in the package. It is specified as follows:

```
AllOperations {OPERATION-PACKAGE:package} OPERATION ::=
{
        ConsumerPerforms {package} | SupplierPerforms {package}
}
```

This can only be used if the assumptions associated with `Forward{}` and `Reverse{}` concerning the linked operations are valid.

10.11 recode

The parameterized operation `recode{}`, given an `operation` as its first ASN.1 parameter, is identical in all respects to that `operation`, except that its `&code` field has the value of the second ASN.1 parameter, `code`. It is specified as follows:

```
recode {OPERATION:operation, Code:code} OPERATION ::=
{
        ARGUMENT                operation.&ArgumentType
                OPTIONAL                operation.&argumentTypeOptional
        RETURN RESULT           operation.&returnResult
        RESULT                  operation.&ResultType,
                OPTIONAL                operation.&resultTypeOptional
        ERRORS                  {operation.&Errors}
        ALWAYS RESPONDS         operation.&alwaysReturns
        LINKED                  {operation.&Linked}
        SYNCHRONOUS             operation.&synchronous
        INVOKE PRIORITY         {operation.&InvokePriority}
        RESULT-PRIORITY         {operation.&ResultPriority}
        CODE                    code
}
```

10.12 switch

The parameterized package `switch{}`, given a `package` as its first ASN.1 parameter, is identical in all respects to that `package`, except that the consumer and supplier roles are reversed and its `&id` field has the value of the second ASN.1 parameter, `id`. It is specified as follows:

```
switch {OPERATION-PACKAGE:package, OBJECT IDENTIFIER:id} OPERATION-PACKAGE ::=
{
        OPERATIONS              {package.&Both}
        SUPPLIER INVOKES        {package.&Supplier}
        CONSUMER INVOKES        {package.&Consumer}
        ID                      id
}
```

10.13 combine

10.13.1 The parameterized operation package `combine{}`, combines a number of operation packages into one. It is specified as follows:

```
combine {OPERATION-PACKAGE:ConsumerConsumes, OPERATION-PACKAGE:ConsumerSupplies,
OPERATION-PACKAGE:base} OPERATION-PACKAGE ::=
{
        OPERATIONS              {ConsumerConsumes.&Both | ConsumerSupplies.&Both}
        SUPPLIER INVOKES        {ConsumerConsumes.&Supplier    | ConsumerSupplies.&Consumer}
        CONSUMER INVOKES        {ConsumerConsumes.&Consumer  | ConsumerSupplies.&Supplier}
        ID                      base.&id
}
```

10.13.2 The ASN.1 parameters which must be supplied to produce a fully-defined operation package are as follows:

a) `ConsumerConsumes` – The set of operation packages in which the consumer of the resulting operation package will play the consumer role.

b) `ConsumerSupplies` – The set of operation packages in which the consumer of the resulting operation package will play the supplier role.

c) `base` – An operation package, typically incompletely defined, whose `&id` field is used in defining the resulting operation package.

NOTE – The `base` package will not normally have any operations defined; even if it does, they will not appear in the resulting operation package unless they also appear in `ConsumerConsumes` or `SupplierSupplies`.

10.14 ROS single abstract syntax

The parameterized type `ROS-SingleAS{}` is the basis of an abstract syntax which allows the invocation and reporting of all of the operations in some `package`, using invoke ids from `InvokeIdSet`. It is specified as follows:

```
ROS-SingleAS {InvokeId:InvokeIdSet, OPERATION-PACKAGE:package} ::=
        ROS {{InvokeIdSet}, {AllOperations{package}}, {AllOperations{package}}}
```

10.15 ROS consumer abstract syntax

The parameterized type `ROS-ConsumerAS{}` is the basis of an abstract syntax which allows the invocation of all of the consumer-performed operations, and reporting of all supplier-performed operations in some `package`, using invoke ids from `InvokeIdSet`. It is specified as follows:

```
ROS-ConsumerAS {InvokeId:InvokeIdSet, OPERATION-PACKAGE:package} ::=
        ROS {{InvokeIdSet}, {ConsumerPerforms{package}}, {SupplierPerforms{package}}}
```

10.16 ROS supplier abstract syntax

The parameterized type `ROS-SupplierAS{}` is the basis of an abstract syntax which allows the invocation of all of the supplier-performed operations, and reporting of all consumer-performed operations in some `package`, using invoke ids from `InvokeIdSet`. It is specified as follows:

```
ROS-SupplierAS {InvokeId:InvokeIdSet, OPERATION-PACKAGE:package} ::=
        ROS {{InvokeIdSet}, {SupplierPerforms{package}}, {ConsumerPerforms{package}}}
```

Annex A

ASN.1 modules

(This annex forms an integral part of this Recommendation | International Standard)

```
Remote-Operations-Information-Objects {joint-iso-itu-t remote-operations(4) informationObjects(5) version1(0)}
DEFINITIONS ::=
BEGIN
-- exports everything
IMPORTS emptyBind, emptyUnbind FROM Remote-Operations-Useful-Definitions {joint-iso-itu-t remote-operations(4)
useful-definitions(7) version1(0)};

OPERATION ::= CLASS
{
        &ArgumentType                   OPTIONAL,
        &argumentTypeOptional           BOOLEAN OPTIONAL,
        &returnResult                   BOOLEAN DEFAULT TRUE,
        &ResultType                     OPTIONAL,
        &resultTypeOptional             BOOLEAN OPTIONAL,
        &Errors                         ERROR OPTIONAL,
        &Linked                         OPERATION OPTIONAL,
        &synchronous                    BOOLEAN DEFAULT FALSE,
        &alwaysReturns                  BOOLEAN DEFAULT TRUE,
        &InvokePriority                 Priority OPTIONAL,
        &ResultPriority                 Priority OPTIONAL,
        &operationCode                  Code UNIQUE OPTIONAL
}
WITH SYNTAX
{
        [ARGUMENT                       &ArgumentType       [OPTIONAL    &argumentTypeOptional]]
        [RESULT                         &ResultType         [OPTIONAL    &resultTypeOptional]]
        [RETURN RESULT                  &returnResult]
        [ERRORS                         &Errors]
        [LINKED                         &Linked]
        [SYNCHRONOUS                    &synchronous]
        [ALWAYS RESPONDS                &alwaysReturns]
        [INVOKE PRIORITY                &InvokePriority]
        [RESULT-PRIORITY                &ResultPriority]
        [CODE                           &operationCode]
}
ERROR ::= CLASS
{
        &ParameterType                  OPTIONAL,
        &parameterTypeOptional          BOOLEAN OPTIONAL,
        &ErrorPriority                  Priority OPTIONAL,
        &errorCode                      Code UNIQUE OPTIONAL
}
WITH SYNTAX
{
        [PARAMETER                      &ParameterType      [OPTIONAL    &parameterTypeOptional]]
        [PRIORITY                       &ErrorPriority]
        [CODE                           &errorCode]
}
OPERATION-PACKAGE ::= CLASS
{
        &Both                           OPERATION OPTIONAL,
        &Consumer                       OPERATION OPTIONAL,
        &Supplier                       OPERATION OPTIONAL,
        &id                             OBJECT IDENTIFIER UNIQUE OPTIONAL
}
-- continued on the next page
```

```
WITH SYNTAX
{
	[OPERATIONS			&Both]
	[CONSUMER INVOKES		&Supplier]
	[SUPPLIER INVOKES		&Consumer]
	[ID				&id]
}
CONNECTION-PACKAGE ::= CLASS
{
	&bind				OPERATION DEFAULT emptyBind,
	&unbind				OPERATION DEFAULT emptyUnbind,
	&responderCanUnbind		BOOLEAN DEFAULT FALSE,
	&unbindCanFail			BOOLEAN DEFAULT FALSE,
	&id				OBJECT IDENTIFIER UNIQUE OPTIONAL
}
WITH SYNTAX
{
	[BIND				&bind]
	[UNBIND				&unbind]
	[RESPONDER UNBIND		&responderCanUnbind]
	[FAILURE TO UNBIND		&unbindCanFail]
	[ID				&id]
}
CONTRACT ::= CLASS
{
	&connection			CONNECTION-PACKAGE OPTIONAL,
	&OperationsOf			OPERATION-PACKAGE OPTIONAL,
	&InitiatorConsumerOf		OPERATION-PACKAGE OPTIONAL,
	&InitiatorSupplierOf		OPERATION-PACKAGE OPTIONAL,
	&id				OBJECT IDENTIFIER UNIQUE OPTIONAL
}
WITH SYNTAX
{
	[CONNECTION			&connection]
	[OPERATIONS OF			&OperationsOf]
	[INITIATOR CONSUMER OF		&InitiatorConsumerOf]
	[RESPONDER CONSUMER OF		&InitiatorSupplierOf]
	[ID				&id]
}
ROS-OBJECT-CLASS ::= CLASS
{
	&Is				ROS-OBJECT-CLASS OPTIONAL,
	&Initiates			CONTRACT OPTIONAL,
	&Responds			CONTRACT OPTIONAL,
	&InitiatesAndResponds		CONTRACT OPTIONAL,
	&id				OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX
{
	[IS				&Is]
	[BOTH				&InitiatesAndResponds]
	[INITIATES			&Initiates]
	[RESPONDS			&Responds]
	ID				&id
}
Code ::= CHOICE
{
	local				INTEGER,
	global				OBJECT IDENTIFIER
}
Priority ::= INTEGER (0..MAX)
END -- end of Information Object specifications
```

```
Remote-Operations-Generic-ROS-PDUs {joint-iso-itu-t remote-operations(4) generic-ROS-PDUs(6) version1(0)}
DEFINITIONS IMPLICIT TAGS ::=
BEGIN
-- exports everything
IMPORTS OPERATION, ERROR FROM Remote-Operations-Information-Objects {joint-iso-itu-t remote-operations(4)
informationObjects(5) version1(0)};

ROS {InvokeId:InvokeIdSet, OPERATION:Invokable, OPERATION:Returnable} ::= CHOICE
{
        invoke          [1]  Invoke {{InvokeIdSet}, {Invokable}},
        returnResult    [2]  ReturnResult {{Returnable}},
        returnError     [3]  ReturnError {{Errors{{Returnable}}}},
        reject          [4]  Reject
}
(CONSTRAINED BY  {-- must conform to the above definition --}
! RejectProblem         : general-unrecognizedPDU)

Invoke {InvokeId:InvokeIdSet, OPERATION:Operations} ::= SEQUENCE
{
        invokeId        InvokeId    (InvokeIdSet)
                                    (CONSTRAINED BY  {-- must be unambiguous --}
                                    ! RejectProblem        : invoke-duplicateInvocation),
        linkedId        CHOICE {
                                        present   [0]  IMPLICIT present <   InvokeId,
                                        absent    [1]  IMPLICIT NULL
                                            }
                                    (CONSTRAINED BY  {-- must identify an outstanding operation  --}
                                    ! RejectProblem        : invoke-unrecognizedLinkedId)
                                    (CONSTRAINED BY  {-- which has one or more linked operations--}
                                    ! RejectProblem        : invoke-linkedResponseUnexpected)
                                        OPTIONAL,
        opcode          OPERATION.&operationCode
                                    ({Operations}
                                    ! RejectProblem        : invoke-unrecognizedOperation),
        argument        OPERATION.&ArgumentType
                                    ({Operations} {@opcode}
                                    ! RejectProblem        : invoke-mistypedArgument)
                                        OPTIONAL
}
(CONSTRAINED BY  {-- must conform to the above definition --}
! RejectProblem         : general-mistypedPDU)
(
        WITH COMPONENTS
        {...,
            linkedId ABSENT
        }
|       WITH COMPONENTS
        {...,
            linkedId PRESENT,
            opcode
            (CONSTRAINED BY  {-- must be in the &Linked field of the associated operation --}
            ! RejectProblem        : invoke-unexpectedLinkedOperation)
        }
)
-- continued on the next page
```

```
ReturnResult {OPERATION:Operations}::= SEQUENCE
{
        invokeId        InvokeId
                                (CONSTRAINED BY {-- must be that for an outstanding operation --}
                                ! RejectProblem         : returnResult-unrecognizedInvocation)
                                (CONSTRAINED BY {-- which returns a result --}
                                ! RejectProblem         : returnResult-resultResponseUnexpected),
        result          SEQUENCE
        {
        opcode          OPERATION.&operationCode
                                ({Operations})(CONSTRAINED BY {-- identified by invokeId --}
                                ! RejectProblem         : returnResult-unrecognizedInvocation),
        result          OPERATION.&ResultType
                                ({Operations} {@opcode}
                                ! RejectProblem         : returnResult-mistypedResult)
        }
                        OPTIONAL
}
(CONSTRAINED BY {-- must conform to the above definition --}
! RejectProblem         : general-mistypedPDU)

ReturnError {ERROR:Errors} ::= SEQUENCE
{
        invokeId        InvokeId
                                (CONSTRAINED BY {-- must be that for an outstanding operation --}
                                ! RejectProblem         : returnError-unrecognizedInvocation)
                                (CONSTRAINED BY {-- which returns an error --}
                                ! RejectProblem         : returnError-errorResponseUnexpected),
        errcode         ERROR.&errorCode
                                ({Errors}
                                ! RejectProblem : returnError-unrecognizedError)
                                (CONSTRAINED BY {-- must be in the &Errors field of the associated operation --}
                                ! RejectProblem : returnError-unexpectedError),
        parameter       ERROR.&ParameterType
                                ({Errors}{@errcode}
                                ! RejectProblem : returnError-mistypedParameter) OPTIONAL
}
(CONSTRAINED BY {-- must conform to the above definition --}
! RejectProblem         : general-mistypedPDU)

Reject ::= SEQUENCE
{
        invokeId        InvokeId,
        problem         CHOICE
            {
                        general         [0]  GeneralProblem,
                        invoke          [1]  InvokeProblem,
                        returnResult    [2]  ReturnResultProblem,
                        returnError     [3]  ReturnErrorProblem
            }
}
(CONSTRAINED BY {-- must conform to the above definition --}
! RejectProblem         : general-mistypedPDU)

GeneralProblem ::= INTEGER
{
        unrecognizedPDU (0),
        mistypedPDU (1),
        badlyStructuredPDU (2)
}
```

-- continued on the next page

```
InvokeProblem ::= INTEGER
{
        duplicateInvocation (0),
        unrecognizedOperation (1),
        mistypedArgument (2),
        resourceLimitation (3),
        releaseInProgress (4),
        unrecognizedLinkedId (5),
        linkedResponseUnexpected (6),
        unexpectedLinkedOperation (7)
}

ReturnResultProblem ::= INTEGER
{
        unrecognizedInvocation (0),
        resultResponseUnexpected (1),
        mistypedResult (2)
}

ReturnErrorProblem ::= INTEGER
{
        unrecognizedInvocation (0),
        errorResponseUnexpected (1),
        unrecognizedError (2),
        unexpectedError (3),
        mistypedParameter (4)
}

RejectProblem ::= INTEGER
{
        general-unrecognizedPDU (0),
        general-mistypedPDU (1),
        general-badlyStructuredPDU (2),
        invoke-duplicateInvocation (10),
        invoke-unrecognizedOperation (11),
        invoke-mistypedArgument (12),
        invoke-resourceLimitation (13),
        invoke-releaseInProgress (14),
        invoke-unrecognizedLinkedId (15),
        invoke-linkedResponseUnexpected (16),
        invoke-unexpectedLinkedOperation (17),
        returnResult-unrecognizedInvocation (20),
        returnResult-resultResponseUnexpected (21),
        returnResult-mistypedResult (22),
        returnError-unrecognizedInvocation (30),
        returnError-errorResponseUnexpected (31),
        returnError-unrecognizedError (32),
        returnError-unexpectedError (33),
        returnError-mistypedParameter (34)
}

InvokeId ::= CHOICE
{
        present    INTEGER,
        absent     NULL
}

noInvokeId InvokeId ::= absent:NULL

NoInvokeId InvokeId ::= {noInvokeId}

Errors {OPERATION:Operations} ERROR ::= {Operations.&Errors}

-- continued on the next page
```

```
Bind {OPERATION:operation} ::= CHOICE
{
        bind-invoke     [16]    OPERATION.&ArgumentType({operation}),
        bind-result     [17]    OPERATION.&ResultType ({operation}),
        bind-error      [18]    OPERATION.&Errors.&ParameterType ({operation})
}

Unbind {OPERATION:operation} ::= CHOICE
{
        unbind-invoke   [19]    OPERATION.&ArgumentType({operation}),
        unbind-result   [20]    OPERATION.&ResultType ({operation}),
        unbind-error    [21]    OPERATION.&Errors.&ParameterType ({operation})
}

END -- end of generic ROS PDU definitions
```

```
Remote-Operations-Useful-Definitions {joint-iso-itu-t remote-operations(4) useful-definitions(7) version1(0)}
DEFINITIONS IMPLICIT TAGS ::=
BEGIN
-- exports everything
IMPORTS OPERATION, ERROR, OPERATION-PACKAGE, Code FROM Remote-Operations-Information-Objects
{joint-iso-itu-t remote-operations(4) informationObjects(5) version1(0)}
InvokeId, ROS{}, FROM Remote-Operations-Generic-ROS-PDUs {joint-iso-itu-t remote-operations(4)
generic-ROS-PDUs(6) version1(0)};

emptyBind OPERATION ::= {ERRORS {refuse} SYNCHRONOUS TRUE}

emptyUnbind OPERATION ::= { SYNCHRONOUS TRUE }

refuse ERROR ::= {CODE  local:-1}

no-op OPERATION ::=
{
        ALWAYS RESPONDS FALSE
        CODE  local:-1
}

Forward {OPERATION:OperationSet} OPERATION ::=
{
        OperationSet |
        OperationSet.&Linked.&Linked |
        OperationSet.&Linked.&Linked.&Linked.&Linked
}

Reverse   {OPERATION:OperationSet} OPERATION ::=
        {Forward{{OperationSet.&Linked}}}

ConsumerPerforms {OPERATION-PACKAGE:package} OPERATION ::=
{
        Forward{{package.&Consumer}} |
        Forward{{package.&Both}} |
        Reverse{{package.&Supplier}} |
        Reverse{{package.&Both}}
}

SupplierPerforms {OPERATION-PACKAGE:package} OPERATION ::=
{
        Forward{{package.&Supplier}} |
        Forward{{package.&Both}} |
        Reverse{{package.&Consumer}} |
        Reverse{{package.&Both}}
}

AllOperations {OPERATION-PACKAGE:package} OPERATION ::=
{
        ConsumerPerforms {package} |
        SupplierPerforms {package}
}
-- continued on the next page
```

```
recode {OPERATION:operation, Code:code} OPERATION ::=
{
        ARGUMENT                operation.&ArgumentType
                OPTIONAL                operation.&argumentTypeOptional
        RETURN RESULT           operation.&returnResult
        RESULT                  operation.&ResultType
                OPTIONAL                operation.&resultTypeOptional
        ERRORS                  {operation.&Errors}
        ALWAYS RESPONDS         operation.&alwaysReturns
        LINKED                  {operation.&Linked}
        SYNCHRONOUS             operation.&synchronous
        INVOKE PRIORITY         {operation.&InvokePriority}
        RESULT-PRIORITY         {operation.&ResultPriority}
        CODE                    code
}

switch {OPERATION-PACKAGE:package, OBJECT IDENTIFIER:id} OPERATION-PACKAGE ::=
{
        OPERATIONS              {package.&Both}
        SUPPLIER INVOKES        {package.&Supplier}
        CONSUMER INVOKES        {package.&Consumer}
        ID                      id
}

combine {OPERATION-PACKAGE:ConsumerConsumes, OPERATION-PACKAGE:ConsumerSupplies,
OPERATION-PACKAGE:base} OPERATION-PACKAGE ::=
{
        OPERATIONS              {ConsumerConsumes.&Both | ConsumerSupplies.&Both}
        SUPPLIER INVOKES        {ConsumerConsumes.&Supplier    | ConsumerSupplies.&Consumer}
        CONSUMER INVOKES        {ConsumerConsumes.&Consumer  | ConsumerSupplies.&Supplier}
        ID                      base.&id
}

ROS-SingleAS {InvokeId:InvokeIdSet, OPERATION-PACKAGE:package} ::=
        ROS {{InvokeIdSet}, {AllOperations{package}}, {AllOperations{package}}}

ROS-ConsumerAS {InvokeId:InvokeIdSet, OPERATION-PACKAGE:package} ::=
        ROS {{InvokeIdSet}, {ConsumerPerforms{package}}, {SupplierPerforms{package}}}

ROS-SupplierAS {InvokeId:InvokeIdSet, OPERATION-PACKAGE:package} ::=
        ROS {{InvokeIdSet}, {SupplierPerforms{package}}, {ConsumerPerforms{package}}}

END -- end of useful definitions.
```

Annex B

Guidelines for the use of the notation

(This annex does not form an integral part of this Recommendation | International Standard)

This annex provides examples and guidelines for application protocol designers on the use of the information object classes corresponding to the basic concepts of ROS, and on the application of the set of parameterized useful definitions.

B.1 Examples of Operations and their Errors

This subclause provides some examples of information objects belonging to the class OPERATION and ERROR.

```
operationExample1 OPERATION ::=
   {
   ARGUMENT                ArgumentType1
   RESULT                  ResultType1
   ERRORS                  {errorExample1 | errorExample2}
   LINKED                  {operationExample2}
   CODE                    local:1
   }

operationExample2 OPERATION ::=
   {
   ARGUMENT                ArgumentType2
   RESULT                  ResultType2 OPTIONAL TRUE
   LINKED                  {operationExample4}
   ALWAYS RESPONDS         FALSE
   CODE                    local:2
   }

operationExample3 OPERATION ::=
   {
   ARGUMENT                ArgumentType3
   ERRORS                  {errorExample3}
   SYNCHRONOUS             TRUE
   CODE                    local:3
   }

operationExample4 OPERATION ::=
   {
   ARGUMENT                ArgumentType4
   RETURN RESULT           FALSE
   ALWAYS RESPONDS         FALSE
   CODE                    local:4
   }
```

The asynchronous remote operation operationExample1, which carries an argument of type ArgumentType1, always responds (as implied by the absence of the key words ALWAYS RESPONDS), returning, in the case of the successful performance of the operation, a result value of type ResultType1, or, if the operation is unsuccessful, one of the errors errorExample1 or errorExample2 depending on the circumstances of the failure. The operation operationExample2 is "linked" to this operation, which means that operationExample2 can be invoked in response to this operation at any time prior to the completion of this operation. This operationExample1 is identified by its code, the integer 1.

The asynchronous remote operation operationExample2 carries an argument ArgumentType2. It is an operation which either never fails or, should it fail, the failure is not reported. When the result of successfully performing this operation is reported, the result value of type ResultType2 may optionally be omitted by the performer. The operation operationExample4 is "linked" to this operation, which means that operationExample4 can be invoked in response to operationExample2 at any time before the completion of the latter. This operation is identified by the integer value 2.

The synchronous remote operation operationExample3, identified by the integer value 3 and carrying an argument of type ArgumentType3, always reports its outcome. Should the operation be performed successfully, an indication is returned but no value of the result is returned. If the operation fails, an error errorExample3 is returned.

NOTE – As this is a synchronous operation, the application designer must ensure that this operation always returns, particularly if further *synchronous* operations are expected to be invoked by the invoker.

The asynchronous operation `operationExample4`, identified by the integer value 4, and whose invocation is accompanied by an argument of type `ArgumentType4`, does not return.

The following are instances of the (information object) class `ERROR` which are used to report the unsuccessful performance of (some of) the operations defined just above:

```
errorExample1 ERROR ::=
   {
   PARAMETER          ParameterType1
   CODE               local:1
   }

errorExample2 ERROR ::=
   {
   PARAMETER          ParameterType2 OPTIONAL TRUE
   CODE               local:2

errorExample3 ERROR
   {
   CODE               local:3
   }
```

`errorExample1`, which can be used to report the unsuccessful performance of either `operationExample1` or `operationExample2`, is identified by the integer value 1. An error diagnostic parameter value of type `ParameterType1` accompanies this error.

`errorExample2`, which can be used to report the unsuccessful completion of `operationExample1`, is accompanied by a value of type `ParameterType2`. At the discretion of the performer, this value may sometimes be omitted. It is identified by the integer 2.

`errorExample3`, which is identified by the integer value 3, is used to report the unsuccessful performance of `operationExample3`. No parameter value (i.e. error diagnostic) is returned with this error.

B.2 Examples of Operation Packages and the use of switch{}

The operations and errors defined in section B.1 can be grouped into an operation package, `package1` as follows:

```
package1 OPERATION-PACKAGE ::=
   {
   CONSUMER INVOKES      {operationExample1 | operationExample3}
   SUPPLIER INVOKES      {operationExample2}
   ID                    objectIdentifierOfPackage1
   }
```

Of the two ROS-objects that interact, one of them, arbitrarily called the "consumer", can invoke `operationExample1` and `operationExample3`. The other ROS-object, designated the "supplier", may only invoke `operationExample2`. This particular combination is identified globally by the value `objectIdentifierOfPackage1`.

NOTE – The terms "consumer" and "supplier", which distinguish the two interacting ROS objects, have to be defined with reference to something outside remote operations. The simplest case is to define them with respect to their roles in forming an association contract (see B.5).

A different combination of the same collection of operations forms `package2`, defined as follows:

```
package2 OPERATION-PACKAGE ::=
   {
   BOTH                  {operationExample3}
   CONSUMER INVOKES      {operationExample1}
   ID                    objectIdentifierOfPackage2
   }
```

In `package2`, either object may invoke `operationExample3`, the consumer may invoke `operationExample1`, while neither side may invoke `operationExample2`.

A third operation package, `package3`, can be derived from `package1` by using the parameterized definition `switch{}`.

```
package3 OPERATION-PACKAGE ::= switch{OPERATION:package1, objectIdentifierOfPackage3}
```

The use of `switch{}` reverses the operations that can be invoked by the "consumer" and "supplier" in `package1` and allocate a new object identifier value to this combination. Thus, `package3` written out in full is as follows:

```
package3 OPERATION-PACKAGE ::=
    {
    CONSUMER INVOKES      {operationExample2}
    SUPPLIER INVOKES      {operationExample1 | operationExample3}
    ID                    objectIdentifierOfPackage3
    }
```

B.3 Examples of Bind and Unbind operations

An association can be established dynamically by invoking a `bind` operation, an example of which is:

```
bindExample1 OPERATION ::=
    {
    ARGUMENT              BindArgumentType1
    RESULT                BindResultType1
    ERRORS                {bindError1}
    SYNCHRONOUS           TRUE
    }

bindError1 ERROR ::=
    {
    PARAMETER     BindErrorType1  OPTIONAL  TRUE
    }
```

The synchronous operation `bindExample1` is used to establish an association between two ROS-objects. The invocation of the operation is accompanied by an argument value of type `BindArgumentType1` and the successful establishment of the establishment of the binding is accompanied by a result value of type `BindResultType1`. In case the association is not successfully established, an error `bindError1` will be returned with the accompanying diagnostic parameter `BindErrorType1` optionally present.

The release of the association is accomplished by the invocation of another synchronous operation `unBindExample1` defined as follows.

```
unBindExample1 OPERATION ::=
    {
    ARGUMENT              UnBindArgumentType1
    RESULT                UnBindResultType1  OPTIONAL  TRUE
    ERRORS                {unBindError1}
    SYNCHRONOUS           TRUE
    }

unBindError1 ERROR ::=
    {
    PARAMETER     UnBindErrorType1  OPTIONAL  TRUE
    }
```

B.4 Examples of Connection Packages

The examples `bindExample1` and `unBindExample1` can be used to define a connection package `connectionPackage1`, which may be used to dynamically establish or release an association between two ROS-objects.

```
connectionPackage1 CONNECTION-PACKAGE ::=
    {
    BIND                  bindExample1
    UNBIND                unBindExample1
    RESPONDER UNBIND      TRUE
    FAILURE TO UNBIND     TRUE
    ID                    objectIdentifierOfConnectionPackage1
    }
```

connectionPackage1, which is identified globally by the value objectIdentifierOfConnection-Package1 when the association between the two ROS-objects is dynamically established or announced, uses the operations bindExample1 and unBindExample1 to, respectively, establish and terminate the association. It allows the association establishment responder to unbind and permits the association to remain even if the unBindExample1 operation fails.

An example of a very simple connection package which defaults to the use of the operations emptyBind and emptyUnbind (see 10.2 and 10.3) to, respectively, establish and terminate an association is given below.

```
simpleConnectionPackage CONNECTION-PACKAGE ::=
    {
    ID       objectIdentifierOfSimpleConnectionPackage
    }
```

For this connection package, only the association establishment initiator may invoke the binding and the association is released even if the attempt at unbinding fails.

B.5 Example of an Association Contract

The association contract contract1 defined as

```
contract1 CONTRACT ::=
    {
    CONNECTION                  connectionPackage1
    INITIATOR CONSUMER OF       {package1}
    ID                          objectIdentifierOfContract1
    }
```

shows that the connection package connectionPackage1 is used to establish and terminate the association and that the ROS-object initiating the association establishment plays the role of the "consumer" in the operation package package1. This contract is identified by the value objectIdentifierOfContract1.

B.6 Examples of ROS-objects

One may define a ROS-object, object1,

```
object1 ROS-OBJECT-CLASS ::=
    {
    INITIATES       {contract1}
    ID              objectIdentifierofROSObject1
    }
```

object1, which is identified by objectIdentifierOfROSObject1, is one of a set of objects that can interact with other ROS-objects by initiating the interaction using the association contract contract1.

Similarly, object2, defined as

```
object2 ROS-OBJECT-CLASS ::=
    {
    RESPONDS        {contract1}
    ID              objectIdentifierOf ROSObject2
    }
```

is one of a set of objects which can interact with other ROS-objects by responding to initiation of the interactions offered by the association contract contract1.

B.7 Example of the use of Forward{} and Reverse{}

The information object set ConsumerInvokes, derived from package1.&Supplier defined in B.2 as follows:

```
ConsumerInvokes OPERATION ::= {package1.&Supplier}
```

gives rise to the set {operationExample1 | operationExample2} which lists the operations that may be invoked by the "consumer." In turn,

```
SupplierLinkedInvokes OPERATION ::= {ConsumerInvokes.&Linked}
```

produces {operationExample2} which is the set of operations which may be invoked in the "reverse" direction, i.e. by the "supplier", while

```
ConsumerLinkedInvokes OPERATION ::= {SupplierLinkedInvokes.&Linked}
```

produces {operationExample4} which is the operation set indirectly linked to the set formed by package1.&Consumer and is invoked by the "consumer".

The operation set Forward{OPERATION: ConsumerInvokes} is the set of operations which include the original operation set ConsumerInvokes together with any indirectly-linked operations with the same "directionality", i.e. in our example, the ones that may be invoked by the "consumer". Thus:

```
Forward {OPERATION:ConsumerInvokes} OPERATION ::=
        {operationExample1 | operationExample3 | operationExample4}
```

On the other hand, the operation set Reverse{OPERATION:ConsumerInvokes} is the collection of operations that are a part of the set formed by package1.&Consumer.&Linked, which are the operations that can be invoked by the "supplier" in response to those in the set of operations given by ConsumerInvokes, together with those indirectly linked to it and with the same "directionality", i.e., from "consumer" to "supplier". That is,

```
Reverse{OPERATION:ConsumerInvokes} OPERATION ::=
                                Forward{OPERATION:SupplierLinkedInvokes}
```

which, when written out explicitly, is

```
Reverse{OPERATION:ConsumerInvokes} OPERATION ::= {operationExample2}
```

On the other hand, for the operation set SupplierInvokes derived as follows

```
SupplierInvokes OPERATION ::= {package1.&Consumer}
```

we obtain {operationExample2}.

Thus, we have

```
Forward{OPERATION:SupplierInvokes} OPERATION ::= {operationExample2}

Reverse{OPERATION:SupplierInvokes} OPERATION ::= {operationExample4}
```

B.8 Examples of ConsumerPerforms{}, SupplierPerforms{} and AllOperations{}

With the help of the examples in B.7, given the operation package package1, the "consumer" can perform all the operations in the set

```
ConsumerPerforms{OPERATION-PACKAGE:package1} OPERATION ::=
        {Forward{OPERATION:ConsumerInvokes} |
        {Reverse{OPERATION:SupplierInvokes}}
```

which produces the operation set {operationExample1 | operationExample3 | operationExample4}.

Similarly, in package1, all the operations that the "supplier" should be capable of performing are

```
SupplierPerforms{OPERATION-PACKAGE:package1} OPERATION ::=
{{Forward{OPERATION:SupplierInvokes} |
        {Reverse{OPERATION:ConsumerInvokes}}
```

which is the set {operationExample2}.

The usefulness of the ConsumerPerforms{} and SupplierPerforms{} constructs is that it permits an application designer to see, in a complex situation consisting many nested linkages between the operations in that package, those with the same "directionality", i.e. those that can be invoked by the consumer and those by the supplier, without regard to the nature of their linkages.

The set `AllOperations{OPERATION-PACKAGE:package1}` lists all the operations both implicitly (i.e. via linkages) and explicitly cited by `package1` that can be invoked by either side in some instance of the use of this package.

```
AllOperations{OPERATION-PACKAGE:package1} OPERATION ::=
        {ConsumerPerforms{package1} |
                                    {SupplierPerforms{package1}}
```

which produces the operation set `{operationExample1 | operationExample2 | operationExample3 | operationExample4}`.

Annex C

Migrating from the ROS macros

(This annex does not form an integral part of this Recommendation | International Standard)

In previous versions of ROS, ASN.1 macros were provided to allow ROS application designers to specify their operations, errors, bind-operations, application-service-elements, etc. In addition, and closely affiliated to ROS, CCITT Recommendation X.407, provided further macros which designers could use in the object-based specification of distributed applications. The macro capability is being phased out from ASN.1, and accordingly, the ROS notation provided by the present Specification makes use instead of the ASN.1 "macro replacement" notation, including information object classes and parameterization.

C.1 Introduction

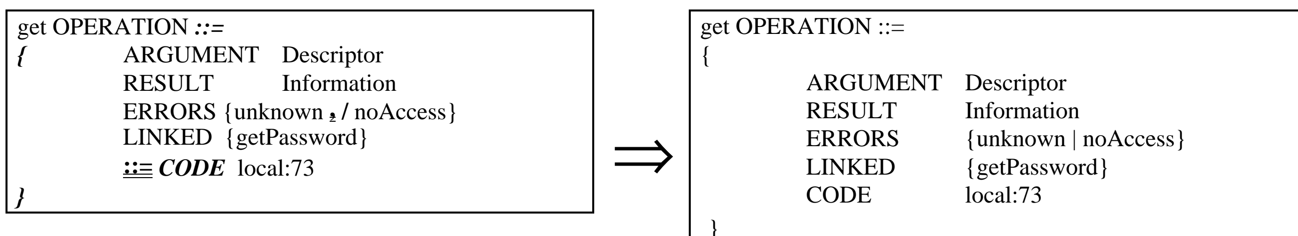
However, there are a significant number of specifications in existence which use the macro approach. Accordingly, this annex shows how the use of these macros could be transformed into the use of the replacement notation. In many cases the replacement notation is more comprehensive than the macro it replaces; however, such features are not generally pointed out in this annex, which is concerned only with understanding existing macro usage. The macros should not be used for new specifications.

This annex is organized with one subclause devoted to each existing macro. The subclause describes the purpose of the macro, gives an example of its use together with the equivalent in the newer notation, and then explains any aspects not clear from the example. In the examples using the macro notation, symbols `underlined` are to be deleted to form the new notation, while symbols in ***BOLD ITALICS*** are to be inserted.

One particular approach to using macros, "two-stage definition" has been widely employed in the existing specifications. In this approach, the "identifiers" for various kinds of information object (e.g. the operation or error code) is assigned in the second stage. However the interpretation of such specifications by the reader sometimes requires taking into account such normally irrelevant factors as the similarity of spelling of distinct ASN.1 references, and the proximity of ASN.1 definitions. This is not acceptable with the replacement notation. Accordingly, this annex describes the mapping only of the second or only stage of the definition using macros.

C.2 Operation

The `OPERATION` macro was used to specify operations (except for bind and unbind operations). The changes as we go from defining an instance, called `get`, of the `OPERATION` macro to a member of the `OPERATION` information object class is shown below as we go, respectively, from right to left. Looking at the macro notation on the left hand side, symbols underlined twice are to be deleted from the old notation while symbols in bold italics are to be inserted into the old notation to form the new notation (as seen on the right hand side).



```
get OPERATION ::=
{       ARGUMENT   Descriptor
        RESULT     Information
        ERRORS {unknown , / noAccess}
        LINKED  {getPassword}
        ::= CODE  local:73
}
```

⇒

```
get OPERATION ::=
{
        ARGUMENT   Descriptor
        RESULT     Information
        ERRORS     {unknown | noAccess}
        LINKED     {getPassword}
        CODE       local:73
}
```

NOTES

1 Omission of the `RESULT` clause in the macro notation ⇒ `RETURN RESULT FALSE` in the new notation.

2 Presence of `RESULT` without data type in the macro notation ⇒ omission of `RESULT` clause in the new notation (the keywords `RETURN RESULT TRUE` is the default and hence omitted).

3 As a result of the above assumption concerning 2-stage definition, the names of individual errors and linked operations start with lower-case letters.

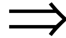
4 The following operation fields permitted by the information object class definition could not be specified with the macro, but, if needed, would have to be specified in text: `&synchronous`, `&InvokePriority`, `&ResultPriority`.

5 It is possible to state in the new notation, through the use of the `&argumentTypeOptional` and `&resultTypeOptional` fields, if, respectively, the argument or result value may, as a user option, be omitted.

C.3 Error

The `ERROR` macro was used to specify errors (except for those of bind and unbind operations).

```
unknown ERROR ::=
{       PARAMETER Descriptor
        ::= CODE local:14
}
```



```
unknown ERROR ::=
{
        PARAMETER       Descriptor
        CODE            local:14
}
```

NOTES

1 The `&ErrorPriority` field could not be specified with the macro, but, if needed, would be specified in text.

2 It is possible to state in the new notation, through the use of the `¶meterTypeOptional` field, if the parameter value, if any, defined to accompany the report of an error may, as a user option, be omitted.

C.4 Bind

The `BIND` macro was used to specify bind operations.

```
Hallohallo ::= BIND OPERATION::=
{       ARGUMENT        HowAreYou
        RESULT          Fine-AndYou
        BIND-ERRORS { GgoAway}
}
```

```
hallo OPERATION::=
{
        ARGUMENT        HowAreYou
        RESULT          Fine-AndYou
        ERRORS          {goAway}
}
```

NOTE – The absence of the key word `CODE` in the new notation indicates that this is a special operation which cannot be invoked using the `Invoke{}` PDU.

C.5 Unbind

The `UNBIND` macro was used to specify unbind operations.

```
Byebye ::= UNBIND OPERATION ::=
{       ARGUMENT        SeeYouSoon
        RESULT          LetsDoLunch
        BIND-ERRORS     { DdontGo}
}
```

```
bye  OPERATION ::=
{
        ARGUMENT        SeeYouSoon
        RESULT          LetsDoLunch
        ERRORS          {dontGo}
}
```

NOTE – The absence of the key word `CODE` in the new notation indicates that this is a special operation which cannot be invoked using the `Invoke{}` PDU.

Annex D

Assignment of object identifier values

(This annex does not form an integral part of this Recommendation | International Standard)

The following object identifier values are assigned in this Recommendation | International Standard:

Clause	Object Identifier Value
Annex A	**{joint-iso-itu-t remote-operations(4) informationObjects(5) version1(0)}**
	{joint-iso-itu-t remote-operations(4) generic-ROS-PDUs(6) version1(0)}
	{joint-iso-itu-t remote-operations(4) useful-definitions(7) version1(0)}