TSG-RAN Working Group 2 (Radio layer 2 and Radio layer 3) ***TSGR2#2(99)087***
Stockholm 8th to 11th March 1999

Agenda Item: 9.0 Specification methodology

Source: Nokia

Title: Change request for R2.01: Use of ASN.1 for definition of abstract syntaxes of protocol messages. Specification of default and special encoding.

Document for: Decision

OVERVIEW

This document contains proposals for the R2.01, Guidelines and Principles for protocol description and error handling [7].

This document provides proposals for the following items specified in [7]:

1. use subset of ASN.1 (compatible with Z.105) for definition of abstract syntax of protocol messages.
2. there is a need for a default encoding, which can be applied in most cases
3. there is a need for a special encoding e.g. by means of CSN.1

The clause 10.~~10.~~ describes the subset of ASN.1. The clause 11.~~11.~~ contains rules for default encoding and specification of specialised encoding.

Proposal:

Add the clauses 10 and 11 of this document to R2.01.

REFERENCES

[1] X.680, Abstract Syntax Notation One (ASN.1): Specification of the basic notation.

[2] X.681, Abstract Syntax Notation One (ASN.1): Information Object Specification.

[3] X.682, Abstract Syntax Notation One (ASN.1): Constraint Specification.

[4] X.690, ASN.1 Encoding rules: Specification of basic encoding rules (BER), canonical encoding rules (CER) and distinguished encoding rules (DER)

[5] X.691, ASN.1 Encoding Rules - Specification of packed encoding rules (PER)

[6] CSN.1 specification, version 2.0.

[7] 3GPP TSG RAN WG2 R2.01, Guidelines and Principles for protocol description and error handling.

10. ~~MESSAGE ABSTRACT SYNTAX SPECIFICATION~~USAGE OF ASN.1

~~The following clauses specify how abstract syntaxes for protocol message are specified using ASN.1.~~The following clauses contain guidelines for specification of protocol messages with ASN.1.The purpose of ASN.1 is to make it possible to specify abstract syntax of a message (i.e. what is the contents of a message) separately from its transfer syntax (i.e. how a message is encoded for transmission). The features that ASN.1 provides include specification of:

- Extensibility (both structural and extension of value set)
- Optional IEs and values (see the clauses 10.2.2.~~10.2.2.~~ and 10.3.10.~~10.3.10.~~)
- Default values (see the clauses 10.2.2.~~10.2.2.~~ and 10.3.10.~~10.3.10.~~)
- Comprehension required (see the clause 10.2.4.~~10.2.4.~~)
- Inter/Intra IE dependency (see the clause 10.3.10.~~10.3.10.~~)
- Specification of partial decoding (see the clause 10.2.5.~~10.2.5.~~)

The clause 11 specifies how message transfer syntax is specified. It should be noted that importance of some transfer syntax properties must be determined early during specification because of their effect on message abstract syntax specification possibilities. The properties are **compactness** and **extensibility.** If extreme compactness is required then extensibility must be restricted. If good extensibility is required then compromises must be done regarding compactness. The sections concerning these issues are marked in the following clauses as **COMPACTNESS** and **EXTENSIBILITY**.

10.1. Message level

10.1.1. Messages

It is presumed that messages share the same structure, namely that they contain an identification part and a contents part. An identification part contains an IE that identifies a message among all messages in some context. A contents part contains message specific IEs.

Example: A protocol layer XYZ contains three messages: A, B and C. The structure of the messages is as presented in the figure 3-1.

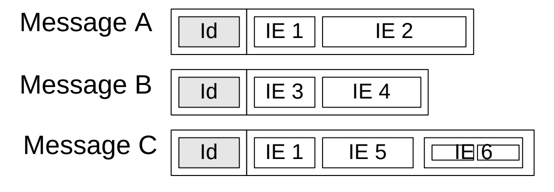

Figure 3-1: Three example messages

Messages are specified using ASN.1 [1]. There are three ASN.1 types, *MessageA*, *MessageB* and *MessageC*, which contain definitions for the contents of the above messages. The mapping between the message contents types and message identifiers is as follows:

Message id	Type of message contents
1	MessageA
2	MessageB
3	MessageC

New message types will be introduced in the future.

In cases where different PDUs have different identification schemes it is possible to apply this categorisation for a set of PDUs that share the same identification scheme.

10.1.2. Message definition

In order to capture information in the previous clause the following three things must be defined:

1. A structure for the table
2. The table itself
3. A generic message structure which can contain both message identifier IE and message contents IEs (i.e. id 1 + *MessageA*, id 2 + *MessageB*, id 3 + *MessageC*)

The table structure is defined as follows using ASN.1 classes [2]:

```
XYZ-MESSAGE ::= CLASS {
   &id      MessageId,
   &Type
}
WITH SYNTAX {
   &id &Type
}

MessageId ::= INTEGER (0..63)
```

The table is defined as follows:

```
XYZ-Messages XYZ-MESSAGE ::= {
   { messageA-id MessageA } |
   { messageB-id MessageB } |
   { messageC-id MessageC } |
   ...                    -- Extension marker => additional messages
                          -- can be introduced.
}

messageA-id MessageId ::= 1
messageB-id MessageId ::= 2
messageC-id MessageId ::= 3
```

The following type represents the generic message structure that can carry values of the messages specified in the *XYZ-Messages* table.

```
XYZ-Message ::= SEQUENCE {
    id          XYZ-MESSAGE.&id    ({XYZ-Messages)},
                -- MessageId: 1, 2 or 3

    contents    XYZ-MESSAGE.&Type ({XYZ-Messages}{@id})
                -- id=1 => MessageA, id=2 => MessageB, id=3 => MessageC
}
```

The above definition means that if *id* is 1 then the *Message* type is equivalent to the following type:

```
XYZ-Message ::= SEQUENCE {
    id          MessageId,    -- 1
    contents    SEQUENCE {
         ie1            IE1,
         ie2            IE2
    }
}
```

If *id* is 2 then the type is equivalent to the following type:

```
XYZ-Message ::= SEQUENCE {
    id          MessageId,    -- 2
    contents    SEQUENCE {
         ie3            IE3,
         ie4            IE4
    }
}
```

10.1.3. Messages and ASN.1 modules

ASN.1 definitions shall be placed in ASN.1 modules such that definitions in a module form a logical unit. For example PDUs definitions for one protocol layer could be in one ASN.1 module and IE definitions in another.

The tagging mode for the modules shall be "AUTOMATIC TAGS".

Example: A message definition module for the XYZ protocol layer.

```
XYZ-Messages DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

XYZ-Messages XYZ-MESSAGE ::= {
    { messageA-id MessageA } |
    { messageB-id MessageB } |
    { messageC-id MessageC } |
    ...                     -- Additional messages can be introduced.
}

MessageA ::= SEQUENCE {
    -- Message contents
}

messageA-id MessageId ::= 1

MessageB ::= SEQUENCE {
    -- Message contents
}

messageB-id MessageId ::= 2

MessageC ::= SEQUENCE {
    -- Message contents
}

messageC-id MessageId ::= 3

END
```

10.1.4. Messages and SDL

The identifiers *messageA-id*, *MessageA*, *messageB-id*, etc. can be used in descriptive SDL when protocol behaviour is specified. Note that classes and objects can not (yet) be referenced in SDL. Types and values however can be imported to SDL definitions. The figures below contain some examples about usage of ASN.1 in SDL specifications.

```
imports
  MessageA,  messageA_id,
  MessageId
from SomeASN1Module;

signal XYZ_MessageA(
  MessageId, MessageA);

dcl aVariable MessageA;
```

Figure 3-2: Import and use of ASN.1 definitions in SDL.

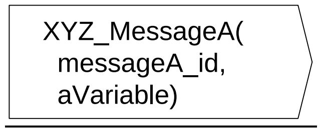

Figure 3-3: Sending of a message id and contents.

10.2. Information element level

Messages consist of information elements.

The following ASN.1 message types are used in the following clauses.

```
MessageA ::= SEQUENCE {
   ie1  IE1,              -- A mandatory IE.

   ie2  IE2  OPTIONAL,    -- An optional IE.

   ...                    -- An extension marker.
}

MessageB ::= SEQUENCE {
   ie3  IE3
        (CONSTRAINED BY {-- ComprehensionRequired(is for receiver) --}
        !comprehensionRequiredFailure),

   ie4  IE4  DEFAULT 0,   -- An optional IE with a default value.

   ...
}

MessageC ::= SEQUENCE {
   ie1  IE1
        (CONSTRAINED BY {-- PartialDecoding(OnErrorIgnoreRest) --}
        !partialDecodingFailure)
        OPTIONAL,

   ie5  IE5
        (CONSTRAINED BY {-- PartialDecoding(OnErrorIgnoreRest) --}
        !partialDecodingFailure)
        OPTIONAL,

   ... ,                  -- An extension marker

   ie6  IE6
        (CONSTRAINED BY {-- PartialDecoding(OnErrorIgnoreRest) --}
        !partialDecodingFailure)
        OPTIONAL          -- A new IE
}

-- Error codes
comprehensionRequiredFailure INTEGER ::= 1
partialDecodingFailure       INTEGER ::= 2
```

10.2.1. Message contents

A message contents structure is defined using a sequence type (10.3.10.~~10.3.10.~~).

Example: *MessageA*, *MessageB* and *MessageC* are message contents structures.

10.2.2. Optional IEs and default values

An IE can be marked as optional.

COMPACTNESS: Optional IEs shall be after mandatory ones.

Example: *MessageA.ie2* is an optional IE.

```
  ie2    IE2  OPTIONAL
```

An IE can be marked as being optional and having a default value. In those cases a missing optional IE may be understood as having a certain value hence a defined meaning.

Example: *MessageB.ie4* is an optional IE with a default value.

```
    ie4  IE4  DEFAULT 0
```

10.2.3. New IEs

EXTENSIBILITY: If new IEs will be added to a message then the message contents structure must be specified as extensible using the ellipsis notation (...). New IEs shall be added after the extension marker. New IEs shall be optional or shall have default values.

Example: *MessageC.ie6* is an additional optional IE.

```
    ... ,
    ie6  IE6  OPTIONAL
```

10.2.4. Comprehension required

"Comprehension required" requirement can be associated with an IE. It means that after an IE value has been decoded then the value is validated. Failure in validation causes rejection of the message.

The requirement is specified as an extension to ASN.1 by using user defined constraints [3]. The comment part of the constraint shall be of the form:

ComprehensionRequired(<additional constraint>)

where <additional constraint> specifies the rule that the IE must satisfy.

Example: The *MessageB* is a broadcast message. The *ie3* IE contains recipient addresses. It is not until the addresses have been decoded when a receiver can decide whether it should decode the rest of the message or not.

```
    ie3  IE3
         (CONSTRAINED BY {-- ComprehensionRequired(is for receiver) --}
         !comprehensionRequiredFailure),
```

10.2.5. Partial decoding

"Partial decoding" means that a PDU can be decoded in parts. One part forms a complete value that can be separated from other parts. A decoding error in a part does not invalidate previously decoded parts. Subsequent parts are however invalidated.

"Partial decoding" is specified as an extension to ASN.1 using user defined constraints. The comment of constraint shall be of the form:

PartialDecoding(<OnErrorClause>)

where <OnErrorClause> specifies action in case of a decoding error. The possible alternatives are:

- OnErrorIgnoreRest: End decoding, ignore rest of the message

Example: The *MessageC* is a multipurpose message. The IEs *ie1*, *ie5* and *ie6* are independent of each other.

```
   ie1  IE1
        (CONSTRAINED BY {-- PartialDecoding(OnErrorIgnoreRest) --}
        !partialDecodingFailure)
```

10.2.6. Error specification

An error specification can be associated with user defined constraints.

A simple integer value can be associated with an exception specification or as elaborate structured value as needed.

Example: If decoding of *ie1* fails then decoder returns the error code *partialDecodingFailure*.

```
   ie1  IE1
        (CONSTRAINED BY {-- PartialDecoding(OnErrorIgnoreRest) --}
        !partialDecodingFailure)
```

10.3. Value level

Information elements consist of values.

If the CSN.1 specified default syntax (see the clause 11.~~11.~~) is used as a transfer syntax then only the following ASN.1 types can be used in the value level:

- Boolean (10.3.4.~~10.3.4.~~)
- Integer (10.3.5.~~10.3.5.~~)
- Enumerated (10.3.6.~~10.3.6.~~)
- Bit string (10.3.7.~~10.3.7.~~)
- Octet string (10.3.8.~~10.3.8.~~)
- Null (10.3.9.~~10.3.9.~~)
- Sequence (10.3.10.~~10.3.10.~~)
- Sequence-of (10.3.11.~~10.3.11.~~)
- Choice (10.3.12.~~10.3.12.~~)
- Character string types (10.3.13.~~10.3.13.~~)

Otherwise there are no restrictions on usage of ASN.1 types.

10.3.1. Extensibility

COMPACTNESS: In the value level use of ASN.1 extensibility is forbidden unless otherwise stated in the following clauses.

10.3.2. Comprehension required

"Comprehension required" can be applied to components of sequence types, alternatives of choice types and elements of sequence-of types. See 10.2.4.~~10.2.4.~~

10.3.3. Partial decoding

"Partial decoding" can be applied to components of sequence types, alternatives of choice types and elements of sequence-of types. See 10.2.5.~~10.2.5.~~

10.3.4. Boolean

Example: A simple boolean type.

```
Flag ::= BOOLEAN

setFlag Flag ::= TRUE
```

10.3.5. Integer

An integer type should be constrained.

COMPACTNESS: An integer type shall be constrained to have a finite value set. The value set can be either continuous or non-continuous.

Named numbers can be associated with an integer type.

COMPACTNESS, EXTENSIBILITY: If an integer type needs to be extended in the future then two value sets must be defined:

- A value set that specifies the values that can be sent in the current protocol version.
- A value set that specifies all the possible values that can be received now and in the future.

The former value set is specified in a user-defined constraint. The comment part shall be of the form:

Send(<value set>)

The latter form is specified using a normal constraint, e.g. a value range constraint.

Examples: Integer types and values.

```
Counter        ::= INTEGER (0..255)         -- 0 <= Counter value <= 255

SparseValueSet ::= INTEGER (0|3|5|6|8|11)

SignedInteger  ::= INTEGER (-10..10)

-- idle stands for value 0.
Status         ::= INTEGER { idle(0), veryBusy(3) } (0..3)

-- Send values 0..3 but be prepared to receive values 0..15.
Extensible     ::= INTEGER (0..15)(CONSTRAINED BY {-- Send(0..3) --})

initialCounter Counter        ::= 0

zero           SparseValueSet ::= 0

initialStatus  Status         ::= idle
```

10.3.6. Enumerated

An enumerated type shall have a continuous finite value set. The enumeration value of the smallest enumeration shall be 0. The list of enumerated values specifies the value set for an enumerated type.

COMPACTNESS, EXTENSIBILITY: If an enumerated type needs to be extended in the future then two value sets must be defined as in case of integer types.

Note: An integer type with named numbers can be used as on alternative to an enumerated type.

Example: Enumerated types and value.

```
Enum         ::= ENUMERATED { a, b, c, d }

-- Send values a, b, c or d but be prepared to receive values
-- a, b, c, d, spare4, spare5, spare6 and spare7.
ExtendedEnum ::= ENUMERATED { a, b, c, d, spare4, spare5, spare6, spare7 }
   (CONSTRAINED BY {-- Send(a|b|c|d) --})

aEnum Enum ::= a
```

10.3.7. Bit string

A size constraint shall be specified. It shall be finite.

Named bits can be associated with a bit string type.

Example: Bit string types and values.

```
FixedLengthBitStr    ::= BIT STRING (SIZE (10))

VariableLengthBitStr ::= BIT STRING (SIZE (0..10))

BitFlags             ::= BIT STRING { a(0), b(1), c(2), d(3)} (SIZE (4))

fix FixedLengthBitStr    ::= '0001101100'B

var VariableLengthBitStr ::= '0'B

flg BitFlags             ::= { a, c, d } -- '1011'B
```

10.3.8. Octet string

A size constraint shall be specified. It shall be finite.

Example: Octet string types and values.

```
FixedLengthOctetStr    ::= OCTET STRING (SIZE (10))

VariableLengthOctetStr ::= OCTET STRING (SIZE (0..10))

UpperLayerPDUSegment   ::= OCTET STRING (SIZE (1..512))

fix FixedLengthOctetStr    ::= '0102030405060708090A'H

var VariableLengthOctetStr ::= 'FF'H
```

10.3.9. Null

A null type has only one value, NULL.

Example: Null type as an alternative type of a choice type.

```
IE ::= CHOICE {
   doThis     ThisArg,
   doThat     ThatArg,
   doNothing  NULL
}
```

10.3.10. Sequence

A sequence type is a record. Components of a sequence type can be optional or they can have default values. Optional components and components with default values should be after mandatory components.

Inner subtyping can be used to force an optional component to be present or absent in a derived type.

If an optional component is conditionally present or absent then the condition shall be specified in a user defined constraint of the form:

Condition(<condition expression>)

<condition expression> shall be such that both sender and receiver are able to evaluate it before a conditional component is encoded or decoded.

"Comprehension required" can be associated with a component of a sequence type.

"Partial decoding" can be associated with a component of a sequence type.

EXTENSIBILITY: A sequence type can be marked as extensible.

Example: Sequence types and values.

```
Record ::= SEQUENCE {
   flag     Flag,
   counter  Counter,
   bitFlags BitFlags     OPTIONAL,
   extEnum  ExtendedEnum DEFAULT a
}

DerivedRecord ::= Record (WITH COMPONENTS {
                      bitFlags  PRESENT
                  })

RecordWithConditionalComponent ::= SEQUENCE {
   mand     INTEGER (0..7),
   opt      BOOLEAN OPTIONAL,
   cond     BOOLEAN
            (CONSTRAINED BY {--Condition(field 'mand' is 7)--})
            OPTIONAL
}

aRecord Record ::= {
   flag     TRUE,
   counter  100
}

anotherRecord DerivedRecord ::= {
   flag     TRUE,
   counter  1000,
   bitFlags '0101'B      -- bitFlags must be present
}
```

10.3.11. Sequence-of

A sequence-of type is a list of some element type. A size constraint shall be specified. It shall be finite.

"Comprehension required" can be associated with an element of a sequence-of type.

"Partial decoding" can be associated with an element of a sequence-of type.

Example: Sequence-of types and values.

```
FixedLengthList        ::= SEQUENCE (SIZE (10)) OF Record

VariableLengthList     ::= SEQUENCE (SIZE (0..10)) OF Status

UpperLayerPDUSegments ::= SEQEUNCE (SIZE (1..10)) OF UpperLayerPDUSegment

aList VariableLengthList ::= { idle, 1, 2, veryBusy, 2, 1, idle }
```

10.3.12. Choice

A choice type is a variant record. Only one alternative component can be selected.

Inner subtyping can be used to force an alternative to be selected in a derived type.

"Comprehension required" can be associated with an alternative component of a choice type.

"Partial decoding" can be associated with an alternative component of a choice type.

EXTENSIBILITY: A choice type can be marked as extensible.

Example: Choice type and value.

```
VariantRecord ::= CHOICE {
   flag      Flag,
   counter   Counter,
   extEnum   ExtendedEnum
}

aVariantRecord variantRecord ::= flag : FALSE
```

10.3.13. Restricted character string types

A size constraint shall be specified. It shall be finite.

Example: Character string types.

```
FixedStr  ::= IA5String (SIZE (10))

VarStr    ::= IA5String (SIZE (1..10))

FixedWStr ::= BMPString (SIZE (10))

VarWStr   ::= BMPString (SIZE (1..10))
```

10.3.14 IEs and ASN.1 modules

If an IE or a value field within an IE is a parameter from another protocol layer then type for such a field should be defined in another module. In this way there is a clear separation of definitions that are specific to different protocol layers.

Example: The XYZ protocol message *MessageC* contains an IE, which contains an OPQ protocol layer specific field *parameter1*. Type for the field is imported from OPQ specific module.

```
XYZ-Messages DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

IMPORTS
   OPQParameter       -- OPQParameter is not defined within XYZ-Messages
                      -- module.
FROM OPQ-DataTypes;

MessageC ::= SEQUENCE {
   -- Other IEs.
   ie6  IE6 OPTIONAL
}
-- Other definitions ...

IE6 ::= SEQUENCE {
   parameter1         OPQParameter,   -- Imported definitions can be
                                      -- referred to.
   parameter2         XYZParameter
}

XYZParameter ::= INTEGER (0..255)

END
```

Example: The OPQ protocol layer specific module exports *OPQParameter* type so that other modules can refer it.

```
OPQ-DataTypes DEFINITIONS AUTOMATIC TAGS ::=

BEGIN

EXPORTS
   OPQParameter
;

OPQParameter ::= INTEGER (0..7)

END
```

11. MESSAGE TRANSFER SYNTAX SPECIFICATION

11.1. Default encodingSelection of transfer syntax specification method

11.1.1 Transfer syntax specification method alternatives

There are three the following alternatives for specification of default encodingmessage transfer syntax. One transfer syntax specification method shall be selected for all the messages of a given protocol.

- BER (Basic Encoding Rules, X.690) [4] (including CER and DER)
- PER (Packed Encoding Rules, X691) [5]
- CSN.1 specified encoding, see the following clauses.
- Tabular format

BER and PER are standard ASN.1 encoding rules. **BER** produces large octet oriented encodings.

CSN.1 is not standardised but it is publicly available. Use of CSN.1 with ASN.1 is presented in the clause 11.2.

There are no formal rules for specification of tabular format transfer syntax.

11.1.2 Comparison of methods

The following table contains comparison of transfer syntax specification methods. The numbers indicate the rank of a method.

Criteria	**BER**	**PER**	**CSN.1**	**Tabular format**
Compactness	4	2	1	3
Extensibility	1	1	2 *	2 *

BER produces large octet oriented encodings with a lot of extra control information. For radio protocol messages encodings are too large. Thus BER should not be used.

PER produces small bit oriented encodings. BASIC UNALIGNED PER produces the most compact encodings whereas BASIC OCTET-ALIGNED PER pads some fields. PER provides good support for extensibility. The support causes some growth of messages. PER produced encodings are self-delimiting.

CSN.1 produces smallest encodings. If the CSN.1 alternative is selected then the **COMPACTNESS** and **COMPACTNESS, EXTENSIBILITY** sections in the clause 10.10. shall be followed. The following clauses do not support the **EXTENSIBILITY** sections. Such support is for FFS. If message level extensibility is specified then CSN.1 produced encodings are not self-delimiting because no length information is encoded for extended fields. If message level extensibility is not used then CSN.1 produces self-delimiting encodings.

PER and **CSN.1** are best suited for cases when the structure of a message is complex, e.g. there are many IEs/value fields, some optional or alternative or repetitive etc.

* The relative order of these two methods can not be definitely stated because they do not provide one fixed way for specification of extensibility.

PER and CSN.1 produce similar encodings. Selection between PER and CSN.1 should be done according to the following criteria:

PER and CSN.1 produce small bit oriented encodings.Selection of encoding rules for default encoding depends of the following two factors:

- Extensibility
- Compactness of encoding
- If **compactness** of encoding is the most important requirement and the restricted extensibility is adequate (message extensions are always added to end of a message as new IEs) then the **CSN.1** alternative should be selected.
- If **extensibility** is the most important requirement and compactness is the second then **BASIC UNALIGNED PER** (or BASIC OCTET ALIGNED PER) should be selected.

If new **extensions** should be able to be inserted in middle of a message and **compactness** is important then **BASIC UNALIGNED PER** should be selected.

If new **extensions** should be able to be inserted in middle of a message and compactness is not so important then **BASIC OCTET-ALIGNED PER** (or BER) should be selected

In case of **tabular** format properties of encoding depend on how a message is specified. This is because there are no formal rules for specification of tabular format transfer syntax. Tabular format is best suited for cases when there are few IEs/value fields and the structure of a message is simple.

11.1.1 Other issues

If there is definite size limit for a message (e.g. a broadcast message must fit into one lower layer message) then the COMPACTNESS sections in the clause 10.10. must be followed.[1]

11.2. CSN.1 encoding for ASN.1 types

The following clauses specify the CSN.1 [6] specific default encoding for ASN.1 types. The rules specify one-to-one mapping from an abstract syntax to a transfer syntax.[2]

11.2.1. Message structures

Message structures shall be encoded as follows:

---

[1] Note that it is the responsibility of a specifier to make sure that an abstract syntax produces a transfer syntax with wanted properties. Additional *user defined constraint* specifications should be considered.

[2] Note that it is possible to specify one set of of bits in multiple ways in CSN.1. For example the following descriptions denote the same set of bits:

{000|001|010|011}

0 {0|1} {0|1}}

0 bit(2)

This document contains a mapping from an ASN.1 type to an CSN.1 description. Other CSN.1 descriptions that denote the same bit set as presented in the document are also valid.

- A choice of all the messages specified in a message table.
- Selection is done according to the identifier field values.

Example: Encoding of the *XYZ-Message* type in 10.1.2.~~10.1.2.~~

```
<XYZ_Message> ::=
  { <id : 000001> <MessageA>
  | <id : 000002> <MessageB>
  | <id : 000003> <MessageC>
  }
;
```

11.2.2. Boolean

A boolean type maps to one bit.

Example: Encoding for the boolean type in 10.3.4.~~10.3.4.~~

```
<Flag> ::=
  <BOOLEAN>
;

<BOOLEAN> ::=
  bit
;
```

11.2.3. Integer

An integer type is encoded as an UNALIGNED variant of a constrained whole number as specified in PER [5].

Explanation: Let "lb" be lower bound and "ub" be the upper bound of an integer type. A value "n" will be encoded as a value e = ("n" - "lb") using the minimum number of bits necessary to represent the values in range.

Named numbers do not affect encoding.

If an integer type is marked as extensible as specified in 10.3.5.~~10.3.5.~~ then the reception and emission value sets are defined separately.

Example: Encodings for integer types in 10.3.5.~~10.3.5.~~

```
<Counter> ::=
  <INTEGER(8)>        -- n = 0..255, e = n-0 = 0..255
;

<SparseValueSet> ::=
  <INTEGER(4)>        -- n = 0|3|5|6|8|11, e = n-0 = 0|3|5|6|8|11
  exclude {
    0001|0010|0100|0111|1001|1010|1100|1101|1111
  }
;

<SignedInteger> ::=   -- n = -10..10, e = n-(-10) = 0..20
  <INTEGER(5)>
  exclude {
    10101|10110|10111|11000|11001|11010|11011|11100|11101|11110|11111
  }
;
```

```
<Status> ::=           -- n = 0..3, e = n-0 = 0..3
  <INTEGER(2)>
;

<Extensible> ::=       -- n = 0..3, e = n-0 = 0..3, two spare bits
  <INTEGER(4)> = 00 <INTEGER(2)>
;

<INTEGER(nBits)> ::=
  bit(nBits)
;
```

See also 11.3.~~11.3.~~ for specialised encoding.

11.2.4. Enumerated

Enumeration values form a value set of 0..(number of enumerations-1). Each enumeration item is encoded as its corresponding numeric value. A value "n" will be encoded using the minimum number of bits necessary to represent all the values in a value set.

If an enumerated type is marked as extensible as specified in 10.3.6.~~10.3.6.~~ then the reception and emission value sets are defined separately.

Example: Encodings for enumerated types in 10.3.6.~~10.3.6.~~

```
<Enum> ::=
  { <a : 00>
  | <b : 01>
  | <c : 10>
  | <d : 11>
  }
;

<ExtendedEnum> ::=
  { <a : 000>
  | <b : 001>
  | <c : 010>
  | <d : 011>
  | <spare4 : 100>
  | <spare5 : 101>
  | <spare6 : 110>
  | <spare7 : 111>
  } =
  { <a : 000>
  | <b : 001>
  | <c : 010>
  | <d : 011>
  }
;
```

11.2.5. Bit string

A bit string is mapped to a string of bits. If the number of bits may vary then a length field precedes the bit string.

A length field is encoded as an integer field of type INTEGER (lb..ub) where "lb" is the lower bound the of size constraint and "ub" is the upper bound.

Named bits do not affect encoding.

Example: Encodings for bit string types in 10.3.7.~~10.3.7.~~

```
<FixedLengthBitStr> ::=
  bit(10)
;

-- length = 0..10, e = length-0 = 0..10
<VariableLengthBitStr> ::=
  <length : <INTEGER(4)> exclude {1011|1100|1101|1110|1111}>
  bit * val(length)
;

<BitFlags> ::=
  bit(4)
;
```

11.2.6. Octet string

An octet string is mapped to a string of bits. If the number of octets may vary then a length field precedes the octet string.

A length field is encoded as an integer field of type INTEGER (lb..ub) where "lb" is the lower bound the of size constraint and "ub" is the upper bound.

Example: Encodings for octet string types in 10.3.8.~~10.3.8.~~

```
<FixedLengthOctetStr> ::=
  <octet>(10)
;

-- length = 0..10, e = length-0 = 0..10
<VariableLengthBitStr> ::=
  <length : <INTEGER(4)> exclude {1011|1100|1101|1110|1111}>
  <octet> * val(length)
;

-- length = 1..512, e = length-1 = 0..511
<UpperLayerPDUSegment> ::=
  <length : <INTEGER(9)>>
  <octet> * val(length)
;
```

11.2.7. Null

The null type is mapped to an empty bit string.

Example: Encoding for the choice type with nested null type in 10.3.9.~~10.3.9.~~

```
<IE> ::=
  { 00 <doThis : <ThisArg>>
  | 01 <doThat : <ThatArg>>
  | 10 <doNothing : null>
  }
;
```

11.2.8. Sequence

Component values are encoded using rules for component types.

Presence or absence of an **optional component** or a **component with a default value** is indicated with a heading bit. Default values do not affect encoding.

If an optional component is forced to be present or absent in a derived type then the heading bit is omitted for the derived type.

If an optional component is conditionally present or absent then the heading bit is omitted. Presence of a conditional component depends on the associated condition expression.

If a sequence type is a message contents type then

- If there is an extension marker then spare bits description follows the last component description.
- Truncation of omitted trailing optional components is allowed.

Example: Encodings for sequence types in 10.3.10.10.3.10.

```
<MessageA> ::=
  <ie1 : <IE1>>
  {
    { 0
    | 1 <ie2 : <IE2>>
    }

    <spare bit>(*)
  } // -- Truncation of optional components is allowed
;

<MessageB> ::=
  <ie3 : <IE3>>
    -- ComprehensionRequired(is for receiver)
    -- !comprehensionRequiredFailure

  {
    { 0 -- DEFAULT 0
    | 1 <ie4 : <IE4>>
    }

    <spare bit>(*)
  } // -- Truncation of optional components is allowed
;

<MessageC> ::=
  {
    { 0
    | 1 { <ie1 : <IE1>> ! <PartialDecodingFailure : bit(*) = <no string>>}
    }

    { 0
    | 1 { <ie5 : <IE5>> ! <PartialDecodingFailure : bit(*) = <no string>>}
    }

    { 0
    | 1 { <ie6 : <IE6>> ! <PartialDecodingFailure : bit(*) = <no string>>}
    }

    <spare bit>(*)
  } // -- Truncation of optional components is allowed
;
```

```
<Record> ::=
  <flag : <Flag>>
  <counter : <Counter>>
  { 0
  | 1 <bitFlags : <BitFlags>>
  }
  { 0
  | 1 <extEnum : <ExtendedEnum>>
  }
;

<DerivedRecord> ::=
  <flag : <Flag>>
  <counter : <Counter>>
  <bitFlags : <BitFlags>  -- Note: no heading bit as in <Record>
  { 0
  | 1 <extEnum : <ExtendedEnum>>
  }
;

<RecordWithConditionalComponent> ::=
  <mand : <INTEGER(3)>>   -- 0..7
  { 0
  | 1 <opt : <BOOLEAN>>
  }
  { null                  -- Note: no heading bit
  | <cond : <BOOLEAN>>    -- if 'mand' is 7 then this field is present
}
;
```

11.2.9. Sequence-of

Element values are encoded using rules for the element type. If the number of elements may vary then a length field precedes the element values.

A length field is encoded as an integer field of type INTEGER (lb..ub) where "lb" is the lower bound the of size constraint and "ub" is the upper bound.

Example: Encodings for sequence-of types in 10.3.11.~~10.3.11.~~

```
<FixedLengthList> ::=
  <Record>(10)
;

-- length = 0..10, e = length-0 = 0..10
<VariableLengthList> ::=
  <length : <INTEGER(4)> exclude {1011|1100|1101|1110|1111}>
  <Status>*val(length)
;

-- length = 1..10, e = length-1 = 0..9
<UpperLayerPDUSegments> ::=
  <length : <INTEGER(4)> exclude {1010|1011|1100|1101|1110|1111}>
  <UpperLayerPDUSegment >*val(length)
;
```

See also 11.3.~~11.3.~~

11.2.10. Choice

A choice value is encoded with a preceding tag, which indicates which alternative has been selected. A tag is encoded as an integer value in range 0..(number of alternatives-1).

Example: Encoding for choice type in 10.3.12.10.3.12.

```
<VariantRecord> ::=
  { 00 <flag : <Flag>>
  | 01 <counter : <Counter>>
  | 10 <extEnum : <ExtendedEnum>>
  }
;
```

See also 11.3.11.3.

11.2.11. Restricted character strings

A character string is mapped to a string of octets (or double octets in case of BMPString). If the number of characters may vary then a length field precedes the character string.

A length field is encoded as an integer field of type INTEGER (lb..ub) where "lb" is the lower bound the of size constraint and "ub" is the upper bound.

Example: Encodings for character string types in 10.3.13.10.3.13.

```
<FixedStr> ::=
  <Char>(10)
;

-- length = 1..10, e = length-1 = 0..9
<VarStr> ::=
  <length : <INTEGER(4)> exclude {1010|1011|1100|1101|1110|1111}>
  <Char>*val(length)
;

<FixedWStr> ::=
  <WChar>(10)
;

-- length = 1..10, e = length-1 = 0..9
<VarWStr> ::=
  <length : <INTEGER(4)> exclude {1010|1011|1100|1101|1110|1111}>
  <WChar>*val(length)
;

<Char> ::=
  bit(8)
;

<WChar> ::=
  bit(16)
;
```

11.3. Specialised encoding

Speciaised encoding can be specified only if the default encoding is specified in CSN.1. If standard ASN.1 encoding rules (BER and PER) are used then specialised encoding definitions have no effect on encoding.

11.3.1. General notation

There are three alternatives for specification of specialised encoding:

1. Definition is within ASN.1 definition in a user-defined constraint. The constraint is of the form

   Encoding(<special encoding>)

2. Definition is stand-alone and there is a reference to it within ASN.1 definition in a user-defined constraint:

   Encoding(<reference to specialised encoding>)

3. Definition is stand-alone and there is a reference to the corresponding ASN.1 definition

Specialised encoding is defined in CSN.1.

```
Example of specialised encoding, specialisation within ASN.1 definition:
B ::= BOOLEAN
   (CONSTRAINED BY {-- Encoding(<B> ::= 0|1;--)}
```

Example of specialised encoding, reference to specialisation within ASN.1 definition:

```
B ::= BOOLEAN
   (CONSTRAINED BY {-- Encoding(specialisation in the clause 2.3.4.5)--)}
```

```
<B> ::= 0|1;
```

Example of specialised encoding, reference to abstract syntax within CSN.1 definition:

```
B ::= BOOLEAN
```

```
-- Specialisation for type B specified in the clause 1.2.3.4
<B> ::= 0|1;
```

The specialised encoding shall be such that all the values of a type can be represented with it, i.e. there shall be a mapping from each abstract value to an encoded value.

Example: An integer value set is not continuous but it is evenly distributed.

```
SparseEvenlyDistributedValueSet ::= INTEGER (0|2|4|6|8|10|12|14)
   (CONSTRAINED BY {
        -- Encoding(
        --  <SparseEvenlyDistributedValueSet> ::=
        --    <INTEGER(3)>
        --  ;
        -- )
        -- Mapping: e = n/2
   }
```

Example: An integer value set is not continuous and evenly distributed.

```
SparseValueSet ::= INTEGER (0|3|5|6|8|11)
   (CONSTRAINED BY {
        -- Encoding(
        --  <SparseValueSet> ::=
        --    000|001|010|011|100|101
        --  ;
        -- )
        -- 0 => 000, 3 => 001, etc.
   })
```

Example: A list type is encoded using more bits instead of explicit length.

```
VariableLengthList ::= SEQUENCE (SIZE (0..10))
   (CONSTRAINED BY {
        -- Encoding(
        --  <VariableLengthList> ::=
        --    { 1 <Status> }(*)
        --    0
        --  ;
        -- )
   })
   OF Status
```

Example: Some alternatives of a choice type are used more frequently as others. Therefore the tags for the frequently used alternatives are specified to be shorter than others.

```
VariantRecord ::= CHOICE {
   flag      Flag,        -- The two first alternatives are mostly used
   counter   Counter,
   extEnum   ExtendedEnum,
   status    Status,
   list      VariableLengthList
}
(CONSTRAINED BY {
   -- Encoding(
   --   <VariantRecord> ::=
   --     { 00  <flag    : <Flag>>
   --     | 01  <counter : <Counter>>
   --     | 100 <extEnum : <ExtendedEnum>>
   --     | 101 <status> : <Status>>
   --     | 110 <List    : <VariableLengthList>>
   --     }
   --   ;
})
```

11.3.2. Shorthand notation

Some specialised encodings can be specified using a shorthand notation.

If an integer value set is not continuous then the encoding can be compressed by specifying the following shorthand:

Encoding(compressed)

Let there be m values in a value set. For each value n_i, $n_i < n_{i+1}$, 0 = i < m. Value n_i is encoded as value i of type INTEGER (0..m-1).

Example: A value set is not continuous but it is evenly distributed.

```
SparseEvenlyDistributedValueSet ::= INTEGER (0|2|4|6|8|10|12|14)
    (CONSTRAINED BY {-- Encoding(compressed) --})

-- Value is encoded as INTEGER (0..7)
-- 0 => 000, 2 => 001, 4 => 010, 6 => 011, 8 => 100, 10 => 101,
-- 12 => 110, 14 => 111
```

Example: A value set is not continuous and evenly distributed.

```
SparseValueSet ::= INTEGER (0|3|5|6|8|11)
    (CONSTRAINED BY {-- Encoding(compressed) --})

-- Value is encoded as INTEGER (0..5)
-- 0 => 000, 3 => 001, 5 => 010, 6 => 011, 8 => 100, 11 => 101
```

A list type can be encoded using more bits instead of explicit length indicator by specifying the following shorthand:

Encoding(morebit)

Example: A list type is encoded using more bits instead of explicit length.

```
VariableLengthList ::= SEQUENCE (SIZE (0..10))
    (CONSTRAINED BY { -- Encoding(morebit) --})
    OF Status

-- Value is encoded as:
--   <VariableLengthList> ::=
--      { 1 <Status> }(*)
--      0
--   ;
--
```

More shorthand notations are for FFS.