ETSI TECHNICAL REPORT

ETR 060

November 1992

Source: ETSI TC-SPS

Reference: DTR/SPS-2001

ICS: 33.020, 33.040.40

Key words: SPS, 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 5

1 Introduction 7

2 References 7

3 Abbreviations 8

4 Overview of ASN.1 9

5 Specification of protocol data units 10
5.1 Modules 10
5.2 Tagging 10
5.2.1 Tag allocation 10
5.2.2 Tag classes 11
5.2.3 Implicit or explicit tagging 13
5.3 Handling of optional and default elements 14
5.4 Subtyping 14
5.5 Importing and exporting data types 15
5.5.1 Exporting 15
5.5.2 Importing 15
5.6 Miscellaneous 16
5.6.1 Use of identifiers 16
5.6.2 Instances and types 16
5.6.3 Order of elements 17
5.6.4 Specification of nested structures 17

6 Specification of operations and errors 18

7 Specification of abstract syntaxes 20
7.1 General aspects 20
7.2 Use of the external type 20

8 Protocol modifications 22
8.1 Changes to abstract-syntaxes descriptions 22
8.1.1 Non compatible changes 22
8.1.2 Changes without impact on the abstract-syntax 22
8.1.3 Extension of an abstract syntax 23
8.1.4 Private extensions 24
8.2 Impact on the transfer-syntax 25
8.2.1 Non compatible changes 25
8.2.2 Changes without impact on transfer-syntaxes 25
8.2.3 Extension of a transfer-syntax 25

9 Compatibility issues 25
9.1 Backward compatibility 25
9.2 Forward compatibility 26

10 Changing names of information objects 27
10.1 Modules names 27
10.2 Abstract syntax names 27
10.3 Application context names 27

Annex A: List of ASN.1 tools 29

History 32

Blank page

Foreword

ETSI Technical Reports (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 application of ETSs or I-ETSs, or which is immature and not yet suitable for formal adoption as an ETS or I-ETS.

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

Blank page

1 Introduction

The purpose of this 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 CCITT Recommendation X.208 [1] which specifies the Abstract Syntax Notation One (ASN.1). In case of misalignment, CCITT Recommendation X.208 [1] 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 as they are specified in CCITT Recommendation X.209 [2].

This ETR is not a tutorial on ASN.1. Tutorial matter exists on this subject (see references: A tutorial on Abstract Syntax Notation One [3], ASN.1 MACRO Facility [4], ASN.1 and ROS [5], An overview of ASN.1 [17]).

Throughout this ETR, the term "user" denotes a person who employs ASN.1 for protocol design.

Terms between quotation marks refer directly to items or productions defined by the ASN.1 specification (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 ISO DIS 9646 [8]) which makes a direct use of the ASN.1 type definitions of the protocol to be tested.

Annex A provides a non-exhaustive list of ASN.1 tools which may help protocol designers in writing formally correct ASN.1 specifications.

2 References

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

[2] CCITT Recommendation X.209 (1988): "Specification of basic encoding rules for abstract syntax notation one (ASN.1)" (Also published as ISO 8825).

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

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

[5] ASN.1 and ROS: The impact of X400 on OSI - James E. White, Member IEEE - IEEE Journal on Selected Areas In Communications, Vol 7, No7 - September 1989.

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

[7] CCITT Recommendations Q.771-Q.775 (1992): "Guidelines for using Transaction Capabilities" (White Book).

[8] ISO DIS 9646 (1989): "Part 3 - Tree and Tabular Combined Notation (TTCN)".

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

[10] CCITT Recommendation X.229 (1988): "Remote operations; Protocol specification".

[11] ISO/IEC CD 8824-1 (June 1991): "Basic ASN.1".

[12] ISO/IEC CD 8824-2 (June 1991): "Information Object Specification".

[13] ISO/IEC CD 8824-3 (June 1991): "Constraint Specification".

[14] ISO/IEC CD 8824-4 (June 1991): "Parameterisation of ASN.1 specifications".

[15] ISO/IEC CD 8825-2 (June 1991): "Packed Encoding Rules".

[16] Draft CCITT Recommendation X.9ros, CCITT Q.19/VII, Oslo June 1991.

[17] An overview of ASN.1 - Gerald Neufeld, Son Vuang - Computers and ISDN Systems - No23 (1992).

3 Abbreviations

AC	Application Context
ASE	Application Service Element
APDU	Application Protocol Data Unit
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
DSS.1	Digital Subscriber Signalling Number One
MAP	Mobile Application Part
PER	Packed Encoding Rules
PDU	Protocol Data Unit
ROSE	Remote Operation Service Element
SS7	Signalling System Number 7
TCAP	Transaction Capabilities Application Part
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 a table whose entry 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 CCITT 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, DSS.1 "layer-3" messages, ...). 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, ...

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 mean 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 instances of the types it defines. This is why, unlike the data declaration portion of programming languages, it provides inherently a mean for associating identification tags with data types.

CCITT Recommendation X.208 [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, ...

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 messages 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 ISO DIS 9646 [8]). 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, ...). 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 that the number of ASN.1 modules used to define the various protocol data units which form a protocol (i.e. an application-context) is independent from the number of ASEs from which this protocol is made. It has also no impact on the number of abstract syntaxes used by an application-context.

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 CCITT Recommendation X.208 [1], each ASN.1 module should be given a "moduleIdentifier". This is used as a formal reference when exporting or importing data types 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 an identifier of type OBJECT IDENTIFIER. Unlike an application-context-name or an abstract-syntax-name, this identifier is never exchanged between peer protocol machines.

For further guidance on assigning "ObjectIdentifierValue" to an ASN.1 module, see "An overview of ASN.1" [17].

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

<Protocol-Name>-<Qualifier>

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, ...)

5.2 Tagging

5.2.1 Tag allocation

Each ASN.1 built-in type (simple type or type constructor) is given a UNIVERSAL tag assigned by CCITT Recommendation X.208 [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 differs only by virtue of their tags.

```
Example:

-- NewTaggedType is derived from the 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 these types.

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. These are the situations where ASN.1 requires distinct tags in a structured type but the original tags associated with its components 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.

In this first example, tagging is required for element 2:

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

In this second example, tagging is not required for element 2:

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

5.2.2 Tag classes

The following guidelines are appropriate when considering tag classes:

a) 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[1)], ensure that there will never be any conflict between tag values, when data types are imported and exported between modules[2)].

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

---

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

2) This rule may be overridden by protocol designers which define common building blocs protocols (e.g. TCAP itself (CCITT Recommendations Q.771-Q.775 [6]), CCITT Recommendations Q.771-Q.775 [7]).

The following 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] TypeC OPTIONAL}

TypeC ::= SET{
        f1              [0] IMPLICIT OCTET STRING,
        f2              OCTET STRING}
```

c) 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 construction 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.

The following 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}
```

This second 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}
```

This third 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}
```

d) Since it is assumed that the scope of context specific tags is the next higher construction, no tagged type shall be defined outside a construction (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 construction).

This means that the examples given in the blue book version of the TCAP user's guide (CCITT Recommendations Q.771-Q.775 [6]) shall not be considered as valid examples. However the white book version of CCITT Recommendation Q.775 [7] will include valid examples.

5.2.3 Implicit or explicit tagging

The following guidelines are appropriate when considering implicit tagging versus explicit tagging:

a) 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 any type). This can be easily specified by including the "IMPLICIT TAGS" keywords in each module definition.

Example:

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 that 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.

b) 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.

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

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 "ElementType" 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 an "ElementType" can be omitted. It is recommended that the protocol designer chooses between them according to the following rules:

The following guidelines are appropriate when considering optional elements:

a) When the "ElementType" corresponds to a genuine option and has a default value, the DEFAULT keyword shall be used

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

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

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

c) When the presence of the "ElementType" depends on the value of another "ElementType", the OPTIONAL keyword shall be used

```
DataUnit ::=            SEQUENCE{
        calledParty             Address,
        invokedSupplService     SS-Code,
        forwardedToNumber       Address OPTIONAL}
```

5.4 Subtyping

The following guidelines are appropriate when considering subtyping:

a) 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) 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.

```
Examples:

upperBound INTEGER ::= 20

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

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

c) 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.
```

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

```
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 "typereference" or "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.

c) There are situations where a "valuereference" should be exported even if it is not to be used as a "DefinedValue" (e.g. if is to be used for defining a particular message value by referring to the value of each of its components).

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".

```
Defines:

IMPORTS TypeA FROM Module-A;

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


rather than

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

5.6 Miscellaneous

5.6.1 Use of identifiers

The following guidelines are appropriate when considering the use of identifiers:

Although the use of identifiers is not mandatory from a syntactical point of view, 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 parsibility 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 ::= 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}
 }
```

Note that this is valid even if the "identifier" is obvious from the "typereference". The use of a "NamedType" without an "identifier" will be deprecated by the next version of ASN.1 specifications.

5.6.2 Instances 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 two instances of the same data type, or if required for decoding, as instances of two isomorphic types derived from the same base type by context-specific tagging. The base type shall be defined only in one place.

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
```

5.6.3 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 construction.

This may allow an optimized coding of constructed data values when other encoding rules than the Basic Encoding Rules (e.g. Packed Encoding Rules [15]).

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}
```

5.6.4 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, each component should be a "DefinedType", except in case of simple built-in types (e.g. a boolean type). This means that the protocol designer should avoid to expand the definition of the component type in the definition of the structured type where the component appears.

As an example, a protocol designer should rather define:

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

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

than define:

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

6 Specification of operations and errors

Material on specification of Operations and Errors is already available in Recommendation Q.775 (TCAP) [7] as well as CCITT Recommendation X.219 (ROSE) [9], CCITT Recommendation X.229 [10]. In both cases, the information given focuses on the use of the MACRO notation for defining Operations and Errors.

The following additional guidelines are appropriate when considering the specification of 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.

   Note that this method is only valid when using the current MACRO notation. The concept of type and values for Operations and Errors will be no longer relevant if the new ROS notation is endorsed by ETSI (Draft CCITT Recommendation X.9ros [16]).

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.

c) The allocated values shall be local values rather than global values. This has two advantages, it reduces the message length and avoid the registration process required for allocation of Object identifier values.

   It should be noticed that the scope of a local value is an abstract syntax. In practice, in a signalling environment, this currently means that local values shall be unique within the scope of a CCITT SS7 sub-system number, a DSS.1 service discriminator or an application-context.

   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 protocol.

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

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.

   One of the advantages of giving a name to the type of the argument or result parameter (rather than expanding this type definition) is for test specification where it is convenient (sometimes necessary (see ISO DIS 9646 [8])) to have names for the PDU of the tested layer/sub-layer.

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, ... have to be defined using separated assignment. They can be regarded as the ROSE-user or TCAP-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, ...).

```
Example of two operation types which differs 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
```

7 Specification of abstract syntaxes

7.1 General aspects

In most cases, explicit identification of abstract syntaxes is currently not used in signalling. For example, the abstract-syntax which represents the set of values each of which is a value of type TCAP Message (see CCITT Recommendations Q.771-Q.775 [7]) where the ANY DEFINED BY clauses are resolved by a particular list of Operation definitions is implicitly identified by a Sub-System Number or an application-context-name.

However the use of an external type (EXTERNAL) to convey user information requires at least that some other abstract syntaxes be named and clearly defined.

Currently, ASN.1 does not provide a formal way to specify an abstract syntax. However for this purpose it is recommended to create an ASN.1 module which includes a choice type built from all the data types which form the abstract syntax. A tagged type shall be derived from each of these types by context-specific tagging to ensure that the different data units can be distinguished.

The abstract syntax can then be informally defined by the following sentence to be included in the protocol specifications:

"The set of data values of type Module-X.Type-A form an abstract syntax which is identified by the following abstract-syntax-name: <objectIdentifierValue>"

Where Type-A is the name of the choice type and Module-X is the name of the module where it is defined.

The new ASN.1 specifications provide new features for Information Object Specifications [12]. A built-in information object class is defined to represent abstract syntaxes. Using the Information Object Specification notation, an abstract-syntax can be defined as follows:

```
Example-Abstract-Syntax ABSTRACT-SYNTAX ::=

{

TypeA IDENTIFIED BY {objectIdentifierValueX}

}
```

where TypeA is the type (generally a choice type) which encompasses all the data values contained in the abstract syntax "Example-Abstract-Syntax" uniquely identified by the object Identifier Value "ObjectIdentifierValueX".

7.2 Use of the external type

An external type enables a protocol designer to include free slots in a protocol data unit so that other protocol designers can "specialize" the protocol by adding specific information elements (e.g. in order to define a national variant for a standardised protocol).

A value of an external type includes both the value of a data type and a direct or indirect reference to the abstract syntax (and possibly to the transfer-syntax) to which this value belongs to.

The following guidelines are appropriate when considering the use of an external type:

a) Due to the absence of presentation layer in the current signalling environment, it is essential that only direct-references are used.

b) A protocol designer who wants to make use of an external type shall first define an abstract syntax (see subclause 7.1) which encompasses all the data values which may be carried by this external type. Then he shall assign a name (i.e. an object identifier value, according to CCITT Recommendation X.208 [1] registration rules) to this abstract syntax. This name serves as a direct-

reference when data values from this abstract-syntax are conveyed using an external type (the transfer-syntax is supposed to be agreed "a priori" between the peers).

Note that 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.

Example:

The following message specification allows a protocol designer to specialize the message structure so that it can carry operator specific information.

```
MessageA ::= SEQUENCE{
        calledParty     Address,
        callingParty    Address,
        service         ServiceCode OPTIONAL,
        specificInfo    EXTERNAL OPTIONAL}
```

The following module defines an abstract syntax whose protocol data units can be carried by the external type included in the previous message.

```
SpecificInfo DEFINITIONS ::=
BEGIN


-- This module defines an abstract syntax which
-- consists of data values of type
-- SpecificInfo.SupplServiceInfo.
-- This abstract syntax is identified by the
-- following name: <objectIdentiferValue1>.


SupplServiceInfo ::= CHOICE{
  callBarring       [0] CallBarringInfo,
  callForwarding    [1] CallForwardingInfo}

CallBarringInfo ::= --

CallForwardingInfo ::= --

END
```

When an instance of the type MessageA is used the value of the "direct-reference" component of the external type will be set to <objectIdentifierValue1>, while the value of the "single-ASN1-type" alternative[3)] of the "encoding" component will be a value of type SupplServiceInfo (i.e. either of type CallBarringInfo or type CallForwardingInfo).

---

3) Other encoding alternatives may be used by the implementation (see CCITT Recommendation X.209 [2]).

8 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.

8.1 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).

8.1.1 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 TCAP);
- remove a "NamedType" from the "AlternativeTypeList" of a Choice type;
- remove a "NamedNumber" 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 "ElementTypeList" of a sequence type;
- make any combination of the above changes to one or more components of a structured type.

8.1.2 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";

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 "typereference", a "modulereference" a "valuereference" or an "identifier" consistently throughout all ASN.1 Modules. This includes adding identifiers where allowed from the syntax;

NOTE: It includes also to remove identifiers where allowed from the syntax, however, this is not considered to be a good specification style and contradicts the recommendations made in the previous subclauses.

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 "Valueassignment" if their associated "valuereference" is never used (even not implicitly in ANY DEFINED BY construction) throughout all ASN.1 Modules;

o) Remove one or more existing "Typeassignment" (including those of Type OPERATION and ERROR) if they are never used throughout all ASN.1 Modules;

p) Add an ERROR Type or Value which was already included in the abstract-syntax (i.e. attached to another OPERATION type) to the list of ERRORS of an OPERATION Type if it had such a list or add a list of ERRORS to an OPERATION Type if it didn't have a list;

q) Add an OPERATION Type or Value which was already included in the abstract-syntax (e.g. not as a linked Operation) to the list of LINKED OPERATIONS of an OPERATION Type if it had such a list or add a list of LINKED OPERATIONS to an OPERATION Type if it didn't have a list.

8.1.3 Extension of an abstract syntax

An abstract-syntax is extended if its associated type which 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, an 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;

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

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 3: The Tag of this component has to remain unchanged and distinctness of Tags has to be ensured;

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

NOTE 4: 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 TCAP messages or ROSE PDUs parameterized by a specific list of operations:

m) Add new value definition of Type OPERATION and ERROR respectively as long as they are distinct with all other value definitions of Type OPERATION and ERROR respectively;

n) Add a Type as ARGUMENT/PARAMETER to an OPERATION Type if it didn't have an ARGUMENT/PARAMETER;

o) Add a Type as RESULT to an OPERATION Type if it had an empty RESULT or no RESULT;

p) Add a Type as PARAMETER to an ERROR Type if it didn't have a PARAMETER.

8.1.4 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 8.1.3;

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 construction which may be extended.

The type for such an information element can be either an external type or a sequence type including an ANY DEFINED BY construction. The new ASN.1 specification Object Specification and Parameterized type will provide other means for specifying PDUs which allow private extensions.

8.2 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 are used (see CCITT Recommendation X.209 [2]).

8.2.1 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 (PER) [15] are used.

8.2.2 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).

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:

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


changed to:

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

8.2.3 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.

9 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.

9.1 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 8.2.2 and 8.2.3.

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 8.2.1

9.2 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 subclause, 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.

Depending on the encoding rules and some forward compatibility rules, the behaviour of an implementation receiving a bit string representing an encoded PDU of a new version of the protocol is one of the following:

a) the entire bit string can be converted to an ASN.1 value satisfying the original PDU type definition (this situation occurs if the new protocol is an extension of the original one and if the new elements are not used for this instance of the PDU;

b) the bit string can be recognized as representing an actual value which deviates from some known value by either or both of:

   1) relaxation of one or more constraints; or

   2) the application of extensibility rules

c) it is not possible to identify any ASN.1 value corresponding to the bit string (i.e. the entire message cannot be decoded).

If no forward compatibility rules are specified, it should be assumed that in cases b) and c), the default error handling behaviour should take place: discard the entire PDU and optionally return an appropriate error indication (e.g. a reject component in case of TCAP or ROSE based protocols).

Forward compatibility rules can be introduced to specify that in case b) (i.e. when the presence of unknown values in an encoded PDU does not render impossible the decoding of this PDU), an implementation shall consider valid the known value which has been recognized.

If the Basic Encoding Rules are in use, the following guidelines are appropriate when forward compatibility is required:

a) The protocol designer of the original protocol should specify extensibility rules so that a receiving protocol machine accepts a value which deviates from a known value because component values have been added in one or more sequence-like value at any level of the PDU structure.

b) In order to make easier the transition to the new ASN.1 notation, this specification shall be made via comments placed in regard of each sequence type in which unknown values can be ignored.

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

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

TypeA ::= SEQUENCE{
        element3 TypeC,
        element4 TypeD} -- added elements allowed

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

New ASN.1 specifications will provide a formal way to specify these rules:

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

TypeA ::= SEQUENCE{
        element3 TypeC,
        element4 TypeD} (!ERROR {errorCode})

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

where errorCode refers to a specific behaviour expected form the receiving protocol machine in case of abnormal situation (e.g. ignore extra information and process recognized information as far as possible).

10 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).

10.1 Modules 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.

10.2 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 (e.g. because one of these types is extended).

10.3 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.

Annex A: List of ASN.1 tools

This list is not intended to be exhaustive and is provided for information only. There have been no attempts to evaluate these tools. They are listed in alphabetical order.

ASTOOL/C

KDD R&D bldg 2-1-23 Nakameguro
Meguroku
Tokyo 153
JAPAN
Contact: Shingo Nomura
Tel: + 81 3 3794 84 01
Fax: + 81 3 5704 20 76
Internet: nomura@kddlab.kddlabs.co.jp
Target Languages: C
Environment: Unix (Sparc, DEC), VAX (VMS), PC (DOS)

CASN92

NOKIA Research Centre
P.O. Box 156
SF-02101 Espoo
Finland
Contact: Ari Ahtiainen
Tel: + 358 0 43 761
Fax: + 358 0 43 76 227
Internet: aanen@rc.nokia.fi
Target Languages: C
Environment: MS-DOS, Unix (Sun-OS, SCO, ...)

DSET ASN-C

DSET coorporation
Lebanon
New Jersey
USA
Contact: Ju Zhang
Tel: + 1 908 832 65 33
Internet: zhang@dset.uucp
Target Languages: C, C++
Environment: Unix (SUN and PC)

ELLEMTEL

ELLEMTEL-ERICSSON LABs
PO BOX 1505
S-125 25 ALVSJO
SWEDEN
Tel: + 46 8 727 39 83
Fax: + 46 8 647 82 76
Internet: klacke@erix.ericsson.se
Target Language: Erlang

ISODE (PEPY, POSY)

PSI Inc.
PSI California Office
420 Whisman Court
Mountain view
CA 94043-2112
USA
Tel: + 1 415 961 3380
Internet: isode-request@nic.ddn.mil
Contact: Marshall T. Rose
Public domain can be FTP'ed from various sites
(e.g. 192.33.4.10)
Environment: Unix (SUN OS, HP-UX, ...)
Target Languages: C

KVATRO

KVATRO A/S
Pirsenteret
N-7005 Trondheim
NORWAY
Contact: Bernt Marius Johnsen
Tel: + 47 7 52 00 90
Fax: + 47 7 52 01 40
Internet: bernt@kvatro.no
Target Languages: CHILL
Environment: Unix (Sun OS) VAX/VMS

MAVROS

INRIA
BP 105
Rocquencourt
78153 Le Chesnay Cedex
FRANCE
Contact: C. Huitema
Internet: huitema@mitsou.inria.fr
Environment: Unix (Sun, HP, Dec, Vax)
Target Languages: C

OSS

Open Systems Solutions Inc.
301 North Harrison Street, Suite 265
Princeton
New Jersey 08540
USA
Contact: Bancroft Scott
Tel:+ 1 609 987 9073
Internet: baos@caip.rutgers.edu
Environment: Unix (HP-UX,Beykerley, AT&T system V, Sun OS, SCO Unix),DEC Ultrix, VAX-VMS, OS/2, DOS/Windows, MacIntosh, Tandem S/2, Tandem Guardian, MVS
Target Languages: C , C++, (ADA)

PLC409

MARBEN
38, rue Copernic
75116 Paris
France
Contact: E.Materra
Tel: + 33 1 45 00 82 06
Fax: + 33 1 45 01 23 31
Environment: Unix
Target Languages: C, Pascal and Cobol

RETIX

2644 30th Street
Santa Monica
California 90405 3009
USA
Tel: + 1 213 399 2200
Fax: + 1 213 458 2685
Target Languages: C
Environment: MS-DOS, Unix , Ultrix

SIEMENS ASN.1 TOOL SET

SIEMENS AUSTRIA
Siemenstrasse 92
AUSTRIA, 1210 Vienna
Contact: A.Karner
Tel: + 43 222 25 01 43 31
Fax: + 43 222 25 01 46 00
Environment: DOS, Unix
Target Language: No (interpretor)

History

Document history	
November 1992	First Edition
April 1996	Converted into Adobe Acrobat Portable Document Format (PDF)