ETSI TECHNICAL REPORT

ETR 060

September 1995

Second Edition

Source: ETSI TC-SPS

Reference: RTR/SPS-02015

ICS: 35.100.60

Key words: ASN.1

Signalling Protocols and Switching (SPS); Guidelines for using Abstract Syntax Notation One (ASN.1) in telecommunication application protocols

ETSI

European Telecommunications Standards Institute

ETSI Secretariat

Postal address: F-06921 Sophia Antipolis CEDEX - FRANCE
Office address: 650 Route des Lucioles - Sophia Antipolis - Valbonne - FRANCE
X.400: c=fr, a=atlas, p=etsi, s=secretariat - **Internet:** secretariat@etsi.fr

Tel.: +33 92 94 42 00 - Fax: +33 93 65 47 16

Contents

Foreword

This ETSI Technical Report (ETR) has been produced by the Signalling Protocols and Switching (SPS) Technical Committee of the European Telecommunications Standards Institute (ETSI).

ETRs are informative documents resulting from ETSI studies which are not appropriate for European Telecommunication Standard (ETS) or Interim European Telecommunication Standard (I-ETS) status. An ETR may be used to publish material which is either of an informative nature, relating to the use or the application of ETSs or I-ETSs, or which is immature and not yet suitable for formal adoption as an ETS or an I-ETS.

This second edition of ETR 060 takes into account the further evolution of ASN.1 since the publication of the first edition in 1992.

Blank page

1 Scope

The purpose of this ETSI Technical Report (ETR) is to provide guidelines on the use of Abstract Syntax Notation One (ASN.1) for specifying telecommunication application protocols.

This ETR is based on ITU-T Recommendations X.680 [1], X.681 [2], X.682 [3] and X.683 [4] which specify the Abstract Syntax Notation One (ASN.1). In case of misalignment, these Recommendations shall be considered as the primary reference.

Unless explicitly indicated, all references to encoding and decoding functions assume the use of the Basic Encoding Rules (BER) or any of their variants as they are specified in ITU-T Recommendation X.690 [5][1].

This ETR is not a tutorial on ASN.1. Tutorial matter exists on this subject, e.g. "A tutorial on Abstract Syntax Notation One" [17], "ASN.1 MACRO Facility" [18], "ASN.1 and ROS" [19], "An overview of ASN.1" [20]. More specific tutorial information on the latest extensions to ASN.1 can be found in "An introduction to the ASN.1 MACRO replacement notation" [21] and "Efficient encoding rules for ASN.1-based protocols" [22].

Annex F of ITU-T Recommendation X.680 [1] also provides a set of general guidelines for use of the notation.

Throughout this ETR, the term "user" denotes a person who employs ASN.1 for protocol design. The term *1988/90 notation* is used to refer to that ASN.1 notation specified in CCITT Recommendation X.208 (1988) | ISO/IEC 8824:1990 [9]. The term *current notation* is used to refer to that specified in ITU-T Recommendation X.680 [1].

Unless explicitly stated, all the guidelines contained in the body of this ETR are also applicable to users of the 1988/90 notation.

Annex A provides guidance for the migration from the 1988/90 notation to the current notation.

Annex B provides specific guidance which only applies to superseded features of the 1988/90 notation.

Terms between quotation marks refer directly to items or productions defined by the ASN.1 standard (e.g. "typereference", "Symbol").

The main objectives of the recommendations made in this ETR are:

a) allow the re-use of data types from one domain to another;

b) ease protocol evolution, taking into account compatibility issues;

c) ease the maintainability of the specifications;

d) ease automated implementation of encoding and decoding functions;

e) ease the production of test specifications, especially when specified using the Tree and Tabular Combined Notation (see ITU-T Recommendation X.292 [11]) which makes a direct use of the ASN.1 type definitions of the protocol to be tested.

[1] ITU-T Recommendation X.690 supersedes CCITT Recommendation X.209 [10].

2 References

This ETR incorporates by dated and undated reference, provisions from other publications. These references are cited at the appropriate places in the text and the publications are listed hereafter. For dated references, subsequent amendments to or revisions of any of these publications apply to this ETR only when incorporated in it by amendment or revision. For undated references the latest edition of the publication referred to applies.

[1] ITU-T Recommendation X.680 (1994): "Specification of abstract syntax notation one (ASN.1): Specification of the basic notation" (also published as ISO/IEC 8824-1).

[2] ITU-T Recommendation X.681 (1994): "Abstract Syntax Notation One (ASN.1): Information Object Specification" (also published as ISO/IEC 8824-2).

[3] ITU-T Recommendation X.682 (1994): "Abstract Syntax Notation One (ASN.1): Constraint Specification" (also published as ISO/IEC 8824-3).

[4] ITU-T Recommendation X.683 (1994): "Abstract Syntax Notation One (ASN.1): Parameterisation of ASN.1 specifications" (also published as ISO/IEC 8824-4).

[5] ITU-T Recommendation X.690 (1994): "Specification of ASN.1 encoding rules: basic encoding rules" (also published as ISO/IEC 8825-1).

[6] ITU-T Recommendation X.691 (1994): "Abstract Syntax Notation One (ASN.1): Packed Encoding Rules" (also published as ISO/IEC 8825-2).

[7] ITU-T Recommendation X.680 (1994): "Specification of abstract syntax notation one (ASN.1): Specification of the basic notation - Amendment 1: Rules for extensibility".

[8] ITU-T Recommendation X.681 (1994): "Abstract Syntax Notation One (ASN.1): Information Object Specification - Amendment 1: Rules for extensibility".

[9] CCITT Recommendation X.208 (1988): "Specification of abstract syntax notation one (ASN.1)" (also published as ISO/IEC 8824:1990).

[10] CCITT Recommendation X.209 (1988): "Specification of basic encoding rules for abstract syntax notation one (ASN.1)".

[11] ITU-T Recommendation X.292 (1993): "OSI Conformance Testing Methodology and Framework: Tree and Tabular Combined Notation (TTCN)" (also published as ISO/IEC 9646-3).

[12] CCITT Recommendation X.219 (1988): "Remote operations; model, notation and service definition".

[13] ITU-T Recommendations Q.771 to Q.775 (1993): "Specifications of Signalling System No 7: Transaction Capabilities (TC)".

[14] CCITT Recommendations Q.771 to Q.775 (1988): "Specifications of Signalling System No. 7, Transaction Capabilities Application Part (TCAP)".

[15] ETS 300 351 (1994): "ETSI object identifier tree; Rules and registration procedures".

[16] ITU-T Recommendation X.880 (1995): "Remote Operations: concepts, model and notation".

[17] "A tutorial on Abstract Syntax Notation One" - David Chappel, Cray Research Inc. - OMNICOM Open System Data Transfer, Trans #25, December 1986.

[18] "ASN.1 MACRO Facility" - Jim Reinstedler, Ungermann-Bass Inc. - OMNICOM Open System Data Transfer, Trans #33, April 1988.

[19] "ASN.1 and ROS: The impact of X400 on OSI" - James E. White - IEEE Journal on Selected Areas In Communications, Vol.7, No.7 - September 1989.

[20] "An overview of ASN.1" - Gerald Neufeld, Son Vuang - Computers and ISDN Systems - No.23 (1992).

[21] "An introduction to the ASN.1 MACRO replacement notation" - Nilo Mitra - AT&T Technical Journal - Vol.73 - No.3 - May/June 1994.

[22] "Efficient encoding rules for ASN.1-based protocols"- Nilo Mitra - AT&T Technical Journal - Vol.73 - No.3 - May/June 1994.

3 Abbreviations

For the purposes of this ETR, the following abbreviations apply

AC	Application Context
APDU	Application Protocol Data Unit
ASE	Application Service Element
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
DSS1	Digital Subscriber Signalling Number one
MAP	Mobile Application Part
PDU	Protocol Data Unit
PICS	Protocol Implementation Conformance Statement
PER	Packed Encoding Rules
ROSE	Remote Operation Service Element
SS7	Signalling System No.7
TC	Transaction Capabilities
TTCN	Tree and Tabular Combined Notation

4 Overview of ASN.1

Signalling messages are often described using a tabular notation; their format and binary representation is specified using tables whose entries are the information elements from which they are built. This method is rather convenient when the message structure is simple and when there is no need to consider different encoding schemes to represent the same information.

The ITU-T Recommendations covering Signalling System No.7 (SS7) and Digital Subscriber Signalling System No. one (DSS1), currently describe most of the Application Protocol Data Units (APDU) in this manner (e.g. Telephone User Part messages, DSS1 "layer 3" messages, etc.). This is also the approach taken in OSI to describe the protocol data units up to layer 5.

However, as far as the signalling information to be exchanged between telecommunication systems becomes more and more complex, the limits of this tabular notation become clear; difficulties for representing structured elements, duplication of definitions due to the mixture between the syntax of an information and the way it is encoded, etc.

For the above reasons, it then becomes necessary to change the description technique of signalling messages. This is achieved using the Abstract Syntax Notation One (ASN.1).

ASN.1 provides a means to describe data types as well as value of these types in an abstract manner. It does this without determining the way instances of these data types are to be represented during transmission.

Since a signalling message, as any protocol data unit, can be represented by a data type (generally a structured one) ASN.1 fulfils very well the requirements for describing complex messages.

Beside the abstraction and the formalism of data descriptions, one of the objectives of ASN.1 is to facilitate the encoding and decoding of values of the types defined using the notation. This is why, unlike the data declaration portion of programming languages, it provides inherently a means for associating identification tags with data types.

ITU-T Recommendation X.680 [1] specifies a number of simple and structured built-in types which allows the user of the notation to define more complex types and associated data values by combining these built-in types. In addition this notation also provides a set of subtype constructors (e.g. value range, size constraint) to define types whose values are only a subset of the values of some other type (the parent type).

Examples of simple built-in types are: boolean type, enumerated type, integer type, octet string type, while examples of built-in structured types are: sequence type, set type, choice type, etc.

Beyond the specification of data units, the latest version of ASN.1 also provides tools for describing other kind of information object classes, relationships between components of a PDU or other kind of constraints, and for parameterizing a specification (see ITU-T Recommendations X.681 [2], X.682 [3], X.683 [4]). Most of these features are intended to serve as a replacement for the MACRO notation and the ANY type defined as part of the 1988/90 notation

Although the term "ASN.1" is still often used to refer both to this notation and the Basic Encoding Rules, new Standards and Recommendations defining signalling protocols should made very clear that the two aspects are distinct (i.e. other encoding rules may be applied to the defined abstract syntax).

While message description is mainly based on the ASN.1 type notation, the ASN.1 value notation is a basis for some implementations and for specifying constraints for test cases written using the TTCN notation (see ITU-T Recommendation X.292 [11]). It is therefore of high importance to define data types in such a way that it is ensured that the resulting value notation is not ambiguous.

5 Specification of protocol data units

5.1 Modules

The following guidelines are appropriate when considering modules:

a) The set of ASN.1 productions which forms a protocol specification shall be organized into one or several ASN.1 modules.

The criteria for organizing the modules are up to the protocol designer (functional domain, PDU type, etc.). However for maintainability purposes, the number of inter-module dependencies (i.e. number of modules seen from one module, number of symbols exported and imported) shall be limited.

NOTE: The number of ASN.1 modules involved in the definition of data units of a particular protocol is independent from the number of ASEs in terms of which this protocol is structured. It has also no impact on the number of abstract syntaxes used to represent instances of these data units.

b) Attention should be paid to avoid cross references between modules which make the parsing of complete protocol data units unnecessarily more complex.

c) As stated in ITU-T Recommendation X.680 [1], each ASN.1 module should be given a module identifier. This is used as a formal reference when exporting or importing definitions between modules or when using external references.

This identifier shall be composed of a "modulereference" (i.e. a name starting with an upper case letter) and optionally a value of type object identifier. Unlike an application-context-name or an abstract-syntax-name, this value is never exchanged between peer protocol machines. However, it is recommended that an object identifier value be always allocated to modules defined in ETSI standards.

For further guidance on the use of object identifiers see "An overview of ASN.1" [20]. Rules for assigning object identifier values within the scope of ETSI are described in ETS 300 351 [15].

d) It is suggested that modules defined for signalling applications be allocated a "modulereference" of the following form:

<Protocol-Name>-<Qualifier>

e.g., MAP-Operations

Where <Protocol-Name> identifies a set of related application layer signalling protocols (e.g. MAP) and <Qualifier> is a suitable acronym for the contents of this module (e.g. Operations, CommonTypes, etc.).

5.2 Tagging

The following guidelines are appropriate when considering tagging:

a) the AUTOMATIC TAGS construct should always be used when defining a new module;

NOTE: The AUTOMATIC TAGS construct is not available to users of the 1988/90 notation.

EXAMPLE:

```
My-Module DEFINITIONS
AUTOMATIC TAGS
::=

BEGIN

My-Type ::= SEQUENCE {
        a       INTEGER,
        b       INTEGER OPTIONAL,
        c       BOOLEAN OPTIONAL
}

END
```

b) protocol designers which need to modify a module defined using the 1988/90 notation should follow the guidelines provided in annex B. They should not add the AUTOMATIC TAGS construct in the module header;

c) protocol designers should avoid to add new definitions in modules where the AUTOMATIC TAGS construct was not used. They should preferably create a new module for that purpose;

d) the protocol designer shall be aware that automatic tagging places restrictions on the possible modifications to a type definition when backward compatibility need to be ensured. In addition to those provided in clause 7, users of automatic tagging shall apply the following rules:

- the order of elements in an existing set- or choice- type shall not be modified in a new version of the specification;

- new elements in a set-, sequence- or choice- type shall be added after existing elements.

5.3 Handling of optional and default elements

When defining a structured type (set or sequence type) the protocol designer has to decide for each "ComponentType" whether the presence of its value is mandatory or not when an instance of this type is being used. ASN.1 provides two means to express that the value of a "ComponentType" can be omitted.

The following guidelines are appropriate when considering optional elements:

a) when the "ComponentType" corresponds to a genuine option and has a default value, the DEFAULT keyword shall be used;

EXAMPLE 1:

```
DataUnit ::= SEQUENCE {
        calledParty         Address,
        isdnSubscriber      BOOLEAN DEFAULT FALSE
}
```

b) when the "ComponentType" corresponds to a genuine option and has no default value, the OPTIONAL keyword shall be used;

EXAMPLE 2:

```
DataUnit ::= SEQUENCE {
        calledParty         Address,
        callingParty        Address OPTIONAL
}
```

c) it is not a good practice to use an OPTIONAL "ComponentType" to represent an information element whose presence depends on the value of another element. A "TableConstraint" should preferable be used (see also subclause 5.7).

NOTE: This facility is not available to users of the 1988/90 notation.

EXAMPLE 3: Define:

```
DataUnit ::= SEQUENCE {
        calledParty         Address,
        supplServiceId      SUPPLEMENTARY-SERVICE.&code,
        supplServiceInfo    SUPPLEMENTARY-SERVICE.&parameters
                            ({SupplServiceSet}{@supplServiceId})
}
-- SUPPLEMENTARY-SERVICE is an information object class defined
-- elsewhere
-- SupplServiceSet is a set of object of this class.
```

rather than:

```
DataUnit ::= SEQUENCE {
        calledParty              Address,
        supplServiceId           SS-Code,
        forwardedToNumber        Address OPTIONAL,
                                 -- present if SS-Code is 1
        callBarringPassword      Password OPTIONAL
                                 -- present if SS-Code is 2
}
```

5.4 Subtyping

The following guidelines are appropriate when considering subtyping:

a) as a general rule, all types which appear in the specification of a PDU shall have their boundaries formally specified. If this is not inherent to the type definition (e.g. a boolean type), this has to be done using the subtyping mechanisms provided by the notation;

   This concerns the integer types, octet string types, bit string types, character string types and all the types derived from them. The maximum and minimum number of components in a sequence-of type or set-of type shall also be specified.

b) if it is not possible to reach an agreement on a particular bound, it is recommended to define a parameterized type, whose parameter is the unresolved bound. The actual bound will have to be provided as part of national specifications or in the PICS. An exception specification should also be added in order to provide a clear specification of the behaviour of an entity in case it receives a value which does not conform to the actual implemented bound;

EXAMPLE 1:

```
UserData {INTEGER:up} ::= OCTET STRING ((SIZE(1..up)) !ErrorSpec:truncate)

RequestId {INTEGER:max) ::= INTEGER ((1..max) ! ErrorSpec: ignore )

ErrorSpec ::= ENUMERATED {
        truncate      (1),
        ignore(2)
}
```

NOTE: This facility is not available to users of the 1988/90 notation.

c) when the same boundaries for value ranges or size constraints are used throughout several subtype specifications or are subject to evolutions, it is recommended to assign a "valuereference" to each of these values and to use them in the subtype definition;

EXAMPLE 2:

```
upperBound INTEGER ::= 20

TypeA ::= OCTET STRING (SIZE (1..upperBound))

TypeB ::= OCTET STRING (SIZE(5..upperBound))
```

d) note that if no lower bound is specified for a type derived from a set-of type or a sequence-type, this means that a valid value for this type is an empty value. If this is semantically not acceptable, it is worth to specify the type as follows:

```
TypeA ::= SEQUENCE SIZE (1..upperBound) OF BaseType.
```

e) use of inner subtyping is also encouraged to avoid duplications when there is a need to define a structured type whose component list is a subset of the component list of an other type (mandatory elements shall be common to both types).

EXAMPLE 3:

```
Address ::= SEQUENCE {
        numberingPlan      [0] NumberingPlan OPTIONAL,
        natureOfAddress    [1] NatureOfAddress OPTIONAL,
        coding             [2] Coding,
        presentation       [3] BOOLEAN,
        screening          [4] ScreeningIndicator,
        addressSignal      [5] DigitString
}

IsdnAddress ::= Address (WITH COMPONENTS {
        ...,
        numberingPlan      ABSENT,
        natureOfAddress    PRESENT
} )
```

5.5 Importing and exporting data types

5.5.1 Exporting

The following guidelines are appropriate when considering exporting of information elements:

a) any "Reference" (e.g. "typereference", "valuereference") which is used by several modules or is foreseen to be of possible interest to other domains shall be included in the export list of the module where they are defined;

b) if it is felt that all symbols can be exported, the EXPORTS keyword shall not appear in the module definition. This is equivalent to exporting every "Symbol" defined in the module.

5.5.2 Importing

The following guidelines are appropriate when considering importing of information elements:

Explicit imports of a "typereference" or a "valuereference" shall be used rather than an "externaltypereference" or a "externalvaluereference".

EXAMPLE: Define:

```
IMPORTS TypeA FROM Module-A;

TypeX ::= SEQUENCE {
        element1           INTEGER,
        element2           TypeA
}
```

rather than:

```
TypeX ::= SEQUENCE {
        element1           INTEGER,
        element2           Module-A.TypeA
}
```

5.6 Comments and user-defined constraints

When considering the use of ASN.1 comments and user-defined constraints the following guidelines are appropriate:

NOTE 1: The notation for User-defined constraints is not available to users of the 1988/90 standard.

a) When there is a need to specify some constraints on the contents of an information element beyond what can be expressed using ASN.1, the notation for user-defined constraint shall be preferred to ordinary comments. The latter are best suited to covey explanatory information. This is illustrated by the following example.

EXAMPLE 1: Define:

```
TBCD-STRING ::= OCTET STRING (CONSTRAINED BY {
        -- two digits per octet, each digit encoded 0000 to 1001 (0 to 9),
        -- 1010 (*), 1011 (#), 1100 (a), 1101 (b) or 1110 (c); 1111 used
        -- as filler when there is an odd number of digits.
        -- bits 8765 of octet n encoding digit 2n
        -- bits 4321 of octet n encoding digit 2(n-1) +1 --
        } )
        -- This type (Telephony Binary Coded Decimal String) is used to
        -- represent several digits from 0 through 9, *, #, a, b , c, ...
```

rather than:

```
TBCD-STRING ::= OCTET STRING
        -- This type (Telephony Binary Coded Decimal String) is used to
        -- represent several digits from 0 through 9, *, #, a, b , c, two
        -- digits per octet, each digit encoded 0000 to 1001 (0 to 9),
        -- 1010 (*), 1011 (#), 1100 (a), 1101 (b) or 1110 (c); 1111 used
        -- as filler when there is an odd number of digits.
        -- bits 8765 of octet n encoding digit 2n
        -- bits 4321 of octet n encoding digit 2(n-1) +1
```

NOTE 2: The above transformation can be made to existing specifications without any impact on the encoding.

b) In case of protocols based on Remote Operations, the specification may indicate that some user error (e.g., unexpectedDataValue) should be returned in case such constraints are violated. In such a case, it is recommended that the user-defined constraint be followed by an exception specification.

EXAMPLE 2:

```
Status ::= OCTET STRING (CONSTRAINED BY
                { -- bit 8-5: 0000 (unused) --
                } !Error: unexpectedDataValue)

Error ::= ENUMERATED {unexpectedDataValue(0)}
```

NOTE 3: The addition of an exception specification to an existing definition does not have any impact on the encoding.

c) When the constraint depends on the value of another information element the user defined constraint should include a formal parameter.

EXAMPLE 3:

```
ExternalSignalInfo {ProtocolId} ::= SEQUENCE {
        protocolId    ProtocolId,
        signalInfo    OCTET STRING (CONSTRAINED BY {
                -- contains the complete encoding according to protocol Id --
                ProtocolId} )
}
```

NOTE 4: The above example only make sense if there is a need to maintain compatibility with an existing specification. If a new protocol were to be defined, the semantic of the above type is better provided trough the use of the EMBEDDED PDV type.

d) When it is expected that the user-defined constraint can be used to automatically invoke some user-specific code for checking the constraint, it is recommended that it includes a reference to the checking procedure to be invoked. This reference can be composed by a keyword (e.g. EXTERNAL-CHECKING) followed by an object identifier which unambiguously identifies the checking procedure.

EXAMPLE 4:

```
AddressString ::= OCTET STRING (CONSTRAINED BY {
      -- EXTERNAL-CHECKING id-check-addressString -- } )
```

EXAMPLE 5: The following Information Object Class may be used by protocol designers to document the specific checking procedures they define:

```
EXTERNAL-CHECKING ::= CLASS {
        &ConstrainedType        ,
        &rules                  PrintableString,
        &id                     OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX {
        TYPE                    &ConstrainedType
        CHECKING RULES          &rules
        IDENTIFIED BY           &id
}
```

The user-defined constraint for checking that the contents of an octet string contains an address formed by a numbering plan indicator followed by a series of digits.

```
addressStringCheck      EXTERNAL-CHECKING ::= {
TYPE                    OCTET STRING
CHECKING RULES          "First octet encoded numbering plan according to
                        Rec Q.763 Subsequent octets encoded as
                        two digits per octet, each digit encoded 0000 to
                        1001 (0 to 9), 1010 (*), 1011 (#), 1100 (a), 1101 (b)
                        or 1110 (c); 1111 used as filler when there is an odd
                        number of digits.
                        bits 8765 of octet n encoding digit 2n
                        bits 4321 of octet n encoding digit 2(n-1) +1"
IDENTIFIED BY           id-check-addressString
}
```

5.7 Information elements dependencies

A protocol designer commonly needs to specify that the type of an information element depends on the value of another information element (classifier). The following guidelines are appropriate when considering the specification of dependencies between information elements:

NOTE: These guidelines are not appropriate to users of the 1988/90 notation which have to specify such dependencies using ordinary comments or a combination of the MACRO notation and the ANY DEFINED BY type.

a) The related information elements should be modelled as fields of an Information Object Class. Instances of that class shall then be defined to associate a particular value of the classifier element with the appropriate types for the other elements. This is illustrated by the following example.

EXAMPLE 1:

```
SUPPLEMENTARY-SERVICE ::= CLASS {
        &Subscription           ,
        &Registration           OPTIONAL  ,
        &code                   INTEGER UNIQUE
}
WITH SYNTAX {
        SUBSCRIPTION INFO       &Subscription
        [REGISTRATION INFO      &Registration]
        IDENTIFIED BY           &code
}

CallForwarding SUPPLEMENTARY-SERVICE ::= {
        SUBSCRIPTION INFO       ForwardingOptions
        REGISTRATION INFO       ForwardedToNumber
        IDENTIFIED BY           1
}
```

b) The PDU which carries the related information elements shall be specified using the notation for component relation constraints, as illustrated below.

EXAMPLE 2:

```
SupplServiceInfo ::= SEQUENCE {
        id                  SUPPLEMENTARY-SERVICE.&code,
        subscriptionInfo    SUPPLEMENTARY-SERVICE.&suscription
                            ({SupplServiceSet} {@id}),
        registrationInfo    SUPPLEMENTARY-SERVICE.&registration
                            ({SupplServiceSet} {@id}) OPTIONAL
}
-- SupplServiceSet represents the set of supported objects of class
-- "SUPPLEMENTARY-SERVICE"
```

5.8 Miscellaneous

5.8.1 Elements and types

The following guidelines are appropriate when considering instances versus types:

An application deals with a great number of parameters some of them being mapped to a protocol information element by the protocol machines. However there is no need to define a distinct data type for each of these information elements. When two information elements are syntactically equivalent they shall be represented as occurrences of the same data type, or if required for decoding, as occurrences of two isomorphic types derived from the same base type by context-specific tagging. The base type shall be defined only in one place.

EXAMPLE: Define:

```
Address ::= -- some suitable definition

Message ::= SEQUENCE {
        calledNumber        Address,
        callingNumber       Address,
        time                Time
}
```

rather than:

```
Message ::= SEQUENCE {
        calledNumber        CalledNumber,
        callingNumber       CallingNumber,
        time                Time
}

CalledNumber ::= -- A suitable definition
CallingNumber ::= -- The same definition
```

5.8.2 Order of elements

The following guidelines are appropriate when considering the order of information elements:

The order of elements when defining a type derived from the sequence type or the set type is up to the protocol designer. However, when possible (i.e. if there is no special need for an other logical order), it is recommended to group all the mandatory elements at the beginning of the construct.

This may allow an optimized coding of constructed data values when other encoding rules than the Basic Encoding Rules (e.g. Packed Encoding Rules (PER), see ITU-T Recommendation X.691 [6]) are used.

EXAMPLE: Define:

```
Example ::= SEQUENCE {
        element1            Type1,
        element2            Type2,
        element3            Type3,
        element4            Type4 OPTIONAL,
        element5            Type5 OPTIONAL
}
```

rather than:

```
Example ::= SEQUENCE {
        element1            Type1,
        element2            Type2,
        element4            Type4 OPTIONAL,
        element3            Type3,
        element5            Type5 OPTIONAL
}
```

5.8.3 Specification of nested structures

The following guidelines are appropriate when considering the specification of nested structures:

When defining a type derived from a sequence type, a set type or a choice type, the protocol designer should avoid to expand the definition of the component type in the definition of the structured type where the component appears.

EXAMPLE: Define:

```
DataUnit ::= SEQUENCE {
        element1            INTEGER,
        element2            TypeA OPTIONAL
}

TypeA ::= SEQUENCE {
        u1                  [0] IMPLICIT INTEGER,
        u2                  [1] IMPLICIT INTEGER
}
```

rather than:

```
DataUnit ::= SEQUENCE {
        element1            INTEGER,
        element2            SEQUENCE {
                            u1      [0] IMPLICIT INTEGER,
                            u2      [1] IMPLICIT INTEGER} OPTIONAL
}
```

5.8.4 Enumerated types

It is recommended that each "EnumerationItem" be an "identifier" rather than a "NamedNumber".

EXAMPLE: Define:

```
Type-A ::= ENUMERATED {
        item1,
        item2,
        item3,
        item4
}
```

rather than:

```
Type-A ::= ENUMERATED {
        item1 (0),
        item2 (1),
        item3 (2),
        item4 (3)
}
```

5.8.5 Specification of operations and errors

Existing signalling protocols based on TC or ROSE have been defined in terms of Operations and Errors, using the ASN.1 MACROs provided in ITU-T Recommendation Q.773 [13] and CCITT Recommendation X.219 [12].

Since the MACRO notation does not belong to the current notation, it is strongly recommended that Operations and Errors specified for new protocols be defined as instances of the Information Object Classes contained in ITU-T Recommendation X.880 [16]. Guidance for migrating from the MACRO notation to its replacement is provided in annex C of ITU-T Recommendation X.880 [16].

It should also be noticed that the new notation provides a formal way to specify whether an operation argument (or a result parameter, or an error parameter) is optional or mandatory.

6 Leaving holes in specifications

6.1 General aspects

There exist a number of circumstances where there is a need to leave holes in the specification of a set of protocol data units. This ETR identifies the following four main cases:

1) need to make provision for extension of the protocol;

2) need for embedding information from another protocol;

3) need to carry information whose syntax is not known to the designer of the main protocol (e.g., private extensions) and/or is expected to evolve independently from this main protocol;

4) need to define a generic PDU whose definition is expected to be refined in other specifications.

The first amendment to ITU-T Recommendation X.680 [7] and the first amendment to ITU-Recommendation X.681 [8] provide a mean for supporting the first type of need. This is known as the "ellipsis notation" and is further discussed in subclause 8.2.

As far as cases 2) and 3) are concerned, ASN.1 offers three possible notations:

- the EXTERNAL type;
- the EMBEDDED PDV type;
- the Instance-Of type.

NOTE 1: The last two alternatives are not available to users of the 1988/90 notation.

The use of parameterized types supports the last requirement (case 4)

NOTE 2: This facility is not available to users of the 1988/90 notation.

6.2 Embedding information

The following general guidelines are appropriate when considering the use of the above types:

a) the use of the EXTERNAL type is deprecated for the design of new protocols;

b) when the external information is related to a different protocol (i.e. embedding of one protocol into another), it is recommended to use an EXTERNAL or EMBEDDED PDV type;

c) when the external information must be encoded using different rules than the embedding protocol, it is necessary to use an EXTERNAL or EMBEDDED PDV type;

d) in all other situations the protocol designer shall prefer Instance-Of types which produce less overhead than EMBEDDED PDV when their values are encoded using the Basic Encoding Rules. The use of the Instance-Of type is illustrated below:

EXAMPLE 1: Given the following main definition:

```
SubscriberProfile ::= SEQUENCE {
        directoryNumber             IsdnAddressString,
        category                    Category DEFAULT {ordinary},
        supplementaryServices       SET OF SupplementaryServiceInformation,
        operatorSpecificServices SET OF OperatorSpecificService
}
```

OperatorSpecificService can be defined as an Instance-Of type as follows:

```
OperatorSpecificService ::= INSTANCE OF OPERATOR-SPECIFIC-SERVICE

OPERATOR-SPECIFIC-SERVICE ::= TYPE-IDENTIFIER
```

Which is equivalent to the following expanded definition:

```
OperatorSpecificService ::= [UNIVERSAL 8] IMPLICIT SEQUENCE {
        type-id      OPERATOR-SPECIFIC-SERVICE.&id
        value        OPERATOR-SPECIFIC-SERVICE.&Type ({@type-id})
}
```

Particular instances can be defined as follows:

```
JupiterFreephone OPERATOR-SPECIFIC-SERVICE ::= {
        SpecificServiceInfo
        IDENTIFIED BY {  iso identified-organization jupiter-telecom (100)
                         supplementaryService (2) freephone (11)}
}
```

NOTE 1: The BER encoding of a value of an instance-of type is compatible with the BER encoding of the equivalent value of an EXTERNAL type when the "Single-ASN1-type" encoding option is used.

e) due to the absence of presentation layer in the current signalling environment, it is essential that no presentation context information be used in external types and embedded-PDV types;

f) it is recommended that the following two subtypes be used by designers of signalling protocols in place of the EXTERNAL and EMBEDDED PDV types;

NOTE 2: The following definitions are not available to users of the 1988/90 notation.

EXAMPLE 2:

```
SIG-EXTERNAL ::= EXTERNAL (WITH COMPONENTS {
                ...,
                identification (WITH COMPONENTS {
                        ...,
                        presentation-context-id          ABSENT,
                        context-negociation              ABSENT
} ) } )

SIG-EMBEDDED-PDV ::= EMBEDDED-PDV (WITH COMPONENTS {
                ...,
                identification (WITH COMPONENTS {
                        ...,
                        presentation-context-id          ABSENT,
                        context-negociation              ABSENT
} ) } )
```

g) a protocol designer who wants to make use of an external or embedded PDV type shall first define an abstract syntax which encompasses all the data values which may be carried by this external or embedded-PDV type. Then he shall assign a name (i.e. an object identifier value) to this abstract syntax.

This name is used as the value of the "identification.syntaxes.abstract" element of the "EmbeddedPdvValue" or of the "identification.syntax" element of the "ExternalValue".

NOTE 3: When an external type is used, the transfer-syntax associated with a particular abstract-syntax is supposed to be agreed "a priori" between the peers. The embedded PDV type allows an explicit identification of the transfer syntax.)

NOTE 4: This abstract syntax is independent from the one to which the containing type belongs to. Thus when defining a set of data types whose values are to be carried within the value of an external type there is no need to worry about possible clashes of the new tag values with the ones already used by the protocol.

NOTE 5: The notation for defining values of the external type has been changed. Users of the 1998/90 notation shall refer to annex B.

h) it is recommended that the abstract-syntaxes be defined in terms of a single ASN.1 type (generally a choice type) which encompasses all the values which make up an abstract syntax, using the ABSTRACT-SYNTAX Information Object Class.

EXAMPLE 3:

```
my-abstract-syntax ABSTRACT-SYNTAX ::= {
        My-PDUs IDENTIFIED BY        {object identifier value }
}

My-PDUs ::= CHOICE {
        initialPDU    [APPLICATION 0] Type-A,
        finalPDU      [APPLICATION 1] Type-B
}
```

NOTE 6: This feature is not available to users of the 1988/90 notation which may use comments to specify the association between the abstract-syntax-name and the type which encompasses the data values.

6.3 Defining generic types

The following guidelines are appropriate when considering the use of the above types:

a) When there is a need to leave undefined the type of some components of a standardized structured type, it is possible to define this structured type as a parameterized type.

EXAMPLE 1:

```
GenericPDU {DummyType} ::= SEQUENCE {
        a             INTEGER,
        b             BOOLEAN,
        c             DummyType OPTIONAL
}
```

b) It is recommended that such dummy references be resolved before or when an abstract-syntax is defined using such parameterized types.

EXAMPLE 2:

```
my-abstract-syntax ABSTRACT-SYNTAX ::= GenericPDU {SpecificType}
IDENTIFIED BY {xxx}

-- note that this is equivalent to:

my-abstract-syntax ABSTRACT-SYNTAX ::= SpecificPDU
IDENTIFIED BY {xxx}

-- where

SpecificPDU ::= GenericPDU {SpecificType}

-- which in turn is equivalent to:

SpecifcPDU ::= SEQUENCE {
        a               INTEGER,
        b               BOOLEAN,
        c               SpecificType OPTIONAL
}
```

7 Protocol modifications

This clause describes the different categories of changes which can be made to a protocol specification as far as the protocol data units are concerned. Subclause 8.1 deals with changes to abstract-syntaxes while subclause 8.2 discusses their impact on transfer-syntaxes when the Basic Encoding Rules are used.

7.1 Changes to abstract-syntaxes descriptions

This subclause identifies three types of changes which can be made to abstract-syntax specifications (i.e. changes to the type(s) in term of which the abstract-syntax is defined).

7.1.1 Non compatible changes

Changes cause incompatibility from an abstract-syntax point of view when a value of the original abstract-syntax is not a valid value for the new abstract-syntax.

A non compatible change to the type(s) in term of which an abstract-syntax is defined causes an incompatibility between the original abstract-syntax and the new one.

The following list provide some examples of such non compatible changes:

- replace a type by another type even if the tag remains the same;
- remove a type definition or a value definition referred either explicitly (IMPORTS) or implicitly (ANY DEFINED BY of TC);
- remove a "NamedType" from the "AlternativeTypeList" of a Choice type;
- remove an "EnumerationItem" from the "Enumeration" of an enumerated type;
- restrict the "ValueRange" of an integer type;
- restrict the "SizeConstraint" of a string type;
- restrict the "SizeConstraint" of a sequence-of type;
- change the order of elements in the "ComponentTypeList" of a sequence type;
- make any combination of the above changes to one or more components of a structured type.

7.1.2 Changes without impact on the abstract-syntax

These changes are purely restricted to the way the abstract syntax and the type used for its definition are specified. They do not affect the set of values defined by the abstract syntax. Such changes may be needed to bring a specification in line with the rules stated in clauses 5 and 6 of this ETR (this list is not necessarily complete).

a) In a set type or sequence type, replace the use of "COMPONENTS OF" with direct inclusion of the equivalent components, or vice versa.

b) In a choice type, replacing nested choice types with direct inclusion of the each "NamedType" which appear in the "AlternativeTypeList".

NOTE 1: This transformation affects the abstract-syntax when automatic tagging is used.

c) Replace a type by a "typereference" representing the same type or vice versa.

d) Replace a value by a "valuereference" representing it or vice versa; this includes replacing a "number" by a "valuereference".

e) Replace a type by an equivalent selection type or vice versa.

f) Add or (if unused) remove one or more "NamedBit" from a bit string type.

g) Add or (if unused) remove one or more "NamedNumber" from an integer type.

h) Change the spelling of a "Reference" (e.g. "typereference", "modulereference", "valuereference", etc.) or an "identifier" consistently throughout all ASN.1 Modules. This includes adding identifiers where allowed from the syntax.

i) Split up one ASN.1 Module into several ASN.1 Modules.

j) Put several ASN.1 Modules together into one ASN.1 Module.

k) Move parts of one ASN.1 Module into another ASN.1 Module.

l) Add one or more "Symbol" to the "EXPORTS" list (or remove the "EXPORTS" statement to indicate that everything is exported).

m) Add one or more "Symbol" to the "IMPORTS" list (symbols from ASN.1 Modules already referenced in the "IMPORTS" list as well as symbols from newly referenced ASN.1 Modules).

n) Remove one or more existing "Assignment" (e.g., "TypeAssignment", "Valueassignment") if their associated "Reference" is never used throughout all ASN.1 Modules.

o) Add a field to an object class definition if it is associated with a "Type" or "DefinedObjectClass" whose values are already encompassed by the abstract-syntax.

NOTE 2: This includes adding an ERROR object of class ERROR to the definition of an OPERATION object if this ERROR object is already used in the definition of another OPERATION object.

7.1.3 Extension of an abstract syntax

An abstract-syntax is extended if its associated type is extended (i.e. if a choice type, it can be extended by adding a new component or extending an existing one). One way of extending a PDU (or any structured type) is to extend the type of any of its components.

One ASN.1 type is considered to be an extension of another if the former includes all the values of the latter, and possibly some others.

Given a certain type, its extensions are those types which could be derived by one or more of the following changes, combined with any number of those described in subclause 8.1.3.

a) Change a single type into a choice type which includes this single type in the "AlternativeTypeList".

NOTE 1: The tag of this alternative has to remain unchanged, no additional (EXPLICIT) Tag is allowed. It has to be taken care, that all references to this changed type throughout all ASN.1 Modules still meet all ASN.1 requirements - in particular distinctness of Tags and uniqueness of Identifiers.

b) Add one/more "NamedType" to the "AlternativeTypeList" of a choice type.

NOTE 2: See note 1 for changing a single type into a choice type.

c) Add an optional component to a sequence type or a set type.

NOTE 3: When automatic tagging is employed or when it is expected that transfer syntaxes which do not transmit tags (e.g. PER) will be used, a new component should be added at the end of the sequence.

NOTE 4: When it is expected that transfer syntaxes which do not transmit tags (e.g. PER) will be used, the tags of a new component should be in ascending order.

d) Add a default component to a sequence type or a set type.

NOTE 5: See notes 3 and 4.

e) Extend one or more components of a choice type, sequence type or a set type.

f) Extend the type in terms of which a sequence-of type or a set-of type is defined.

g) Change a mandatory component of a sequence type or set type to an optional or default component.

NOTE 6: The Tag of this component has to remain unchanged and distinctness of Tags has to be ensured.

h) Add one or more new "EnumerationItem" to an enumerated type.

NOTE 7: Distinctness of values and uniqueness of identifiers has to be ensured.

i) Extend the "ValueRange" of an integer type by decreasing the "LowerEndpoint" and/or increasing the "UpperEndpoint".

j) Extend the "SizeConstraint" of an octet string type, a bit string type or a character string type by decreasing the "LowerEndpoint" and/or increasing the "UpperEndpoint".

k) Extend the "SizeConstraint" of a sequence-of type or a set-of type by decreasing the "LowerEndpoint" and/or increasing the "UpperEndpoint".

l) Change the "Value" assigned to a "valuereference" if the effect for all references still meets all other rules (e.g. Increase a Value used only as "UpperEndpoint" in a "ValueRange" or "SizeConstraint").

The following changes to OPERATION and ERROR definition affect the abstract-syntax formed by the set of values whose type is the one of TC messages or ROSE PDUs parameterized by a specific list of operations:

m) Add new definition of class OPERATION and ERROR respectively as long as they are distinct with all other definitions of class OPERATION and ERROR respectively.

n) Add an &Type field an object of class OPERATION if it didn't have such a field.

o) Add an &ResultType field to an object of class OPERATION if it did not have such a field.

p) Add an &ParameterType field to an object of class ERROR if it didn't have such a field.

7.1.4 Private extensions

Private extensions are those added to standard protocols outside standardization bodies (e.g. national specifications). The actual use of private extensions outside the domain for which they are defined may cause incompatibility problems if no forward compatibility rules are specified for the standard protocol which has been extended.

The following guidelines are appropriate when considering private extensions:

a) a private extension should be a valid extension according to the rules described in subclause 7.1.3;

b) a private extension should be defined in such a way that it is ensured that its element can be distinguished from new elements introduced in future version of a protocol;

c) if it is felt that a PDU type may require private extensions, the protocol designer should insert a specific information element for that purpose, in any construct which may be extended.

   The type for such an information element should preferably be an Instance-Of type (see subclause 6.2).

7.2 Impact on the transfer-syntax

The impact of changes to abstract-syntaxes on transfer syntaxes depends on the set of encoding rules used to derive the transfer-syntax. This subclause deals only with the situation where the Basic Encoding Rules (ITU-T Recommendation X.690 [5]) are used.

7.2.1 Non compatible changes

In general, a non-compatible change from an abstract-syntax point of view causes a non-compatible change from a transfer-syntax point of view. However for a given set of encoding rules there may be some exceptions.

In addition it is obvious that changing the encoding rules causes in most cases incompatibility from a transfer-syntax point of view.

Note that changing an IMPLICIT tag to an EXPLICIT tag causes a protocol incompatibility when BER are used but not necessarily when the Packed Encoding Rules (ITU-T Recommendation X.691 [6]) are used.

7.2.2 Changes without impact on transfer-syntaxes

As a general rule, changes which do not affect the abstract-syntax, do not affect the transfer-syntax (providing that the encoding rules are unchanged).

This is always true with the Basic Encoding Rules (ITU-T Recommendation X.690 [5]). However, for the Packed Encoding Rules (ITU-T Recommendation X.691 [6]), one major exception would be rule 7.1.2 b) where in case of nested choices, the inner choice type is replaced by the direct inclusion on its components. In addition, there may be some situations where a modification of the abstract-syntax does not cause a modification to the transfer syntax.

The following example illustrates a situation where a non-compatible change is made to a type definition without affecting the transfer-syntax when BER are used:

Replace an integer type by an enumerated type if this integer type is only used to define tagged types.

EXAMPLE: Change:

```
Colour ::= [1] IMPLICIT INTEGER {red (0), blue (1), white(2)} (0..2)
```

to:

```
Colour ::= [1] IMPLICIT ENUMERATED {red (0), blue (1), white(2)}
```

7.2.3 Extension of a transfer-syntax

The Basic Encoding Rules ensure that if the abstract-syntax is extended, the transfer-syntax is also extended and the original values are preserved.

However, for the Packed Encoding Rules (ITU-T Recommendation X.691 [6]), one major exception would be rule 7.1.3 a) where a a type is replaced by a choice type which contains this type.

8 Compatibility issues

The following subclauses deal only with compatibility from the encoding point of view. They do not deal with functional compatibility aspects which have also to be taken into account when extending or modifying a protocol.

8.1 Backward compatibility

The purpose of this subclause is to provide guidelines on the types of changes which can be made to a protocol specification to while ensuring backward compatibility.

Changes which do not affect the transfer-syntax (i.e. the bits and bytes exchanged between peer entities) or which extend it are backward compatible.

Using simple words, backward compatibility means that an encoded PDU of the original protocol is a valid encoded PDU for the new protocol. Such changes are described in subclauses 7.2.2 and 7.2.3.

Non backward compatible changes are those which affect the transfer-syntax in a non compatible way. In this case an encoded PDU of the original protocol is not necessarily a valid encoded PDU for the new protocol. Example for such changes are listed in subclause 7.2.1

8.2 Forward compatibility

Changes to an abstract-syntax affect in most cases the transfer syntax.

If changes are made according to the rules indicated in the previous subclauses, the new protocol is backward compatible with the original one. However an encoded PDU of a new version of this protocol is not necessarily a valid PDU for the original protocol.

Forward compatibility is most generally achieved through application-context negotiation. However in order to minimise the number of protocol fall-backs on signalling interfaces, it will sometimes be necessary to define forward compatibility rules which allow a version of a protocol to accept protocol data units generated by a future version without having to provide a new application-context-name.

This feature is also necessary where application context negotiation is not supported, e.g., the optional dialogue portion of TC is not supported.

If the protocol designer wishes to ensure that a value of a PDU of the new version of a protocol be (at least) always partly recognised by an implementation of an older version of this protocol, the following guidelines shall be followed:

- the new protocol version shall complies with the rules described in subclauses 8.1.2 and 8.1.3 (i.e. the new abstract syntax shall be an extension of the old one);

- the applicable encoding rules (which shall be unchanged) shall permit the unknown parts of the encoding to be delimited[2];

- Extensibility rules shall be included in the specification of the original protocol so that additions not be treated as errors during the decoding process.

If the last recommendation is not followed, the behaviour of the receiving entity is implementation dependent.

It is recommended to restrict the use of the extensibility rules to:

- the addition of OPTIONAL and DEFAULT components in types derived from the SEQUENCE or SET type;

- the addition of alternatives to a CHOICE type, providing that it does not correspond to a mandatory component of a higher structure;

- the addition of enumerated values to an ENUMERATED type, providing that it does not correspond to a mandatory component of a higher structure;

- the relaxation of a constraint, providing that the constrained type does not correspond to a mandatory component of a higher structure.

The protocol designer shall be aware that extensibility rules beyond those listed here (e.g., relaxing a size constraint or a value range, or extending a CHOICE type corresponding to a mandatory element) may require special care (e.g., specifying error codes to be used when an unrecognized information element is encountered) to avoid important functional errors.

The first amendment to ITU-T Recommendation X.680 [7] enables the protocol designer to indicates in a module header that extensibility rules apply to each type for which it is permitted. However this feature shall be use cautiously since it does not take into account the above restrictions. It also provides a new piece of notation to flag individually the types which can be extended in such a way that unknown additions be ignored. This flag is an ellipsis (...) and is called an extensions marker. It is recommended that protocol designers make use of this facility for the specification of extensibility rules. Users of the 1988/90 version of ASN.1 may include this flag as an ASN.1 comment.

2 This is always ensured if the Basic Encoding Rules (or any variant) or the Packed Encoding Rules are employed.

EXAMPLE: The following illustrates a specification where extra (unknown) element values will be accepted for PDU-A and TypeA, but not for TypeB.

```
PDU-A ::= SEQUENCE {
        element1     TypeA,
        element2     TypeB,
        ...
}

TypeA ::= SEQUENCE {
        element3     TypeC,
        element4     TypeD,
        ...
}

TypeB ::= SEQUENCE {
        element5     TypeE,
        element6     TypeF
}
```

The extensibility mechanism may also be required when application layer relays are involved in the exchange of messages, if they support a lower version of the protocol than the one used by the actual senders and receivers of these messages (e.g., Vn-Vn dialogue relayed by a Vn-1 node). However, in this case, there is an additional requirement that the relay node passes the received unrecognized information to the subsequent node.

The specification of extensibility rules without any other indication does not place any requirement the receiving entity regarding further processing of the part of encoding which corresponds to unknown values. In the absence of any additional specification, the undecoded parts are ignored.

If there is a need to take any other special action (e.g. return a specific error, retransmit this part of the encoding to a third party, etc.) the protocol designer shall provide an explicit specification of the expected behaviour, in addition to the extensibility rules. As a possible option, it can include in the original version of the protocol some information elements which convey dynamically an indication of the behaviour to be followed on receipt of unrecognized information elements.

9 Changing names of information objects

This clause gives guidelines on the criteria to be used for changing the name of an information object (or more precisely for deciding that a new information object has been created). It focuses only on syntax related aspects. However the protocol designer shall be aware that other types of changes may have impact on the names of information objects (e.g. addition or removal of an ASE to an AC).

9.1 Module names

A module is identified by a a "modulereference" and optionally an "ObjectIdentifierValue". This value shall be changed if one or more changes which affect directly or indirectly a symbol visible outside the module or used for defining an abstract-syntax are made.

It is recommended that the last arc of the object identifier values used for representing module names be a version number.

9.2 Abstract syntax names

An abstract-syntax should be allocated a new name if a new type is added to the list of types in term of which it is defined or if the set of data values associated with one of the existing types is modified and the new type is not extension-related to the old type.(e.g. because one of these types is extended and does not contain an extensions marker).

It recommended that the last arc of the object identifier values used for representing abstract syntax names be a version number.

9.3 Application context names

An application-context should be allocated a new name if:

a) one or more associated abstract-syntaxes are modified;

b) one or more abstract-syntaxes are removed from the set of associated abstract-syntaxes;

c) one or more abstract-syntaxes are added to the set of associated abstract-syntaxes.

NOTE: This covers modifications where a new abstract-syntax is added to the application-context in relation to a particular use of an external type which appear in one the PDU of the main abstract-syntax.

It is recommended that the last arc of the object identifier values used for representing application contexts be a version number.

Annex A: Migration from 1988/1990 notation to 1994 notation

Guidance for the migration from the 1988/1990 notation to the current notation is given in annex A of ITU-T Recommendation X.680 [1].

More specific guidance for operations and errors is provided in ITU-T Recommendation X.880 [16].

Annex B: Specific guidance for users of the 1988/1990 notation

B.1 Use of identifiers

Although the use of identifiers is not mandatory from a syntactical point of view in the 1988/90 version of ASN.1, it is recommended, when defining a type derived from the sequence, set and choice types, that each element be allocated an identifier. This makes the specification more readable and eases the parsing of the value notation (e.g. when used in a TTCN test case description). It may also be necessary for defining unambiguously a value of such a type.

EXAMPLE:

```
Example ::= SEQUENCE {
        calledNumber        Address,
        duration            Duration OPTIONAL,
        time                Time
}

Address ::= OCTET STRING (SIZE(1..10))
Duration ::= INTEGER (0..30) -- mn

Time ::= SEQUENCE {
        hour                Hour,
        min                 Min
}

Hour  ::= INTEGER (0..23)
Min   ::= INTEGER (0..59)

exampleValue1 Example ::= {
        calledNumber        "1122334455"H,
        duration            10,
        time          {
                hour        3,
                min         25}
}
```

B.2 Choice and Any values

CCITT Recommendation X.208 [9] contained an erroneous value notation for choice and any types. In some specific circumstances, the use of this notation resulted in unparsable modules. A new notation has been published in 1991 as a corrigendum to the ISO/IEC standard and is now integrated in ITU-T Recommendation X.680 [1].

It is recommended that users of the 1988/90 notation apply the ISO corrigendum, irrespective of the version of ASN.1 to which they claim to conform. This correct notation is reproduced below:

```
ChoiceValue ::= identifier : Value

AnyValue ::= Type : Value
```

EXAMPLE: This example illustrates the use of the correct value notation by defining to valid values for Type-A.

```
Type-A ::= CHOICE {
        element1            INTEGER,
        element2            [0] ANY
}

value1 Type-A ::= element1 : 1
value2 Type-A ::= element2 : BOOLEAN : TRUE
```

B.3 Tagging

Each ASN.1 built-in type (simple type or type constructor) is given a UNIVERSAL tag assigned by ITU-T Recommendation X.680 [1]. However, the tagging mechanism provided by ASN.1 allows the user of this notation to create a new type (tagged type) by assigning a new tag to an existing type (base type). Both types are isomorphic (same syntax) and differ only by virtue of their tags.

EXAMPLE 1:

```
-- NewTaggedType is derived from built-in OCTET STRING type by tagging.

NewTaggedType ::= [APPLICATION 10] OCTET STRING
```

Theoretically, the user is allowed to assign a tag to any or all the types he defines. However, this is only necessary when the original tag does not unambiguously identify values of the type.

The following guidelines are appropriate when considering tag allocation:

a) A tag is not a way to identify an application parameter uniquely throughout an application. Tagging should only be used when necessary to avoid possible ambiguities at decoding time. This is the case where ASN.1 requires distinct tags in a structured type but the original tags associated with its component types do not fulfil this requirement.

   This occurs mainly when:

   - the information elements are members of a (non-ordered) set (i.e. a set type) and therefore their relative position cannot be used to discriminate between two information elements of the same type (thus with the same tag);

   - the information elements are members of an ordered set (i.e. a sequence type) but the presence or absence of optional elements makes impossible to discriminate between the presence of an optional element and the presence of an immediately following information element of the same type;

   - because two occurrences of the same type would appear in a choice type;

   - or any combination of the above situations.

EXAMPLE 2: Tagging is required for element 2.

```
DataUnit ::= SEQUENCE {
        element1            INTEGER,
        element2            [0] IMPLICIT INTEGER OPTIONAL,
        element3            INTEGER
}
```

EXAMPLE 3: Tagging is not required for element 2.

```
DataUnit ::= SEQUENCE {
        element1        INTEGER,
        element2        BOOLEAN OPTIONAL,
        element3        INTEGER
}
```

b) When tagging is necessary, the tags shall always be allocated in the context-specific class.

The main reason is that the context-specific class is the only one (when used in a correct manner) which, in the absence of a central registry for signalling protocols[3], ensure that there will never be any conflict between tag values, when data types are imported and exported between modules[4].

c) The protocol designer shall consider that the context (i.e. the scope of the tag value) is the context of the next higher construct (an embedding sequence type or set type or an explicit tagged type).

EXAMPLE 4: This example illustrates the re-usability of context-specific tags:

```
DataUnit ::= SEQUENCE {
        element1        INTEGER,
        element2        [0] IMPLICIT SEQUENCE{
                                u1      [0] IMPLICIT INTEGER,
                                u2      [1] IMPLICIT INTEGER} OPTIONAL,
        element3        [1] IMPLICIT TypeA OPTIONAL
}
TypeA ::= SEQUENCE {
        e1              [0] IMPLICIT INTEGER OPTIONAL,
        e2              [1] TypeB OPTIONAL
}
TypeB ::= SET {
        f1              [0] IMPLICIT OCTET STRING,
        f2              OCTET STRING
}
```

d) The scope of the tags used within the definition of a choice type is not limited to this choice. Thus in situation where ASN.1 required distinct tags, these tags shall be different from the other tags which appear in the next higher construct where this choice type is attached. To avoid worrying about these problems a protocol designer can decide to always define a tagged (explicit) type derived from the choice type.

EXAMPLE 5: This example illustrates a situation where uniqueness of tag values has been taken care of by the protocol designer. If this approach is followed it is recommended to reserve always the same set of tag values for this purpose:

```
DataUnit ::= SET {
        e1          TypeA,
        e2          [2] TypeB
}
TypeA ::= CHOICE {
        a1          [0] IMPLICIT INTEGER,
        a2          [1] IMPLICIT INTEGER
}
```

[3] This is also because there is currently no systematic mean in signalling systems to explicitly convey abstract-syntax-names or transfer-syntax-names.

[4] This rule may be overridden by protocol designers which define common building blocs protocols (e.g. TC itself (ITU-T Recommendations Q.771 to Q.775 [13]).

EXAMPLE 6: This example illustrates how the uniqueness of tag values is ensured by defining a tagged type derived from a choice type which limits the scope of the tags used in the "AlternativeTypeList".

```
DataUnit ::= SET {
        e1              [0] TypeA,
        e2              [1] TypeB
}
TypeA ::= CHOICE {
        a1              [0] IMPLICIT INTEGER,
        a2              [1] IMPLICIT INTEGER
}
TypeB ::= CHOICE { -- [0], [1] can be re-used
        a1              [0] IMPLICIT INTEGER,
        a2              [1] IMPLICIT INTEGER
}
```

EXAMPLE 7: This example illustrates a wrong specification where uniqueness of tag values is not ensured.

```
DataUnit ::= SET {
        e1              TypeA,
        e2              [0] TypeB
}
TypeA ::= CHOICE {
        a1              [0] IMPLICIT INTEGER,
        a2              [1] IMPLICIT INTEGER
}
```

e) Since it is assumed that the scope of context specific tags is the next higher construct, no tagged type shall be defined outside a construct (i.e. only data types derived from a universal type by subtyping or built from other types using a construction mechanism shall be defined outside any construct).

   This means that the examples which were given in the Blue Book version of the TC user's guide (CCITT Recommendation Q.775 [14]) shall not be considered as valid examples. However, the White Book version of ITU-T Recommendation Q.775 [13] includes valid examples.

f) In order to minimize the number of octets required for encoding, the IMPLICIT tagging method shall always be used when defining a tagged type (except if the base type is a choice type or an open type). This can be easily specified by including the "IMPLICIT TAGS" keywords in each module definition.

EXAMPLE 8: The following specification

```
Dummy-Module
DEFINITIONS IMPLICIT TAGS ::=
BEGIN
        -- module body

TypeA ::= SEQUENCE {
        element1            [0] TypeB,
        element2            [1] TypeC
}

END
```

is equivalent to:

```
Dummy-Module
DEFINITIONS EXPLICIT TAGS ::=
BEGIN
        -- module body

TypeA ::= SEQUENCE {
        element1            [0] IMPLICIT TypeB,
        element2            [1] IMPLICIT TypeC
}

END
```

NOTE: When the "TagDefault" value of a module is IMPLICIT it is not necessary to include the keyword EXPLICIT when defining a tagged type derived from a choice type or an any type because this is implicit. It is obviously not allowed to include the IMPLICIT keyword.

g) The meaning of an "empty" "TagDefault" has been changed between versions of ASN.1 specification. Therefore, it is suggested to always include a non "empty" "TagDefault" to avoid any ambiguity when defining a module.

h) It is recommended that the "TagDefault" indicates the tagging method used by the majority of the types defined in the module.

B.4 Operations and Errors

Guidance on the specification of Operations and Errors using MACROs is already available in Recommendation Q.775 [13] (TC) as well as CCITT Recommendation X.219 [12] (ROSE).

The following additional guidelines are appropriate when considering the use of MACROs for defining Operation and Errors:

a) In order to allow the re-use in other domain of Operations and Errors defined, the specification should be based on a two steps approach: First, Operation and Error types shall be defined. Then Operation and Errors of these types shall be defined by allocating operation and error codes.

   Using this method only makes sense if the operations and errors types are exported from the modules where they are defined.

b) In order to make easier the transition to the new notation, it is recommended that:

   - within the scope of an abstract-syntax, only one operation (error) value be assigned to a particular operation (error) type;

   - no semantics shall be attached to the potential conceptual difference between an operation (error) type and an operation (error) value. This is because the two-step approach is mainly an artificial mechanism which allows the protocol designer to re-arrange operation and error codes according to his requirements[5].

c) When defining a protocol (thus an abstract syntax) the protocol designer may import operation and error types (not values) from other domains where these types have been allocated a local value. The protocol designer is free to allocate other local values in order to ensure the uniqueness of these values within the new domain. If the macro-replacement notation is employed, the same kind of mechanism is provided through the use of the "recode" parameterized operation.

d) It is also recommended that the value assignment not be made in the modules where the types are defined.

[5] When the macro-replacement notation is used, an operation definition which has no associated operation code is not interpreted as an operation type but as a Bind operation.

e) It is suggested that the operation argument/result type or the error parameter type not be expanded in the Operation or Error type definition. This is in line with the new ROSE notation which require the use that only a "typereference" (as opposed to a "Type") be used when defining operations and errors.

It should also be noted that one of the advantages of giving a name to the type of the argument or result parameter (rather than expanding the type definition) is for test specification where it is convenient or indeed necessary (see ITU-T Recommendation X.292 [11]) to have names for the PDU of the tested layer/sub-layer.

EXAMPLE 1: This means that operation type is to be defined in the following form:

```
OperationA ::= OPERATION
PARAMETER               OperationA-Arg

RESULT                  OperationA-Res

ERRORS                  {ErrorA,
                        ErrorB}

OperationA-Arg ::= SEQUENCE {
        element1        Type1,
        elementX        TypeX OPTIONAL
}
```

rather than:

```
OperationA ::= OPERATION
PARAMETER               SEQUENCE {
        element1 Type1  elementX TypeX OPTIONAL}

RESULT                  OperationA-Res

ERRORS                  {ErrorA,
                        ErrorB}
```

OperationA-Arg, OperationA-Res, ErrorA-Par, etc. have to be defined using separated assignment. They can be regarded as the ROSE-user or TC-user Protocol Data Units.

This approach makes also easier the definition of several Operations which have the same argument type and/or result type but differs by another aspects (e.g. errors, class, etc.).

EXAMPLE 2: The two operation types differ only by their class:

```
-- TC class 1 operation type
OperationA ::= OPERATION
PARAMETER               OperationA-Arg

RESULT                  OperationA-Res

ERRORS                  {ErrorA,
                        ErrorB}

-- TC class 3 operation type
OperationB ::= OPERATION
PARAMETER               OperationA-Arg

RESULT                  OperationA-Res
```

History

Document history	
November 1992	First Edition
September 1995	Second Edition
February 1996	Converted into Adobe Acrobat Portable Document Format (PDF)

ISBN 2-7437-0123-4
Dépôt légal : Septembre 1995