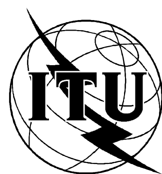
Superseded by a more recent version



INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.105

(03/95)

PROGRAMMING LANGUAGES

SDL COMBINED WITH ASN.1 (SDL/ASN.1)

ITU-T Recommendation Z.105

Superseded by a more recent version

(Previously "CCITT Recommendation")

Superseded by a more recent version

FOREWORD

The ITU-T (Telecommunication Standardization Sector) is a permanent organ of the International Telecommunication Union (ITU). The ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Conference (WTSC), which meets every four years, establishes the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

The approval of Recommendations by the Members of the ITU-T is covered by the procedure laid down in WTSC Resolution No. 1 (Helsinki, March 1-12, 1993).

ITU-T Recommendation Z.105 was prepared by ITU-T Study Group 10 (1993-1996) and was approved under the WTSC Resolution No. 1 procedure on the 6th of March 1995.

____

NOTE

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

© ITU 1995

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

CONTENTS

Page

1 Introduction ... 1
 1.1 Objective ... 1
 1.2 Differences between SDL data and ASN.1 ... 1
 1.3 The characteristics of the combination of SDL and ASN.1 ... 1
 1.4 Restrictions on SDL and ASN.1 ... 2
 1.5 The structure of this Recommendation ... 2
 1.6 Conventions used in this Recommendation ... 2

2 Additions to Recommendation Z.100 ... 3
 2.1 Lexical units ... 3
 2.2 Keywords ... 3
 2.3 Names ... 3
 2.4 Strings ... 4
 2.5 Quoted operators ... 4
 2.6 Single line notes ... 4
 2.7 Special symbols ... 4
 2.8 Use of space characters in names ... 5

3 Package ... 5

4 Definition and use of data ... 6
 4.1 Variable and data definitions ... 6
  4.1.1 Sort assignments ... 6
  4.1.2 Value assignments ... 7
 4.2 Sort Expressions ... 7
  4.2.1 Fields and optionality ... 8
  4.2.2 Sequence ... 9
  4.2.3 Sequenceof ... 12
  4.2.4 Choice ... 13
  4.2.5 Enumerated ... 14
  4.2.6 Integer Naming ... 15
  4.2.7 Subrange ... 16
 4.3 Range condition ... 16
 4.4 Value expressions ... 19
  4.4.1 Choice primary ... 19
  4.4.2 Composite primary ... 20
  4.4.3 String primary ... 22

Annex A – SDL in Combination with ASN.1 Predefined Data ... 23

Appendix I – Restrictions on ASN.1 and SDL ... 30
 I.1 Summary of the supported subset of ASN.1 ... 30
 I.2 Summary of the supported subset of SDL ... 32

Appendix II – Examples ... 32
 II.1 Defining ASN.1 data types in SDL ... 33
  II.1.1 Importing a type definition from an ASN.1 module ... 33
  II.1.2 Defining ASN.1 types directly in SDL ... 33
  II.1.3 User defined operators on ASN.1 types ... 34

Page

II.2 Using ASN.1 values in SDL 35
II.3 Illustration of the use of some ASN.1 types in SDL 36
II.3.1 Operators on simple ASN.1 types 36
II.3.2 SEQUENCE 37
II.3.3 SEQUENCE OF 38
II.3.4 SET OF 40
II.3.5 CHOICE 41
II.4 Guidelines to circumvent restrictions on ASN.1 42
II.4.1 ANY DEFINED BY / TYPE-IDENTIFIER information object class 42
II.4.2 The OPERATION macro / information object class 42
II.5 Some guidelines for the combined use of SDL and ASN.1 46

Appendix III – Supported operators for ASN.1 47
Any 47
Bit string 47
Boolean 48
Character string types 48
Choice 48
Enumerated 49
Integer 49
Null 49
Object identifier 49
Octet string 50
Real 50
Sequence 51
Sequence of 51
Set 51
Set of 51
Subtypes 52
Tagged types 52
Useful types 52

Appendix IV – Summary of the syntax 52
IV.1 <lexical unit> 53
IV.2 <special> 53
IV.3 <national> 53
IV.4 <quoted operator> 53
IV.5 <composite special> 53
IV.6 <package> 53
IV.7 <data definition> 53
IV.8 <sort> 54
IV.9 <extended properties> 54
IV.10 <range condition> 54
IV.11 <extended primary> 54
IV.12 <extended literal identifier> 55

Index 56

SUMMARY

Objective

This Recommendation defines how ASN.1 can be used in combination with SDL. The intention is that the structure and the behaviour of systems are described with SDL, while parameters of exchanged messages and internally used data are described with ASN.1. This Recommendation is an extension to Z.100. For users familiar with SDL and ASN.1, the extensions are straightforward, in the sense that ASN.1 can be used at all places where data can be used in SDL.

Coverage

This Recommendation presents a syntax and semantics definition for the combination of SDL and ASN.1. In the appendices several overviews of the combined language are given, as well as examples and guidelines for the usage of this Recommendation.

Application

The main area of application of this Recommendation is the specification of telecommunication systems. The combined use of SDL and ASN.1 permits a coherent way to specify the structure and behaviour of telecommunication systems, together with data, messages and encoding of messages that these systems use.

Status/stability

This Recommendation is the complete reference manual describing the combination of SDL and ASN.1.

Associated work

Recommendation Z.100: SDL.

Recommendation X.680: ASN.1.

SDL COMBINED WITH ASN.1 (SDL/ASN.1)

(Geneva, 1995)

1 Introduction

This Recommendation defines the combination of ASN.1 and SDL. The combination allows use of ASN.1 in SDL diagrams, and the import of ASN.1 modules in SDL descriptions.

SDL is a language for the specification and description of telecommunication systems. SDL has concepts for:

- structuring systems;
- defining behaviour of systems;
- defining data used by systems.

ASN.1 is a language for the definition of data. Related to ASN.1 are encoding rules, that define how ASN.1 values are transferred as bit streams during communication.

1.1 Objective

The combination of SDL and ASN.1 permits a coherent way of specifying the structure and behaviour of telecommunication systems, together with data, messages, and encoding of messages that these systems use: structure and behaviour using SDL, data and messages using ASN.1, encoding by reference to the relevant encoding rules that are defined for ASN.1.

The full use of SDL data types is supported by this Recommendation.

1.2 Differences between SDL data and ASN.1

Both SDL and ASN.1 provide facilities for the definition of data. But they are based on different concepts.

In SDL, a data type is characterized by the sets of values (called sorts), and the operators that are defined for the sorts. Facilities are available to define sorts, values, and operators on values. SDL has predefined sorts, and provides facilities for the definition of new sorts, values and operators, either based on existing definitions, or definitions with completely new properties. The properties of data are based on *algebraic equivalence of terms*.

ASN.1 is based on *sets*: an ASN.1 data type defines a set of values. A notation is provided to write down particular values of a data type. No operators are provided on data types (e.g. a "+" to add two integer values does not exist in ASN.1). New data types can be defined, based on existing types. By choosing suitable encoding rules (e.g. the basic encoding rules as defined in Recommendation X.690), it can be specified how values are encoded to bit streams in order to transmit them to another device.

1.3 The characteristics of the combination of SDL and ASN.1

This Recommendation is a pure extension to Recommendation Z.100, except for some small restrictions in the lexical rules (see Appendix II.2). Basically, this Recommendation defines an extension of clause 5/Z.100 on data.

Systems described in SDL combined with ASN.1 have the following characteristics:

- the structure and behaviour are defined using SDL concepts;
- the parameters of signals are defined by ASN.1 types or SDL sorts;
- data can be defined with ASN.1 type definitions or with SDL sort definitions;
- encoding of ASN.1 data values can be defined by reference to the relevant encoding rules. Encoding is not in the scope of this Recommendation.

Table 1 summarizes the data characteristics of SDL, ASN.1, and their combination:

TABLE 1/Z.105

	SDL	ASN.1	SDL/ASN.1
Definition of types	X	X	X
Notation for values	X	X	X
Definition of operators	X		X
Expressions	X		X
Encoding of values		X	X

1.4 Restrictions on SDL and ASN.1

The use of ASN.1 as defined in Recommendation X.680 is supported in combination with SDL, with a few restrictions. Appendix I.1 gives a summary of the supported subset of ASN.1. The features defined in Recommendations X.681, X.682 and X.683 are not supported.

The features of Recommendation X.208 that have been superseded in Recommendation X.680 are not supported, except for the ANY type.

The use of SDL as defined in Recommendation Z.100 is supported, with a few restrictions regarding the lexical rules.

Appendix I.2 summarizes the restrictions imposed on SDL when used in combination with ASN.1.

1.5 The structure of this Recommendation

This Recommendation is not self-contained: the language defined in this Recommendation is based on Recommendation Z.100. The language as defined in Recommendation Z.100 applies, except that 13 syntax production rules of Recommendation Z.100 are redefined in this Recommendation. This Recommendation is structured in the following manner:

Clause 2 defines the changes to the lexical rules of Recommendation Z.100.

Clause 3 defines the changes to Recommendation Z.100 in order to incorporate ASN.1 modules.

Clause 4 defines the changes to Recommendation Z.100 data chapter in order to incorporate ASN.1 data types and values.

Annex A defines the package with predefined data.

Appendix I summarizes the restrictions on the use of SDL and ASN.1.

Appendix II gives examples of the use of SDL in combination with ASN.1, together with some guidelines.

Appendix III gives an overview of the supported constructs of ASN.1, and the operators that are available for these constructs.

Appendix IV summarizes the syntactical changes to the production rules of Recommendation Z.100.

1.6 Conventions used in this Recommendation

Basically, the conventions of Recommendation Z.100 apply, i.e. keywords appear in boldface, predefined names start with a capital, etc. However, in some examples, the ASN.1 conventions are used in order to improve readability for ASN.1 users, i.e keywords are in captitals (no boldface).

In the index, only the defining occurrences are listed. Non-terminals that are referred to in the text, but not included in the index, are defined in Recommendation Z.100.

2 Additions to Recommendation Z.100

2.1 Lexical units

The production <lexical unit> is modified as follows:

```
<lexical unit>     ::=    <word> |
                          <string> |
                          <special> |
                          <composite special> |
                          <note> |
                          <single line note> |
                          <keyword>
```

NOTE – There are two changes to <lexical unit> as defined in Recommendation Z.100:

1) <character string> is replaced by <string> to allow for quotes as string delimiters.
2) <single line note> is added to support use of two dashes as start of a comment.

2.2 Keywords

The following keywords are added as alternative to the production <keyword>:

absent	**application**	**automatic**	**begin**	**by**	**choice**
component	**components**	**defined**	**definitions**	**end**	**enumerated**
explicit	**exports**	**implicit**	**imports**	**includes**	**max**
min	**of**	**optional**	**present**	**private**	**sequence**
size	**tags**	**universal**			

NOTES

1 In this Recommendation, there is no distinction between lower and upper case letters in keywords and names. In accordance with Recommendation Z.100 conventions, keywords are therefore shown in boldface lower case letters.

2 Not all keywords in Recommendation X.680 are keywords in this Recommendation. For example, BOOLEAN (in this Recommendation written as Boolean) is a keyword in Recommendation X.680, but a predefined name in this Recommendation.

2.3 Names

The productions <national> and <special> are replaced by:

```
<special>     ::=    + | - | ! | / | > | * | ( | ) | " | , | ; | < | = |
                     : | [ | ] | { | } | | | <full stop>
<national>    ::=    # | ‘ | $ | @ | \ | <overline> | <upward arrow head>
```

In a <name>, a <full stop> must not be immediately followed by an <underline> or another <full stop>.

An <underline> must not be immediately followed by a <full stop>.

NOTES

1 The above conditions imply that both a..b and a . b denote three <lexical unit>s rather than a single name. A <full stop> as a <lexical unit> (i.e. as alternative in <special>) is used in the syntax rule <existingsort>.

2 The above two productions imply that the characters left brace, right brace, left bracket, right bracket and vertical bar cannot be part of names in this Recommendation as opposed to the case in Recommendation Z.100.

2.4 Strings

<string> ::= <character string> |
<quoted string> |
<bitstring> |
<hexstring>

<quoted string> ::= **"** <text> **"**

<bitstring> ::= <apostrophe> { **0** | **1** }* <apostrophe> **B**

<hexstring> ::= <apostrophe>
{<decimal digit> | **A** | **B** | **C** | **D** | **E** | **F** }*
<apostrophe> **H**

<character string> is defined in Recommendation Z.100.

To represent a quote character (") inside a <quoted string>, two quotes are used.

<bitstring>s and <hexstring>s are the literals of the predefined sorts Bit_string and Octet_string (see 4.4.3 and Annex A).

2.5 Quoted operators

<quoted-operator> is changed from a syntactic construct to a lexical unit:

<quoted operator> ::= <quote> <infix operator> <quote> |
<quote> **not** <quote>

where <infix operator> is part of the lexical unit even though <infix operator> in other productions appears as a syntactic construct.

There is lexical ambiguity between <quoted string> and <quoted operator>, since the sequence of characters identifying a <quoted operator> is also a valid sequence of characters for identifying a <quoted string>. In such cases, the lexical unit is a <quoted operator>.

NOTES

1 In case of conflict, a <character string> has to be used instead of a <quoted string>. For example, a character string consisting of a single asterisk has to be written ’*’ as the construct "*" is always a <quoted operator>.

2 Changing <quoted operator> from a syntactic construct to a lexical unit implies that it is no longer allowed to put separators or comments inside a <quoted operator>.

2.6 Single line notes

<single line note> ::= **--** <text> [**--**]

The same rules apply for <single line note> as for <note>, except that a <single line note> starts with **--** rather than with /* and that a <single line note> is terminated by a line break or the sequence **--** (whatever comes first) rather than */.

NOTE – The two forms:

1) **task** v := 0; /* Initialization */

2) **task** v := 0; -- Initialization --

are identical, whereas the form:

task v := 0 **comment** "Initialization";

is somewhat different as the comment construct has its own notation in the graphical syntax (the comment symbol).

2.7 Special symbols

The production <composite special> is extended as follows:

<composite special> ::= **<<** | **>>** | **==** | **==>** | **/=** | **<=** | **>=** |
// | **:=** | **=>** | **>** | **(.** | **.)** | **..** | **...** | **::=**

NOTE – The last three alternatives are additions.

2.8 Use of space characters in names

Subclause 2.2.2/Z.100 lists five exceptions to the rule that space and control characters can replace underline characters in <name>s. Below are listed additional exceptions, which only apply to the new constructs defined in this Recommendation:

- The <underline>s in the <name> of a <namedvalue> must be specified explicitly.
- The <underline>s in the <name>s contained in a <composite primary> must be specified explicitly.

3 Package

The production <package> is extended as follows:

<package>	::=	<package definition> | <package diagram> | <module definition>

where <module definition> is

<module definition>	::=	<module> **definitions** [<tagdefault>] **::=** **begin** [<modulebody>] **end**
<module>	::=	<<u>package</u> name> [<objectidentifiervalue>]
<tagdefault>	::=	**explicit tags** | **implicit tags** | **automatic tags**
<modulebody>	::=	[<exports>] [<imports>] <entity in package>*
<exports>	::=	**exports** [<definition selection list>] <end>
<imports>	::=	**imports** <symbolsfrommodule>* <end>
<symbolsfrommodule>	::=	{<definition selection list> **from** <module>}*

In an <objectidentifiervalue> of a <module> only the literals and operators of the sorts defined in the predefined package Predefined are visible.

Whether and how <tagdefault> is specified, is not significant.

Model

If an <objectidentifiervalue> is present in a <module>, the <module> is transformed into a new <module> where the <objectidentifiervalue> is omitted and where the <<u>package</u> name> includes information about the value which was specified in the <objectidentifiervalue>. How this transformation actually takes place is not defined by this Recommendation.

A <module definition> has the same meaning as a <package definition> where:

- the <module> (without any <objectidentifiervalue>) corresponds to the <<u>package</u> name>;
- <imports> corresponds to the <package reference clause>s;
- <exports> corresponds to the <interface>.

Example:

```
myway DEFINITIONS ::=
BEGIN
EXPORTS yes no;
    yes BOOLEAN ::= TRUE;
    no  BOOLEAN ::= FALSE;
END
```

is the same as

package myway;

interface synonym yes, **synonym** no;

synonym yes Boolean = True,
no Boolean = False;

endpackage myway;

Similarly when the package is used in the <imports> of another package:

IMPORTS yes FROM myway;

this is the same as the <package reference clause>:

use myway/yes;

NOTES

1 A <module definition> does not end with an <end> as opposed to a <package definition>.

2 When packages/modules are used in the <system definition>, the Recommendation Z.100 notation must be used (i.e. <package reference clause> rather than <imports>).

3 As a guideline, for reasons of compatibility with Recommendation X.680, <module definition>s should as much as possible contain ASN.1 definitions.

4 Definition and use of data

4.1 Variable and data definitions

The production <data definition> is extended as follows:

<data definition> ::= {<partial type definition> |
<syntype definition> |
<generator definition> |
<synonym definition> |
<sort assignment> |
<value assignment>} <end>

The four alternatives <partial type definition>, <syntype definition>, <generator definition>, and <synonym definition> are defined in Recommendation Z.100 whereas <sort assignment> and <value assignment> are defined below.

NOTES

1 In this Recommendation, the production <data definition> allows one to use the X.680 notation for defining sorts (i.e. ASN.1 types) and synonyms. The alternatives <sort assignment> and <value assignment> are new.

2 In this Recommendation, the <end> is mandatory whereas there is no separating symbol between definitions in Recommendation X.680.

4.1.1 Sort assignments

<sort assignment> ::= <sort name> **::=** <extended properties>

Model

<sort assignment> is a short form of a <partial type definition> or a <syntype definition> having the <sort name> as the defined name.

If the <extended properties> is an <existingsort> or a <subrange>, then the <sort assignment> is the same as a <syntype definition> only containing the <extended properties>.

A <sort assignment> is the same as a <partial type definition> where <properties expression> is empty and where <formal context parameters> is omitted.

Example:

The sort assignment

Integerlist ::= SEQUENCE OF INTEGER;

is the same as

newtype Integerlist
sequence of Integer
endnewtype Integerlist;

The sort assignment

Integerlist ::= INTEGER (0..5 | 10);

is the same as

syntype S = Integer(0:5 | 10) **endsyntype** S;

NOTE – It follows that if additional literals/operators are necessary, a <partial type definition> (i.e. the full form) must be used.

4.1.2 Value assignments

<value assignment> ::= <synonym name> <sort> **::=** <ground expression>

<ground expression> is defined in Recommendation Z.100.

A <value assignment> introduces a name for a specific value.

Model

A <value assignment> is the same as a <synonym definition item>.

Example:

The definition

yes BOOLEAN ::= TRUE;

is the same as

synonym yes Boolean = True;

4.2 Sort expressions

The productions <sort> and <extended properties> are changed as follows:

<sort> ::= <sort expression>

<extended properties> ::= <sort expression>

where <sort expression> is

<sort expression> ::= {<existingsort> | <subrange> | <sort constructor> |
<inheritance rule> | <generator transformations> |
<structure definition>}

<existingsort> ::= [<package name> **.**] {<sort identifier> | <syntype identifier>} |
any [**defined by** <identifier>] |
<selection>

<selection> ::= <name> **<** <sort>

<sort constructor> ::= <tag> <sort expression> |
<sequence> |
<sequenceof> |
<choice> |
<enumerated> |
<integernaming>

<tag> ::= **[[universal | application | private]**
<simple expression> **]** [**implicit** | **explicit**]

If **defined by** <identifier> is specified, the <identifier> must denote a sort or syntype. The <identifier> is not otherwise significant.

The alternatives <existingsort> and <subrange> must not be used as the <extended properties> of a <partial type definition> or <syntype>.

A <sort expression> defines the properties of a sort. It is not significant whether <tag>s are present. However, if present, the contained <simple expression> must be of sort Integer.

A <sort>, in a referenced <operator definition> (see Recommendation Z.100), which by position corresponds to a <sort> in a <textual operator reference>, must syntactically be the same as that <sort>.

A <sort>, in an <operator signature> (see Recommendation Z.100), which by position is corresponding to a <sort> in an <operator definition>, must syntactically be the same as in the <operator definition>. However, if the <sort> is an <existingsort>, they need not be syntactically the same, but they must denote the same sort or syntype.

There is syntactic ambiguity between a <generator transformations> and a <subrange>. A <sort> which starts with an <identifier> followed by a left parenthesis is treated as a <sort identifier> or <syntype identifier>, if possible according to the visibility rules (see 2.2.2/Z.100) and otherwise as a <generator identifier>.

Model

In the following description of how the above constructs are transformed into their Z.100 representation, the order is significant:

- The keyword **any** is derived syntax for giving the predefined sort Any_type (see Annex A).
- A <selection> is derived syntax for giving the sort of the field given by the <name>. This field must be associated to the <sort>.
- An <existingsort> represents a <sort identifier> or <syntype identifier>. If a <module> is present, it represents a <package name> (as explained in clause 3) which then constitutes the leftmost part of the qualifier of the identifier.
- A <sort> which is neither an <existingsort> nor a <subrange> represents a <sort identifier> of an implicitly defined sort. This sort has an anonymous and unique name and has the <sort expression> as its <extended properties>. The sort is defined in the nearest enclosing scope unit.
- A <sort> which is a <subrange> represents a <syntype identifier> of an implicitly defined syntype containing the <subrange>. This syntype has an anonymous and unique name and is defined in the nearest scope unit enclosing the place where the <sort> occurs. A special Model applies for a <subrange> being a <parent sort identifier> (see 4.2.7).
- A <partial type definition> containing <extended properties> being a <sort constructor> is represented as specified in the subclauses below.

4.2.1 Fields and optionality

For convenience in the *Model* for the type and value notations, the notion of fields is defined:

To a sort, zero or more field names are associated. The field names of a sort are those names which, when concatenated with the name Extract!, form operator names of operators which are locally defined for the sort and which have one argument being the sort. Each field also has a *field sort* which is the result sort of the operator.

A field is *optional* if there in addition exists a local operator whose name is the field name concatenated with the name Present, whose (only) argument is the sort having the field associated and whose result sort is the predefined sort Boolean.

NOTE 1 – In the underlying data model (ACT ONE) the properties of a sort are characterized only by its literals, operators and axioms. Composite data structures like SDL <structure definition> and ASN.1 <sequence>s are therefore derived notations for defining literals, operators and axioms. For this reason, the notion of fields and optional fields is synthesized from existence of specific operators rather than being properties of sorts defined with a construct like <sequence>.

NOTE 2 – From the *Model* of <sequence> (see 4.2.2) it follows that a field having a default value is treated as an optional field which always contains a well-defined value and whose Present operator always yields True.

NOTE 3 – It follows that sorts defined with the shorthand notations <sequence> and <choice> have fields associated. However, use of these shorthand notations is not the only way to associate fields to sorts.

NOTE 4 – It follows that fields specified with the keyword **default** inside a <sequence> (see 4.2.2) are also optional fields.

Example:

As is explained in 4.2.2, the <sequence>:

```
S ::= SEQUENCE {
        a       INTEGER,
        b       REAL OPTIONAL,
        c       IA5String DEFAULT "initial string" };
```

is the same as

```
newtype S
    operators
    Make!      : Integer      -> S;
    AExtract!  : S            -> Integer;
    BExtract!  : S            -> Real;
    CExtract!  : S            -> IA5String;
    AModify!   : S, Integer   -> S;
    BModify!   : S, Real      -> S;
    CModify!   : S, IA5String -> S;
    BPresent   : S            -> Boolean;
    CPresent   : S            -> Boolean;
axioms
/* Properties of the above operators defined here, see 4.2.2 */
endnewtype S;
```

NOTE 5 – From this newtype definition, it follows (as expected) that the sort S has three fields A, B and C, where B and C are optional fields. It also follows that the field sort of A is Integer, the field sort of B is Real and the field sort of C is IA5String.

4.2.2 Sequence

<sequence> ::= { **sequence** | **set** } **{** [<elementsort> { **,** <elementsort>}*] **}**

<elementsort> ::= <namedsort> [**optional** | **default** <ground expression>] |
components of <sort>

<namedsort> ::= [<name>] <sort>

The <name> in <namedsort> may only be omitted if the <sort> is a <selection>. In this case, the <name> is derived as being the same as the <name> contained in the <selection> (this applies for all usages of <namedsort> in this Recommendation).

It is not significant whether the keyword **sequence** or the keyword **set** is specified.

A <ground expression> in an <elementsort> must be of the sort denoted by the <sort> contained in the <namedsort> of the <elementsort>.

The <name>s in the <namedsort>s of the <sequence> must be distinct.

Model

Although not shown in the axioms derived in the *Model*, all operator identifiers contained in the axioms are fully qualified to avoid ambiguity with operators of other sorts.

An <elementsort> which is **components of** <sort> is derived syntax for a list of ordered <elementsort>s, one for each field associated to <sort> (see 4.2.1). Those <elementsort>s which in <sort> are optional fields have the keyword **optional** specified. The field names are ordered using the same rules as for **nameclass** literal names, see 5.3.1.14/Z.100.

A <sequence> is derived syntax for a <properties expression> containing:

- "Extract!" operators which associate the field <name>s with the <sequence> (see 4.2.1).
- "Present" operators which make optional fields of those <name>s in <namedsort>s which are followed by the keyword **optional** or **default** (see 4.2.1).
- a "Make!" operator, to create sequence values, or the literal Empty if all fields are optional or no <elementsort>s are specified. In the case of the literal Empty, all occurrences of the Make! operator in the axioms of the *Model* are regarded as that literal instead.
- "Modify!" operators to modify fields of sequence values. The name of the implied operator for modifying a field is the field <name> concatenated with "Modify!".

The <argument list> for the Make! operator is the list of <sort>s occurring in the <elementsort>, ordered according to the field names. The field names are ordered using the same rules as for **nameclass** (see 5.3.1.14/Z.100).

The field <sort>s which are followed by the keyword **optional** or **default** are excluded from the list. The <result sort> for the Make! operator is the sort of the enclosing context. This is also the <sort> of the sequence.

The <argument list> for each field modify operator is the <sort> of the sequence followed by the field <sort> of that field. The <result sort> for a field modify operator is the <sort> of the sequence.

The <argument list> for each field extract operator is the <sort> of the sequence. The <result sort> for a field extract operator is the field <sort> of that field.

The <argument list> for each field present operator is the <sort> of the sequence.

The <result sort> for each field present operator is the predefined sort Boolean.

For each field, there is an axiom describing the meaning of extracting a field:

```
FExtract!(Make!(a,b,...,n)) == E;
```

where F is the name of the field, and where E is that argument to the Make! operator which by position corresponds to the field F in case F is not an optional field. If F is an optional field specified with **optional** then E is **error!**, and if the field is specified with **default** then E is the associated <ground expression>. If F is an optional field derived from the **components of** construct, then E is FExtract!(<<**type** S1>> Make!(a1,b1,...,n1)), where S1 is the <sort> mentioned in the **components of** construct.

For each optional field, there is a present operator that is defined by axioms describing the meaning of testing for presence of a field:

```
FPresent(Make!(a,b,...,n))      == B;
FPresent(FModify!(s,t))         == True;
FPresent(GModify!(s,t))         == FPresent(s);
```

where F and G are names of different fields and F is an optional field. B is True if F is specified with **default**. If F is derived from the **components of** construct then B is FPresent(<<**type** S1>> Make!(a1,b1,...,n1)), where S1 is the <sort> mentioned in the **components of** construct. In other cases, B is False.

For every field there are axioms that describe that its value is not influenced by modifying other fields:

```
FExtract!(GModify!(s,t)) == FExtract!(s);
```

where F and G are names of different fields.

There is also an axiom expressing equality of sequence values:

for all s1, s2 in S

```
(s1 = s2 ==     A1Extract!(s1) = A1Extract!(s2) and
                A2Extract!(s1) = A2Extract!(s2) and
                ... and
                if B1Present(s1) and B1Present(s2) then
                B1Extract!(s1) = B1Extract!(s2) else
                B1Present(s1) =} B1Present(s2) fi and
                if B2Present(s1) and B2Present(s2) then
                B2Extract!(s1) = B2Extract!(s2) else
                B2Present(s1) =} B2Present(s2) fi and
                ... and
                if BnPresent(s1) and BnPresent(s2) then
                BnExtract!(s1) = BnExtract!(s2) else
                BnPresent(s1) =} BnPresent(s2) fi;
```

where S is the sequence sort, A1,...,An are the non-optional fields and B1,...,Bn are the optional fields.

Example:

The sequence sort:

```
newtype S
    sequence {
        A Integer,
        B Charstring optional,
        C Character default 'd'}
endnewtype S;
```

is the same as

```
newtype S
    operators
        Make!       : Integer       -> S;
        AExtract!   : S             -> Integer;
        BExtract!   : S             -> Charstring;
        CExtract!   : S             -> Character;
        AModify!    : S, Integer    -> S;
        BModify!    : S, Charstring -> S;
        CModify!    : S, Character  -> S;
        BPresent    : S             -> Boolean;
        CPresent    : S             -> Boolean;
    axioms
      for all i in Integer (
      for all s, s1, s2 in S (
      for all t in Integer (
      for all cstr in Charstring (
      for all c in Character (
        AExtract!(Make!(i))   == v;
        BExtract!(Make!(i))   == error!;
        CExtract!(Make!(i))   == 'd';

        BPresent(Make!(i))            == False;
        BPresent(BModify!(s,cstr))    == True;
        BPresent(CModify!(s,c))       == BPresent(s);
        CPresent(s)                   == True;
```

```
AExtract!(BModify!(s,cstr))  == AExtract!(s);
AExtract!(CModify!(s,c))     == AExtract!(s);
BExtract!(AModify!(s,i))     == BExtract!(s);
BExtract!(CModify!(s,c))     == BExtract!(s);
CExtract!(AModify!(s,i))     == CExtract!(s);
CExtract!(BModify!(s,cstr))  == CExtract!(s);
s1 = s2 ==     AExtract!(s1) = AExtract!(s2) and
               if BPresent(s1) and BPresent(s2) then
                      BExtract!(s1) = BExtract!(s2)
               else BPresent(s1) = BPresent(s2) fi and
               if CPresent(s1) and CPresent(s2) then
                      CExtract!(s1) = CExtract!(s2)
               else CPresent(s1) = CPresent(s2) fi;
) ) ) ) )
endnewtype S;
```

NOTES

1 Fields specified with **default** have a "Present" operator, and are therefore treated as an optional field which yields a default value instead of an error if accessed before being assigned a value.

2 The fields derived from **components of** must be ordered in some way such that the order of sorts in the <argument list> of the Make! operator is deterministic. In this Recommendation, alphabetic order is used.

3 There is no distinction between use of keyword **sequence** and **set**. This is a relaxation compared to Recommendation X.680.

4 In this Recommendation, tags are not necessary to distinguish between components of the same type: ASN.1 automatic tagging is assumed.

4.2.3 Sequenceof

<sequenceof> ::= { **sequence** | **set** } [<sizeconstraint> | <asn1 range condition>]
of <sort>

Model

Specifying a <sequenceof> is the same as specifying a <generator transformation> having <sort> as the first <generator actual> and the name Emptystring as the second <generator actual>. The generator identifier in the <generator transformation> is the predefined generator String in case the keyword **sequence** is specified, otherwise the generator identifier is the predefined generator Bag.

If <sizeconstraint> is specified, the <sequenceof> is a syntype having the <sizeconstraint> as <range condition> (see 4.2.7).

If <asn1 range condition> is specified, the <sequenceof> is a syntype having the <range condition> as specified in the <asn1 range condition> (see 4.2.7).

The parent sort of the syntype (i.e. the <sequenceof> without the <sizeconstraint> or <asn1 range condition>) has an implicit and unique name and is defined in the nearest scope unit enclosing the place where the <sequenceof> occurs.

NOTE – The generator String is defined in Recommendation Z.100. The generator Bag is defined in Annex A. In principle, the ordering of the elements in a Bag is not significant as opposed to the elements in a String. Even though the elements are in fact ordered in some way in the predefined Bag generator (to allow for applying range conditions on each element), it is not a good practice to utilize this information.

Example:

The definition:

```
phonenumber ::= SEQUENCE SIZE (8) OF INTEGER (0..9);
```

is the same as the three SDL definitions:

```
newtype S1 String(S2,Emptystring) endnewtype;
syntype S2 = Integer constants 0:9 endsyntype;
syntype phonenumber = S2 constants size (8) endsyntype  phonenumber;
```

Note that the **size** construct is an extension to the abstract grammar of SDL (see 4.2.7).

4.2.4 Choice

<choice> ::= **choice { ** [<namedsort> { **,** <namedsort>}*] **}**

Model

The <name>s in the <namedsort>s of the <choice> must be distinct. A <choice> is derived syntax for a <properties expression> containing:

- "Make!" operators, to create choice values;
- "Extract!" operators, to associate the <name>s in the <namedsort>s to the <choice> (see 4.2.1);
- "Present" operators, to make the <name>s in <namedsort>s optional fields (see 4.2.1);
- "Modify!" operators, to modify choice values. The name of the implied operator for modifying a component is the field component in the field <name> concatenated with "Modify!".

There is one "Make!" operator for each <namedsort>. Each "Make!" operator has the name "FMake!" where F is the field <name> occurring in <namedsort>. The <argument list> of the "FMake!" operator is the field <sort> occurring in <namedsort>. The <result sort> for the "FMake!" operators is the sort which encloses the <properties expression>.

Names and signatures of the "Extract!, "Modify!" and "Present" operators for fields are the same as for <sequence>s (see 4.2.2).

The <choice> also implicitly defines an enumerated sort which has literal names identical to the field names of the <choice>. The order of the literals is the same as the order in which the fields are specified. To describe the *Model*, the literals are numbered 1, 2, 3, ... n, where n is the number of fields/literals, excluding the derived field (as defined below). There is an implicit derived field of this implicit enumerated sort. This field indicates which explicit field is present at any time. The name of the field is "Present".

For each field F, there is a set of equations describing the meaning of extracting the field:

```
FExtract!(F1Make!(A)) == E;
FExtract!(F2Make!(A)) == E;
...
FExtract!(FnMake!(A)) == E;
```

where E is **error!** except for the equation mentioning "FExtract!" and "FMake!" for the same field. In this case E equals A.

For each field F, there is a set of equations describing the meaning of testing for presence:

```
FPresent(F1Make!(A)) == B;
FPresent(F2Make!(A)) == B;
...
FPresent(FnMake!(A)) == B;
```

where B is False except for the equation mentioning "FPresent" and "FMake!" for the same field. In this case B equals True.

For each field F, there is a set of equations describing the meaning of modifying a choice value:

```
F1Modify!(s,f) = F1Make!(f);
F2Modify!(s,f) = F2Make!(f);
...
FnModify!(s,f) = FnMake!(f);
```

For each field F, there is a set of equations describing the value for the implicit "Present" field:

```
PresentExtract!(F1Make!(v)) = F1;
PresentExtract!(F2Make!(v)) = F2;
...
PresentExtract!(FnMake!(v)) = Fn;
```

There is one equation describing equality of choice values:

```
s1 = s2 ==      if PresentExtract!(s1) /= PresentExtract!(s2) then False
                else if F1Present(s1) then F1Extract!(s1) = F1Extract!(s2)
                else if F2Present(s1) then F2Extract!(s1) = F2Extract!(s2)
                ...
                else if FnPresent(s1) then FnExtract!(s1) = FnExtract!(s2)
                else False fi ... fi
```

NOTES

1 The derived field and the derived enumerated type is a field keeping information about which choice field is active. The "PresentExtract" operator can thus be used for multi-branching.

2 In this Recommendation, tags are not necessary to distinguish between components of the same type: ASN.1 automatic tagging is assumed.

Example:

The choice type

```
C ::= CHOICE {
        a     INTEGER,
        b     REAL };
```

is the same as

```
xxx ::= enumerated {A(1),B(2) };

newtype C
  operators
        PresentExtract!: C          -> xxx;
        AExtract!   : C             -> Integer;
        BExtract!   : C             -> Real;
        AMake!      : Integer       -> C;
        BMake!      : Real          -> C;
        AModify!    : C, Integer    -> C;
        BModify!    : C, Real       -> C;
        APresent    : C             -> Boolean;
        BPresent    : C             -> Boolean;
axioms
     for all s, s1, s2 in C (
     for all i in Integer (
     for all r in Real (
        AExtract!(AMake!(i))        == i;
        AExtract!(BMake!(r))        == error!;
        APresent(AMake!(i))         == True;
        APresent(BMake!(r))         == False;
        BExtract!(AMake!(i))        == error!;
        BExtract!(BMake!(r))        == A;
        BPresent(AMake!(i))         == False;
        BPresent(BMake!(r))         == True;
        AModify!(s,i)               == AMake!(i);
        BModify!(s,r)               == BMake!(r);
        PresentExtract!(AMake!(i))  == A;
        PresentExtract!(BMake!(r))  == B;
        s1 = s2 ==  if PresentExtract!(s1) /= PresentExtract!(s2) then False
                    else if APresent(s1) then AExtract!(s1) = AExtract!(s2)
                    else if BPresent(s1) then BExtract!(s1) = BExtract!(s2)
                    else False fi fi fi;
     ) ) )
endnewtype C;
```

4.2.5 Enumerated

<enumerated> ::= **enumerated** **{** <named number> { **,** <named number>}* **}**

<named number> ::= <named value> | <name>

The <simple expression>s in the <named value>s must be of the Integer sort and the values must be disjoint.

Model

A <named number> being a <name> is derived syntax for a <named value> containing the <name> and containing a <simple expression> denoting the lowest possible non-negative integer value not occurring in any other <named value>s of the <enumerated> sort. The replacement of <name>s by <named value>s takes place one by one from left to right. After this replacement, the literals are reordered on the basis of their <simple expression>s.

An <enumerated> sort is represented as a specialization of the predefined sort Enumeration as defined in Annex A. The <name>s in the <named value>s are new literals added in ascending order based on their associated <simple expression>. The specialized sort has axioms which define First to denote the value of the literal with the lowest <simple expression>, Last to denote the value of the literal with the highest <simple expression>, Succ to denote the value of the next-higher literal (if it exists, otherwise **error!**), Pred to denote the value of the next-lower literal (if it exists, otherwise **error!**) and Num to denote the Integer value of the <simple expression>.

The definition:

```
colours ::= ENUMERATED {blue(3),red,yellow(0)};
```

is the same as

```
newtype colours
    inherits Enumeration
    adding
    literals blue,red,yellow;
    axioms
      for all c in colours (
         First(c)        == yellow;
         Last(c)         == blue;
         Succ(yellow) == red;
         Succ(red)       == blue;
         Succ(blue)      == error!;
         Pred(yellow)  == error!;
         Pred(red)       == yellow;
         Pred(blue)      == red;
         Num(yellow) == 0;
         Num(red)       == 1;
         Num(blue)      == 3;
      )
endnewtype colours;
```

NOTE – An enumerated sort also has relational operators, but since the predefined Enumerated sort is defined with the keyword **ordering**, the properties for the relational operators are not given in the subtype.

4.2.6 Integer Naming

<integernaming> ::= <identifier> **{** <named value> { **,** <named value>}* **}**

<named value> ::= <name> **(** <simple expression> **)**

The <identifier> must denote the predefined sort Integer or the predefined sort Bit_String and the <simple expression>s in the <named value>s must be of the Integer sort.

Model

Specifying an <integernaming> is the same as specifying an <existingsort> containing the Integer <identifier> and specifying a <synonym definition item> for each <named value> in the nearest enclosing scope unit.

Example:

The **newtype** definition:

newtype standards
    sequence of Integer{Z.100(0),X.680(1),Z.105(2)}
endnewtype standards;

is the same as

newtype standards
    sequence of Integer
endnewtype standards;
synonym Z.100 Integer = 0,
    X.680 Integer = 1,
    Z.105 Integer = 2;

NOTES

1 An <integernaming> provides a means for grouping together synonym definitions which are related somehow. The grouping may be convenient for readability, but from the point of view of this Recommendation, the grouping is not significant.

2 There are no bindings between the Integer synonym values defined by the <named value>s and the Integer or Bit_string sort denoted by the <integernaming>.

4.2.7 Subrange

<subrange> ::= <sort> (<range condition>)

A <subrange> defines the range of a syntype. When a <sequenceof> or <selection> is followed by **(** <range condition> **)**, the <subrange> applies to the contained <sort> rather than to the entire <sequenceof> or <selection> construct.

Model

Specifying <subrange> as the <<u>parent sort</u> identifier> of a <syntype> is the same as specifying the contained <sort> and adding the <range condition> after the **constants** keyword in the <syntype> (if specified, otherwise the construct is introduced).

Example:

The syntype definition:

syntype s = Integer(0..5 | 10) **endsyntype** s;

is equivalent to

syntype s = Integer **constants** 0..5 | 10 **endsyntype** s;

How this construct corresponds to a syntype in Recommendation Z.100 is described below.

4.3 Range condition

The abstract grammar for range condition is changed as follows:

Range-condition :: *Operator-identifier*

A range condition defines a set of values. It is used for defining a syntype and for branching in a decision action. It has an associated parent sort, which is for a syntype the sort specified in the syntype definition, and for decision actions is the question expression. A value is within the value set if the operator denoted by the operator identifier yields True when applied to the value.

The operator identifier for a given range condition is thus defined as:

newtype A
    operators o: S -> Boolean;
    axioms
    /* Derived from the concrete syntax as explained below */
endnewtype A;

where o is the implicit and anonymous operator occurring in the range condition, A is an implicit and anonymous sort and S is the parent sort.

The concrete syntax for range condition is changed as follows:

<range condition> ::= <range> { { **,** | **|** } <range>}*

<range> ::= <closed range> |
<open range> |
<contained subrange> |
<sizeconstraint> |
<innercomponent> |
<innercomponents>

<closed range> ::= <lowerendvalue> { **:** | **..** } <upperendvalue>

<lowerendvalue> ::= {<ground expression> | **min** } [**<**]

<upperendvalue> ::= [**<**] {<ground expression> | **max** }

<open range> ::= [**=** | **/=** | **<** | **>** | **<=** | **>=**] <ground expression>

<contained subrange> ::= **includes** <sort>

<sizeconstraint> ::= **size** **(** <range condition> **)**

<innercomponent> ::= { **from** | **with component** } **(** <range condition> **)**

<innercomponents> ::= **with components**

{ [**... ,**] <named constraint> { **,** <named constraint>}* **}**

<named constraint> ::= <name> [<asn1 range condition>]

[**present** | **absent** | **optional**]

<asn1 range condition> ::= **(** <range condition> **)**

It is not significant whether the keyword **from** or the keywords **with component** is used in <innercomponent>.

It is not significant whether parentheses enclose the <range list>.

It is not significant whether the colon symbol (**:**) or the two dots symbol (**..**) is used in a <closed range>.

There is syntactic ambiguity between a <closed range> specified with **:** and a <range> starting with a <ground expression> which is a <choice primary> (see 4.4.1). In this case, it is determined by context (see 2.2.2/Z.100) which one is meant. A <ground expression> which is a <choice primary> must be enclosed in parenthesis when occurring as an entire <lowerendvalue> (e.g. A((b:c):d)) or <open range> (e.g. A((b:c))).

It is not significant whether the comma symbol (**,**) or the vertical bar symbol (**|**) is used in <range condition>.

The <name>s in the <named constraint>s of an <innercomponents> must denote names for optional fields (see 4.2.1) of the parent sort.

The field names must be distinct.

Each <range> in the <range condition> contributes to the properties of the operator defining the value set:

o(V) == range1 **or** range2 **or** ... **or** rangeN

If a syntype is specified without a <range condition> then the operator result is True.

Superseded by a more recent version

In the following explanation of how each <range> contributes to the operator result, V denotes the argument value. Each contribution must be well-formed, which means that used operators must exist with a signature appropriate for the context.

- If neither of the keywords **min** and **max** are specified in a <closed range> then a <closed range> contributes with:

  E1 rel1 V and V rel2 E2

  where E1 is <ground expression> of <lowerendvalue> and E2 is <ground expression> of <upperendvalue>.

  If "<" is specified for <lowerendvalue> then rel1 is the "<" operator, otherwise it is the "<=" operator.

  If "<" is specified for <upperendvalue> then rel2 is the "<" operator, otherwise it is the "<=" operator.

  If the keyword **min** is specified and the keyword **max** is not specified, <range condition> contributes with:

  V rel2 E2

  If the keyword **max** is specified and the keyword **min** is not specified, <range condition> contributes with:

  E1 rel1 V

  If both keywords **min** and **max** are specified, the operator always yields True.

- An <open range> contributes with:

  V rel E

  where E is the <ground expression> of the <open range> and rel is the infix operator of the <open range>. If no infix operator is specified in the <open range>, rel is the = operator.

- A <contained subrange> contributes with:

  o1(V)

  where o1 is the implicit operator defining the value set for the <sort> mentioned in the <contained subrange>.

- A <sizeconstraint> contributes with:

  o1(Length(V))

  where o1 is the implicit operator defining the value set for the <range condition> mentioned in the <sizeconstraint>.

- <innercomponent> contributes with either:

  if Length(V) = 0 **then** True **else** o1(First(V)) **and** o(Substring(V,2,Length(V)-1)) **fi**; or

  if Length(V) = 0 **then** True **else** o1(Take(V)) **and** o(Del(Take(V), V)) **fi**

  whatever is appropriate for the sort of V. o is the implicit operator <innercomponent> contributes to and o1 is the implicit operator for the <range condition> specified in <innercomponent>.

  <innercomponents> has a contribution for each contained <named constraint> which specifies constraints of the field (see 4.2.1) denoted by <name> of the parent sort.

  If the symbol **...** is omitted, the keyword **present** is added to the <named constraint>s having no ending keyword (**present**, **absent** or **optional**) and <named constraint>s of the form:

  <name> **absent**

  are added for all fields (i.e. <name>s) not mentioned explicitly in a <named constraint>. The <named constraint>s are added to the <innercomponents> before the contributions of each <named constraint> are derived.

Superseded by a more recent version

If a <range condition> is specified for a <named constraint>, the contribution is:

E **and if** FPresent(V) **then** o1(V) **else** True **fi**

where E is the present constraint for the field, F (from the operator name FPresent) is the name of the optional field and o1 is the implicit operator for the <range condition>. If the <range condition> is omitted, the contribution is only the present constraint E.

The present constraint for a field F is:

FPresent(V)

in case the <named constraint> for the field contains the keyword **present**; and

not FPresent(V)

in case the <named constraint> for the field contains the keyword **absent**. In all other cases, the present constraint is True.

4.4 Value expressions

```
<extended primary>            ::=    <synonym> |
                                     <indexed primary>  |
                                     <field primary> |
                                     <structure primary> |
                                     <choice primary> |
                                     <composite primary>

<extended literal identifier> ::=    <character string literal identifier> |
                                     <generator formal name> |
                                     <string primary>
```

NOTES

1 In <extended primary>, <choice primary> and <composite primary> are additional to Recommendation Z.100.

In <extended identifier>, <string primary> is additional to Recommendation Z.100.

2 <extended primary> is part of expressions, while <extended literal identifier> is part of terms in equations.

4.4.1 Choice primary

```
<choice primary>    ::=    <identifier>  :  <primary>
```

Model

A <choice primary> is derived syntax for an <operator application> having the <primary> as argument. The <operator identifier> in the <operator application> contains the <qualifier> of the <identifier> and an operator name being the <name> in the <identifier> followed by "Make!".

Example:

The <choice primary>:

Myvalue : Mychoice

is derived syntax for:

MyvalueMake!(Mychoice)

If a particular <choice primary> can denote one of several operator applications (i.e. a field of more than one choice sort), a qualifier is used:

<< **type** Mytype >> Myvalue : Mychoice

which is then derived syntax for:

<< **type** Mytype >> MyvalueMake!(Mychoice)

NOTE – Operator names consisting of a field name followed by "Make!" are available in every sort constructed using <choice>. The <choice primary> notation is therefore mainly used on such sorts. However, the notation is also suitable for any values, for example, an operator RealMake! can be defined with the signature:

RealMake! : **any** -> Real;

which implies that the notation:

Real : <expression>

can be used to "convert" an **any** expression to a Real value.

4.4.2 Composite primary

<composite primary> ::= [<qualifier>]
{<sequencevalue> |
<sequenceofvalue> |
<objectidentifiervalue> |
<realvalue>}

A <composite primary> is a value constructed using its constituent components. There is syntactic ambiguity between the various alternatives in <composite primary>. In those cases, it must be determined by context (see 2.2.2/Z.100) what the <composite primary> denotes.

NOTE – This probably implies that tools only distinguish <sequencevalue> and <sequenceofvalue> in the concrete syntax. As part of the static analysis it can be determined whether a <sequenceofvalue> instead denotes one of the remaining alternatives.

4.4.2.1 Sequence value

<sequencevalue> ::= **{** [<namedvalue> { **,** <namedvalue> }*] **}**

<namedvalue> ::= <name> <expression>

The sort of the <sequencevalue> must have the <name>s occurring in the <namedvalue>s as associated fields (see 4.2.1). The <expression> in each <namedvalue> must be of the sort of the field given by the <name>. Each non-optional field associated to the sort must be mentioned exactly once and each optional field associated to the sort must be mentioned at most once.

Model

A <sequencevalue> is derived syntax for:

- An operator application of an operator with the name "Make!". It takes as arguments the <expression>s of the non-optional fields. The <expression>s in the derived operator application occurs in alphabetic order sorted by the field <name>s. If all fields associated to the sort are optional, the construct is a literal identifier with the name "Empty".
- If the sort has optional fields, the operator application is given as first argument to an operator application of an operator with the name "FModify!", where F is the alphabetically first <name> of the optional fields mentioned in the <sequencevalue>. The second argument of the "FModify!" operator is the <expression> associated to the field. The resulting operator application is in turn given as first argument to the "FModify" operator of the next optional field and so on until all optional fields have been considered.

Example:

If the sort S has four fields A, B, C, D associated and C and D are optional, then the <composite primary>:

<< **type** S >> {A E1, B E2, C E3, D E4}

is derived syntax for:

<< **type** S >> DModify!(CModify!(Make!(E1,E2),E3),E4)

The order of the <namedvalue>s in the <sequencevalue> is not significant, e.g.:

<< **type** S >> {D E4, C E3, B E2, A E1}

denotes the same expression. The <qualifier> can be omitted:

{A E1, B E2, C E3, D E4}

in which case it must be possible to determine by context (see 2.2.2/Z.100) which sort is meant.

NOTE – The <sequencevalue>s are mostly used for constructing values of sorts defined with the <sequence> construct, since such sorts have a "Make!"operator whose arguments are field values ordered alphabetically according to the field names. For sort defined with the <structure definition> construct, it is often better to use the <structure primary> as the order of appearance is important for the "Make!" operator for such sorts.

4.4.2.2 Sequenceof value

<sequenceofvalue> ::= **{** [<expression> { **,** <expression> }*] **}**

Model

A <sequenceofvalue> is derived syntax for:

MkString(E1) // MkString(E1)// ... // MkString(En)

where E1, E2, ...,En are the <expression>s of the <sequenceofvalue> in the order of appearance. If no <expression>s are specified, the <sequenceofvalue> is derived syntax for the name Emptystring.

NOTE – A <sequenceofvalue> may denote a value of the predefined sort Octet_string, even though Recommendation X.680 (as opposed to this Recommendation) requires the <expression>s to be <identifier>s in this case.

4.4.2.3 Object identifier value

<objectidentifiervalue> ::= **{** <objidcomponent>+ **}**

<objidcomponent> ::= <identifier> [**(** <ground expression> **)**]

Model

If the leftmost <objidcomponent> is a synonym or variable <identifier> of the predefined sort Object_identifier, then at least one <objidcomponent> must follow. The <objectidentifiervalue> is then the same as the expression:

Id // {Olist}

where Id is the first <objidcomponent> and Olist is the remaining <objidcomponent>s.

If an <objidcomponent> is specified with a <ground expression>, then it is derived syntax for defining the <identifier> as a **synonym** for the <ground expression> in the nearest enclosing scope unit and omitting the <ground expression> in the <objidcomponent>.

After the above transformations, all <objidcomponent>s are <identifier>s, which must denote a variable or synonym of the predefined sort Object_element (see Annex A). The construct is then derived syntax for:

MkString(Id1) // MkString(Id2)// ... // MkString(Idn)

where Id1, Id2, ..., Idn are the <identifier>s of the <objectidentifiervalue> given in the same order as in the <objectidentifiervalue>.

Example:

The <objectidentifiervalue>:

{root,1,2,level3,level4(4)}

is the same as:

root // mkstring(1) // mkstring(2) // mkstring(level3) // mkstring(level4)

where root is a variable or synonym of the predefined sort Object_identifier, and a synonym definition of level4 is derived in the nearest enclosing scope unit:

synonym level4 Object_element = 4;

4.4.2.4 Real value

<realvalue> ::= **{** <mantissa> **,** <base> **,** <exponent> **}**

<mantissa> ::= <expression>

<base> ::= <simple expression>

<exponent> ::= <expression>

All three constituent expressions must be of the Integer sort. The <base> must be identical to the value 2 or 10.

Model

A <realvalue> is represented as the <expression>: (<mantissa>)*Power(<base>,<exponent>) where Power is an additional operator defined for the Real sort.

The operator has the signature:

```
Power : Integer, Integer -> Real;
```

and the equations:

```
for all a,b in Integer (  Power(a,0) == 1;
                          b>=0   ==>   Power(a,b+1)  == Power(a,b)*a;
                          b<0    ==>   Power(a,b)    == 1/Power(a,-b);
                       )
```

Example:

The expression:

```
{3,2,-1}
```

is the same as:

```
3*Power(2,-1)
```

which is equal to 1.5

4.4.3 String primary

<string primary> ::= [<qualifier>] {<bitstring> | <hexstring> | <quoted string>}

<bitstring>s and <hexstring>s are the literals of the predefined sorts Bit_string and Octet_string (see Annex A).

Whether any specific <bitstring> or <hexstring> is a literal of Bit_string or Octet_string is determined by context or by using a <qualifier>.

Model

A <string primary> containing a <quoted string> represents a <character string literal identifier> consisting of the <qualifier> and a <character string literal> with the same <text> as the <quoted string>.

A <string primary> containing a <bitstring> or <hexstring> represents an identifier consisting of the <qualifier> and a <name> formed by removing the apostrophes from the <bitstring> or <hexstring>.

NOTES

1 A <quoted string> is a character string where quotation marks (") have been used rather than apostrophes.

2 As a guideline, value notations for bit strings and hex strings should use apostrophes, e.g. '01'B should be used instead of 01B.

Superseded by a more recent version

Annex A

SDL in Combination with ASN.1 Predefined Data

(This annex forms an integral part of this Recommendation)

This annex defines package Predefined, which contains the predefined data sorts and data generators.

```
package Predefined

/* Boolean: defined in Annex D/Z.100 */

/* Character: defined in Annex D/Z.100*/

/* Real: defined in Annex D/Z.100, but extended with operator Power
      (see 4.2.4.4) */

/* String0 generator */

/* Definition            */

generator String0(type Itemsort, literal Emptystring)
   /* String0 generates strings with indexes starting at position 0 */
   literals  Emptystring;
   operators
      MkString   :  Itemsort  -> String0;           /* make a string from an item */
      Length     :  String0      -> package Predefined Integer;
                                                    /* length of string              */
      First:     :  String0          -> Itemsort;   /* first item in string          */
      Last       :  String0          -> Itemsort;   /* last item in string           */
      "//"       :  String0,  String0       -> String0;   /* concatenation           */
      Extract!   :  String0,  package Predefined Integer
                                     -> Itemsort;   /* get item from string          */
      Modify!   : String0,   package Predefined Integer, Itemsort
                                     -> String0;    /* modify value of string        */
      SubString:  String0,  package Predefined Integer,
            package Predefined Integer
                                     -> String0;    /* get substring from string    */
      /*  substring (s,i,j) gives a string of length j
            starting from the ith element */

axioms
   for all item,itemi,itemj,item1,item2 in Itemsort (
   for all s,s1,s2,s3 in String0 (
   for all i,j in package Predefined Integer (
   /* constructors are Emptystring, MkString, and "//" */
   /* equalities between constructor terms */
   s // Emptystring    = = s;
   Emptystring // s    = = s;
   (s1 // s2) // s3       = = s1 // (s2 // s3);

   /* definition of Length by applying it to all constructors */
   type String Length(Emptystring)          = = 0;
   type String Length(MkString(item))      = = 1;
   type String Length(s1 // s2)                  = = Length(s1)  +  Length(s2);

   /* definition of Extract! by applying it to all constructors,
      Error! cases handled separately */
   Extract!(MkString(item),1)                         = = item;
         i <= Length(s1)  = = > Extract!(s1 // s2,i)           = = Extract!(s1,i);
         i > Length(s1)  = = > Extract!(s1 // s2,i)            = = Extract!(s2,i-Length(s1));
         i<=0 or i>Length(s)  = = > Extract!(s,i)             = = Error!;

   /* definition of First and Last by other operations */
   First(s)        = = Extract!(s,0);
   Last(s)         = = Extract!(s,Length(s)-1);
```

Superseded by a more recent version

```
    /* definition of Substring(s,i,j) by induction on j,
       Error! cases handled separately */
            i>= 0 and i<Length(s)                    = = >     Substring(s,i,0) = = Emptystring;
    i>= 0 and j>0 and i+j<Length(s)     = =>
       Substring(s,i,j) = = Substring(s,i,j-1) //
                                                  MkString(Extract!(s,i+j-1));
            i<0 or j<0 or i+j>= Length(s)          = = >     Substring(s,i,j) = = Error!;

    /* definition of Modify! by other operations */
    Modify!(s,i,item) = = Substring(s,1,i-1) // MkString(item) //
    Substring(s,i+1,Length(s)-i);)));
endgenerator String0;

/*  Usage */

/*
```

A string0 generator can be used to define strings of any item sort. The difference with the generator String (defined in Z.100) is the position '0' of the first element. ASN.1 bit strings start from position 0 according to Recommendation X.680.

The Extract! and Modify! operators will normally be used by means of shorthands notations defined in 5.3.3.4/Z.100 and 5.4.3.1/Z.100 for accessing the values of strings and assigning values to strings.
*/

```
/*  Charstring: defined in Annex D/Z.100 */

/*  Definition of ASN.1 character strings          */

syntype
    IA5String  =  Charstring
endsyntype;

syntype
    NumericString  =  Charstring  (from ("0123456789 "))
endsyntype;

syntype
    PrintableString  =  Charstring
                              (from ("A".."Z"|"a".."z"|"0".."9"|"'()+,-./:=?"))

endsyntype;

syntype
    VisibleString  =  Charstring
                                (from ("A".."Z"|"a".."z"|"0".."9"|"'()+,-./:=? "))

endsyntype;

/* Not all value notations for GraphicString are supported */
newtype GraphicString
    inherits Charstring;
endnewtype GraphicString;

/* Not all value notations for UniversalString are supported */
newtype UniversalString
    inherits Charstring;
endnewtype GraphicString;

/*      Usage      */

/*
```

These sorts define the ASN.1 character strings. Note that only characters from the International Alphabet No. 5 can be used because the character set of SDL is restricted.
*/

```
/*      Integer sort: defined in Annex D/Z.100 */

/*      Usage                    */

/*
```

ASN.1 INTEGER definitions can declare additional constants:

INTEGER { name-1(number-1), ... ,name-n(number-n) }

Such constants are defined using synonyms:

```
synonym  name_1 Integer  =  number_1,
              ...
     name_n Integer  =  number_n;
*/

/*   Natural: defined in Annex D/Z.100 */

/*   Enumeration sort */

/*   Definition    */

newtype  Enumeration
     operators
        Pred        : Enumeration -> Enumeration;
        Succ        : Enumeration -> Enumeration;
        First       : Enumeration -> Enumeration;
        Last        : Enumeration -> Enumeration;
        Num         : Enumeration -> Integer;
        "<"         : Enumeration, Enumeration -> Boolean;
        "<="        : Enumeration, Enumeration -> Boolean;
        ">"         : Enumeration, Enumeration -> Boolean;
        ">="        : Enumeration, Enumeration -> Boolean;
     axioms
        for all  e1,e2  in  Enumeration (
              e1 <       e2    = =  Num(e1) <        Num(e2);
              e1 <=      e2    = =  Num(e1) < =      Num(e2);
              e1 >       e2    = =  Num(e1) >        Num(e2);
              e1 >=      e2    = =  Num(e1) < =      Num(e2);
        )
endnewtype Enumeration;

/* Real:  defined in Annex D/Z.100, but extended in this
        recommendation with operator Power, as defined in 4.4.2.4,
        and two additional ASN.1 values, which are represented
        by external synonyms:

*/

synonym  PLUS_INFINITY         Real  =  external,
            MINUS_INFINITY        Real  =  external;

/*      Usage      */

/*
```

The real sort is used to represent real numbers known from SDL as well as ASN.1. The ASN.1 values notations { mantissa, base, exponent } is translated to

mantissa * Power (base, exponent)

```
*/

/* Array generator:  defined in Annex D/Z.100  */

/* Powerset generator:  defined in Annex D/Z.100  */

/* Bag generator  */

/*   Definition    */
generator Bag ( type  Itemsort )
     literals Empty;
     operators
        Incl        : Itemsort, Bag     -> Bag;
        Del         : Itemsort, Bag     -> Bag;
        Length      : Bag               -> Integer;
        Take        : Bag               -> Itemsort;
        Makebag     : Itemsort          -> Bag;
        "in"        : Itemsort, Bag     -> Boolean;
        "<"         : Bag, Bag          -> Boolean;
        ">"         : Bag, Bag          -> Boolean;
        "<="        : Bag, Bag          -> Boolean;
        ">="        : Bag, Bag          -> Boolean;
```

```
      "and"     : Bag, Bag          -> Bag;
      "or"      : Bag, Bag          -> Bag;
      noequality;
    axioms
      for all  i,j  in  Itemsort (
      for all  bag, b1, b2, b3  in  Bag (
      /*  constructors are Empty and or */

      /* definition of "or" by applying it to all constructors */
        Empty  or  bag      = =  bag;
        bag  or  Empty      = =  bag;
        (b1  or  b2)  or  b3    = =  b1  or  (b2  or  b3);
        bag  or  Makebag(i)  or  b1  or  Makebag(i)  or  b2
                    = =  bag  or  Makebag(i)  or  Makebag(i)  or  b1  or  b2;

      /* definition of Incl by applying to Makebag */
        Incl(i, bag)  = =  Makebag(i)  or  bag;

      /* definition of Length */
        Length(type  bag Empty)  = =  0;
        i  in  bag  = =  True  = = > Length(bag)  = =  1  +  Length(Del(i,bag));

      /* definition of Take */
        Take(Empty)  = =  Error!;
        Take(Makebag(i)  or  bag)  = =  i;

      /* definition of "in" */
        i  in type  Bag Empty       = =  FALSE;
        i  in  Incl(j,bag)          = =  i  =  j  or  i  in  bag;

      /* definition of Del */
        type Bag Del(i,Empty)       = =  Empty;
        Del(i,Incl(i,bag))          = =  bag;
            i/ = j = = > Del(i,Incl(j,bag))  = =  Incl(j,Del(i,bag));

      /* definition of "<" */
        bag<type Bag Empty              = =  FALSE;
        type Bag Empty<Incl(i,bag)      = =  TRUE;
        Incl(i,b1) < b2                 = =  i  in  b2  and  b1 < Del(i,b2);

      /* definition of ">" by other operations */
        b1 > b2  = =  b2 < b1;

      /* definition of "=" and "/=" by other operations */
      /* Note that b1  =  b2 does not imply b1  = =  b2! */
        b1  =  b2  = =  b2  =  b1;
        b1 = b1  = =  True;
        Empty  =  Incl(i,bag)           = =  False;
        (Makebag(i)  or  b1)  =  b2     = =  i  in  b2  and  b1  =  Del(i,b2);
        b1 / = b2                       = =  not  b1  =  b2;

      /* definition of "< = " and "> = " by other operations */
        b1 < = b2  =  b1 < b2  or  b1  =  b2;
        b1 > = b2  =  b1 > b2  or  b1  =  b2;

      /* definition of "and" */
        Empty  and  bag    = =  Empty;
            i  in  b2          = = > Incl(i,b1)  and  b2 = = Incl(i,b1  and  Del(i,b2));
            not(i  in  b2)     = = > Incl(i,b1)  and  b2 = = b1  and  b2;
      ));
endgenerator  Bag;

/*   Usage        */

/*
```

Bags are used to represent multi-sets. For example:

```
newtype Boolset Bag(Boolean) endnewtype Boolset;
```

can be used for a variable which can be empty or contain (True), (False), (True,False) (True,True), (False,False),... .
Bags are used to represent the SET OF construction of ASN.1.
*/

```
/*	PId: defined in Annex D/Z.100 */

/*	Duration: defined in Annex D/Z.100 */

/*	Time: defined in Annex D/Z.100 */

/*	ASN.1 Bit sorts	*/

/*	Definition	*/

newtype Bit
    inherits Boolean
      literals 0 = FALSE, 1 = TRUE;
      operators all ;
endnewtype Bit;

newtype Bit_String String0 (Bit, ''B)
    adding
      literals   nameclass ('0' or '1')*'B',
                 nameclass ( ('0':'9') or ('A':'F') )*'H';
      operators
      "not"     : Bit_String            ->    Bit_String;
      "and"     : Bit_String,Bit_String ->    Bit_String;
      "or"      : Bit_String,Bit_String ->    Bit_String;
      "xor"     : Bit_String,Bit_String ->    Bit_String;
      "=>"      : Bit_String,Bit_String ->    Bit_String;
      noequality;
      axioms
      '0000'B == '0'H;  '0001'B == '1'H;   '0010'B == '2'H;  '0011'B == '3'H;
      '0100'B == '4'H;  '0101'B == '5'H;   '0110'B == '6'H;  '0111'B == '7'H;
      '1000'B == '8'H;  '1001'B == '9'H;   '1010'B == 'A'H;  '1011'B == 'B'H;
      '1100'B == 'C'H;  '1101'B == 'D'H;   '1110'B == 'E'H;  '1111'B == 'F'H;
      /* for use of apostrophes in literals, see 4.4.3 */
      MkString(0) == '0'B; MkString(1) == '1'B;
      for all s, s1, s2, s3 in Bit_String (
        s  =    s    ==  True;
        s1 =    s2   ==  s2 = s1;
        s1 /=   s2   ==  not ( s1=s2 );
        s1 =    s2   ==  True ==> s1 == s2;
        ((s1 = s2)  and (s2 = s3))   ==> s1 = S3 == True;
        ((s1 = s2)  and (s2 /= s3))  ==> s1 = S3 == False;
        for all b, b1, b2 in Bit (
          not(''B)                   == ''B;
          not(MkString(b) // s)      == MkString( not(b) ) // not(s);

      /* the length of adding two strings is the maximal length
        of both strings */
        ''B and ''B == ''B;
        Length(s) > 0 ==> ''B and s == MkString(0)   and s;
        Length(s) > 0 ==> s and ''B == s             and MkString(0);
        (MkString(b1) // s1) and (MkString(b2) // s2) ==
        MkString(b1 and b2) // ( s1 and s2 );

      /* definition of remaining operators based on "and" and "not" */
        s1 or    s2   == not (not s1 and not s2);
        s1 xor   s2   == (s1 or s2) and not(s1 and s2);
        s1 =>    s2   == not (not s1 and s2);
```

```
    ));
    map
    for all b1,b2,b3,h1,h2,h3 in Bit_String literals (
      for all bs1, bs2, bs3, hs1, hs2, hs3 in Charstring literals (
        Spelling(b1) = "" // bs1 // "" // 'B',
        Spelling(b2) = "" // bs2 // "" // 'B',
        bs1 /= bs2
          ==> b1 = b2 = False;
        Spelling(h1) = "" // hs1 // "" // 'H',
        Spelling(h2) = "" // hs2 // "" // 'H',
        hs1 /= hs2
          ==> h1 = h2 = False;
        Spelling(b1) = "" // bs1 // "" // 'B',
        Spelling(b2) = "" // bs2 // "" // 'B',
        Spelling(b3) = "" // bs3 // "" // 'B',
        Spelling(h1) = "" // hs1 // "" // 'H',
        Spelling(h2) = "" // hs2 // "" // 'H',
        Spelling(h3) = "" // hs3 // "" // 'H',
        Length(bs1) = 4,
        Length(hs1) = 1,
        Length(hs2) > 0,
        Length(bs2) = 4 * Length(hs2),
        h1 = b1
          ==> h3 = b3 == h2 = b2;
        /* connection to the String generator */
        for all b in Bit literals (
        Spelling(b1) = "" // bs1 // bs2 // "" // 'B',
        Spelling(b2) = "" // bs2 // "" // 'B',
        Spelling(b) = bs1,
              ==> b1 == MkString(b) // b2;
    )));
endnewtype Bit_String;

/*  Usage */
/* ASN.1 bit strings are represented by Bit_String.
    named bits can be provided by synonyms:
BIT STRING { name-1(number-1), ... ,name-n(number-n) }

The named bits can be represented by:

synonym     name_1 Integer == number _1,
            ...
            name_n Integer = number_n;
*/

/* ASN.1 Octet sorts */

/* Definition     */

syntype Octet = Bit_String constants size (8)
endsyntype Octet;

newtype Octet_String String (Octet,''B)
    adding
      literals  nameclass (('0' or '1')8)*'B',
                nameclass ((('0':'9') or ('A':'F'))2)*'H';
      operators
      Bit_String        : Octet_String      -> Bit_String;
      Octet_String   : Bit_String      -> Octet_String;
      noequality;
      axioms
      for all b,b1,b2 in Bit_String (
        for all s in Octet_String (
          for all o in Octet(
          Bit_String(''B) == ''B;
          Bit_String( MkString(o) // s ) == o // Bit_String(s);
```

```
        Octet_String(''B) == ''B;
        Length(b1) > 0, Length(b1) < 8, b2 == b1 or '00000000'B
            ==> Octet_String(b1) == MkString(b2);
        b == b1 // b2, Length(b1) == 8
            ==> Octet_String(b) == MkString(b1) // Octet_String(b2);
    )));
    map
    for all o1, o2 in Octet_String literals (
        for all b1, b2 in Bit_String literals (
        Spelling( o1 ) = Spelling( b1 ),
        Spelling( o2 ) = Spelling( b2 )
         ==> o1 = o1 == b1 = b2
    ));

endnewtype Octet_String;
```

/* Usage */

/* ASN.1 octet strings are represented by Octet_String */

/* ASN.1 Null sort */

/* Definition */

```
newtype Null
  literals Null;
endnewtype Null;
```

/* Usage */

/* This type can be used to transmit a presence information.
The declaration of a variable of sort Null is not useful */

/* ASN.1 Object Identifier sort */

/* Definition */

```
newtype object_element
  literals nameclass ('0':'9')+;
endnewtype object_element;

newtype Object_Identifier String (object_element, EmptyString )
endnewtype Object_Identifier;
```

/* Usage */

/*
Object identifier values should be supported according the ASN.1 rules. The use of NamedForm's is provided by synonyms. There is no reference to the world-wide information structure in that definition, i.e. this additional semantics is out of scope of SDL in combination with ASN.1.

Note that **synonym** ccitt object_element = 0;
implies that (. ccitt, ccitt, ccitt .) is a valid value in SDL.
*/

/* ASN.1 ANY sort */

/* Definition */

```
newtype Any_type
endnewtype Any_type;
```

/* Usage */

/*
The ASN.1 type ANY is mapped to the sort Any_type. Note that values of this sort are incompatible to other values, i.e. only assignments and application of '=', '/=' are possible. For example it is possible to handle unknown contents of signals:

```
dcl  a1, a2 Any_type;
…
input (Signal_with_ANY(a1));
decision  ( a1 /= a2 );
    (True) : output (Signal_with_ANY(a2));
    else    : ;
enddecision;

*/

/*   ASN.1 Useful types */

GeneralizedTime      ::=    VisibleString;

UTCTime              ::=    VisibleString;

EXTERNAL_Type      ::= sequence {
            direct_reference              Object_Identifier    optional,
            indirect_reference             Integer             optional,
            date_value_descriptor         ObjectDescriptor     optional,
            encoding                choice {
               single_ASN1_type       Any_type,
               octet_aligned              Octet_String,
               arbitrary                  Bit_String
                                          }
                          };

ObjectDescriptor::=  GraphicString;

endpackage Predefined;
```

Appendix I

Restrictions on ASN.1 and SDL

(This appendix does not form an integral part of this Recommendation)

I.1 Summary of the supported subset of ASN.1

This subclause contains a summary of the subset of ASN.1 that is supported in this Recommendation.

In principle, the version of ASN.1 as defined in Recommendation X.680 is supported. The features of Recommendations X.681, X.682, and X.683 are not supported, i.e. information object specification, parameterisation of ASN.1 specifications and constraint specification are not supported. In addition, the ANY type that is defined in Recommendation X.208 is partly supported.

The main restrictions on ASN.1 are:

- Macros are not supported. This means that the operation macro is also not supported. In SDL, the operation macro is not needed, because there are mechanisms that are more suitable for the definition of operations, namely signal exchange or remote procedures. An example of replacing the operation macro by SDL constructs is given in Appendix II.
- Tags are allowed to be used, but they have no meaning: they are ignored. The motivation is that tags are only used for the purpose of encoding, and encoding is outside the scope of SDL.
- Encoding rules are outside the scope of this Recommendation. It is possible that tool vendors will support encoding rules, but this is implementation freedom.
- No value notation for ANY (or ANY DEFINED BY) is supported. By introducing special operators, a limited form of value notation for ANY can be imitated, as shown in 4.4.1. ANY can in some cases be replaced by a CHOICE over the types for which value notations are needed. Appendix II gives an example of this.

- The dash in ASN.1 names is not supported inside SDL descriptions. It is allowed to use dashes in names within ASN.1 modules that are imported in SDL, but when they are imported in SDL, the dashes should be transformed to underscores. The motivation for this restriction is that the dash is considered as the minus operator in SDL.

- Case sensitivity is not supported. The motivation for this restriction is that SDL is case insensitive.

  The restriction implies that introducing two types with the same name (apart from case sensitivity) is an error. However, it is allowed to have the same name if they are of different entity classes. Entity classes are for example type names, value names and identifiers, i.e.:

  SameName :: = INTEGER {sameName (0)}

  sameName SameName : = sameName

  is allowed.

- Definitions must be ended with a semicolon. The motivation is that SDL definitions are ended by a semicolon. If the semicolon was not mandatory, the grammar would be ambiguous, because SDL is case insensitive.

- ASN.1 type REAL is not represented as a sequence of three integers, as is the case in Recommendation X.680. The value notation of Recommendation X.208 shall be used instead, i.e. { 314, 10, –2 } should be used instead of { mantisse 314, base 10, exponent –2 }. Alternatively, the SDL syntax for denoting REAL values can be used, i.e. 3.14 is also allowed. As a consequence, no subtyping of mantisse, base, or exponent is allowed, and no operators for accessing or changing the mantisse, base, or exponent of a REAL value are supported.

- The use of the same identifier for named numbers or named bits of different types in the same scope shall be avoided. The motivation is that named numbers and named bits are mapped on SDL integer synonyms. Using the same identifier twice would result in illegal SDL (redefinition of the same synonym).

  In other words, the double use of 'notAllowed' in the below type definitions is not allowed:

  Int1 ::= INTEGER { notAllowed(0) }

  Int2 ::= INTEGER { notAllowed(0) }

  Neither is the following:

Int	::= INTEGER	{ notAllowed(0) }
BitString	::= BIT STRING	{ notAllowed(5) }

Double use of the same identifier in different enumerated types, or in an enumerated type and in a named integer or named bit is allowed, because the identifiers in enumerated types are not mapped on integer synonyms. In other words, the following is allowed:

Enum1	::= ENUMERATED	{ allowed(0) }
Enum2	::= ENUMERATED	{ allowed(1) }
BitString	::= BIT STRING	{ allowed(5) }

- In external type and value references, spaces shall be put around the ".", e.g. Modulereference . Typereference instead of Modulereference.Typereference. Leaving out spaces would give syntactical problems in SDL, because identifiers in SDL may contain dots.

- The OBJECT IDENTIFIER component values that are assigned by ITU-T, ISO, or both, are not defined in the package Predefined, because they cannot be maintained in this Recommendation. For example, in order to use:

  { ccitt recommendation z }

  the component values CCITT, recommendation, and z have to be defined in a user defined package.

I.2 Summary of the supported subset of SDL

This subclause contains a summary of the subset of SDL that is supported in this Recommendation.

Full use of SDL is allowed, with the following exceptions:

- The use of the characters {, }, [,], and | (vertical bar) in names is not allowed. This is because they have a special meaning in ASN.1.
- This Recommendation introduces several new keywords that cannot be used as names. These keywords are listed in clause 3.
- A dot in a name must not be followed by an underline or another dot. An underline in a name must not be followed by a dot. For example, the following names are not allowed:

  invalid..name, invalid._name, invalid_.name

- Use of double negation (e.g. 1 -- 2) is not supported. The reason is that -- is used for ASN.1 comments.
- Separators and comments are not allowed inside the definition of quoted operators, because this would cause syntactic ambiguity. For example, the following is not allowed:

  operators

  "+ /* infix operator*/": MyType, MyType -> MyType

- An underline in a name contained in a composite primary must be specified explicitly, e.g. for the following type definition:

  T ::= SEQUENCE
  {a_b INTEGER},

  the following value notation is not allowed:

  tT ::= {a b 5}

  Instead,

  tT ::= {a_b 5} must be used.

Appendix II

Examples

(This appendix does not form an integral part of this Recommendation)

This appendix contains examples of the use of ASN.1 in combination with SDL. In this appendix, it is assumed that the reader is familiar with ASN.1 and with SDL. Only the graphical representation of SDL is used.

The purpose of this appendix is to show the principles for using SDL in combination with ASN.1. The general topics that are treated are:

- Defining ASN.1 data types in SDL diagrams;
- Defining ASN.1 values in SDL diagrams;
- Using the operators on ASN.1 data types that are defined in this Recommendation;
- Circumventing some restrictions on ASN.1 that are imposed by this Recommendation;
- A 'style guide' for using ASN.1 in combination with SDL.

The intention is not to give a complete overview of all operators on all ASN.1 data types. These can be found elsewhere in this Recommendation.

II.1 Defining ASN.1 data types in SDL

There are three ways to define ASN.1 data types in SDL:

1) By importing a type definition from an ASN.1 module;
2) By defining the type directly in SDL, using ASN.1 syntax;
3) By defining the type directly in SDL, inside an SDL newtype clause.

II.1.1 Importing a type definition from an ASN.1 module

An ASN.1 type that is defined in an ASN.1 module can be imported in SDL using the IMPORTS construct, which can be put in an SDL text symbol.

The types Address, DisconReason, and DisconIndParameters are defined in the ASN.1 module ProtocolData that is shown below:

```
ProtocolData {ccitt recommendation z 105 examples(0) 1}  DEFINITIONS ::=

Address ::= GraphicString;

DisconReason ::= ENUMERATED {
        outOfOrder(0), fatalError(1), otherReason(2) };

DisconIndParameters ::=     SEQUENCE {
        origin              Address,
        destination         Address,
        reason              DisconReason };

END
```

These types can be imported in SDL as shown in Figure II.1.1.1.

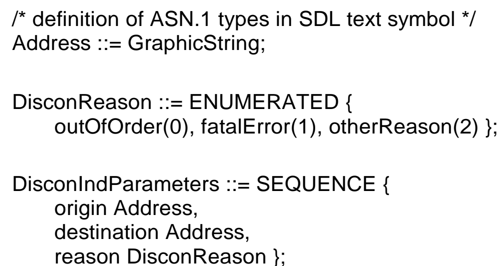
T1008150-95/d01

FIGURE II.1.1.1/Z.105

Import of ASN.1 types in SDL

II.1.2 Defining ASN.1 types directly in SDL

An ASN.1 type can be defined directly in SDL (i.e. without being imported from an ASN.1 module), using normal ASN.1 syntax. The only difference is that a semicolon has to be put at the end of a type definition. The definitions of new data types are put in text symbols. This is illustrated in Figure II.1.2.1 below:



T1008160-95/d02

FIGURE II.1.2.1/Z.105

Defining ASN.1 types directly in SDL

Also variables can be declared to be of an ASN.1 type, as illustrated in Figure II.1.2.2.

```
/* declaration of variables */
DCL
DIdata DisconIndParameters,
index INTEGER,
cube_frequency SEQUENCE (SIZE(6)) OF INTEGER;
```

T1008170-95/d03

FIGURE II.1.2.2/Z.105

Declaration of variables of ASN.1 types

II.1.3 User defined operators on ASN.1 types

It is possible to define ASN.1 types in SDL and add user defined operators to this type. For this purpose, the ASN.1 type can be defined directly in SDL inside a *newtype* clause.

As an example, a type Rectangle is taken that contains information about the length and the width of a rectangle. This ASN.1 type could be defined as:

```
Rectangle ::= SEQUENCE {
        length      INTEGER,
        width       INTEGER };
```

It is possible to define in SDL an operator 'area', that returns the area (length times width) of the rectangle. Two ways will be illustrated to do this:

1) *Introduction of a new type that inherits from Rectangle*

   If the type Rectangle is already defined somewhere (for example in an ASN.1 module), then a new type can be introduced that inherits Rectangle, and adds a new operator. This is shown if Figure II.1.3.1, where type Rectangle2 inherits Rectangle, but has additionally an operator. The operator can only be used on values of type Rectangle2, but it can not be used on values of type Rectangle.

   Note that in the axiom, Make!(len, wid) is used, instead of the ASN.1 value notation {length len, width wid}.

```
- - definition of Rectangle
- - without any operators


     Rectangle ::= SEQUENCE {
          length INTEGER,
          width INTEGER };

- - new type Rectangle2 is the
- - same as Rectangle, except that
- - it provides an operator

NEWTYPE Rectangle2
  INHERITS Rectangle
  ADDING
  OPERATORS
    area: Rectangle -> INTEGER;
  AXIOMS
    for all len, wid in INTEGER (
       area (Make! (len, wid)) == len * wid;
    )
ENDNEWTYPE Rectangle2;
```

T1008180-95/d04

FIGURE II.1.3.1/Z.105

Inheriting an existing type and adding an operator

2) *Defining Rectangle + the operator from scratch*

If Rectangle is not yet defined, it is possible to define the type and the operator in one type definition. For this purpose, Rectangle is defined within the SDL newtype construct, and the operator is added after the definition of the fields, as shown in Figure II.1.3.2.

```
NEWTYPE Rectangle
  SEQUENCE {
    length INTEGER,
    width INTEGER };
  OPERATORS
    area: Rectangle -> INTEGER;
  AXIOMS
    for all len, wid in INTEGER (
       area (Make! (len, wid)) == len * wid;
    )
ENDNEWTYPE Rectangle;
```

T1008190-95/d05

FIGURE II.1.3.2/Z.105

Defining operators on ASN.1 types

II.2 Using ASN.1 values in SDL

ASN.1 values can be used in SDL in expressions. Expressions are used at a number of places in SDL, for example:

- in an assignment to a variable;
- in decision symbols;
- in the parameters of an output of a signal, procedure call or process creation;
- in the definition of constants.

The simplest way to use ASN.1 values is by using the normal ASN.1 value notation. Figure II.2.1 shows the use of an ASN.1 value in an output of SDL signal DisconInd. It is assumed that this signal carries as parameter the ASN.1 type DisconIndParameters that is defined in II.1.1.

```
DisconInd
({origin "215.87.43.12",
destination "52.91.160.202",
reason fatalError})
```

T1008200-95/d06

FIGURE II.2.1/Z.105

ASN.1 value in an SDL output

The ASN.1 value notation can also be used to define constants in SDL, just like defining values in ASN.1, as is shown in Figure II.2.2.

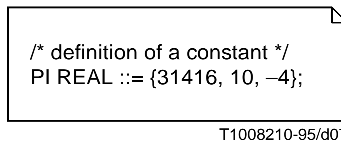


FIGURE II.2.2/Z.105

Defining a constant using the ASN.1 value notation

II.3 Illustration of the use of some ASN.1 types in SDL

ASN.1 does not have operators. In SDL, operators on ASN.1 data are needed to compare and manipulate data. In this subclause, the use of some operators on ASN.1 types is illustrated. The complete set of predefined operators and their meaning is defined in Annex A and in the main text of this recommendation.

II.3.1 Operators on simple ASN.1 types

Comparing data values

For every ASN.1 data type, at least two operators are available: = (equals) and /= (not equals). All ordered types also have operators < (less than), > (greater than), <= (less than or equal) and >= (greater than or equal).

The comparison operators are frequently combined with boolean operators (and, or, not, etc.), for example in decision symbols. Figure II.3.1.1 shows three examples of comparison operators.

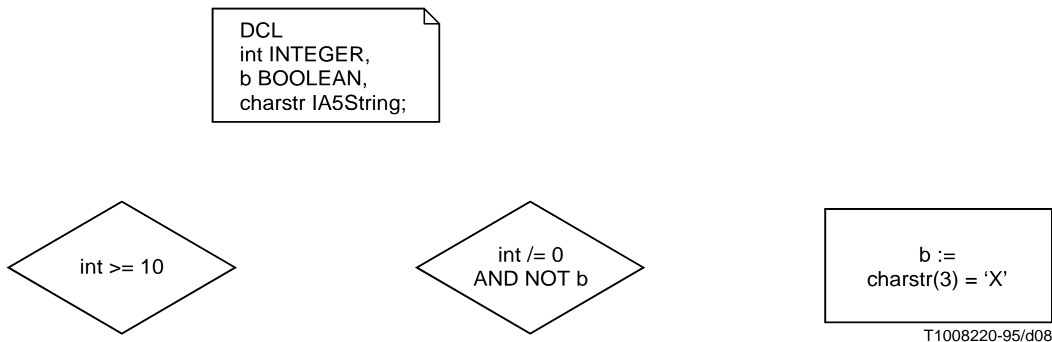


FIGURE II.3.1.1/Z.105

Comparison operators and boolean operators

Manipulating values

There are a number of operators defined on ASN.1 simple types, like +, –, *, / on integers and reals. Figure II.3.1.2 shows a few of them:

- + to add two integers (index + 1) or reals.

- // to concatenate character strings (prefix // "X" // suffix) or bit strings ('03FC'H // Mkstring(1)). This operator is available for all string types (i.e. character strings, BIT STRING, OCTET STRING, SEQUENCE OF types)

- Mkstring to make a string consisting of one element (Mkstring(1)). This operator is available for all string types.

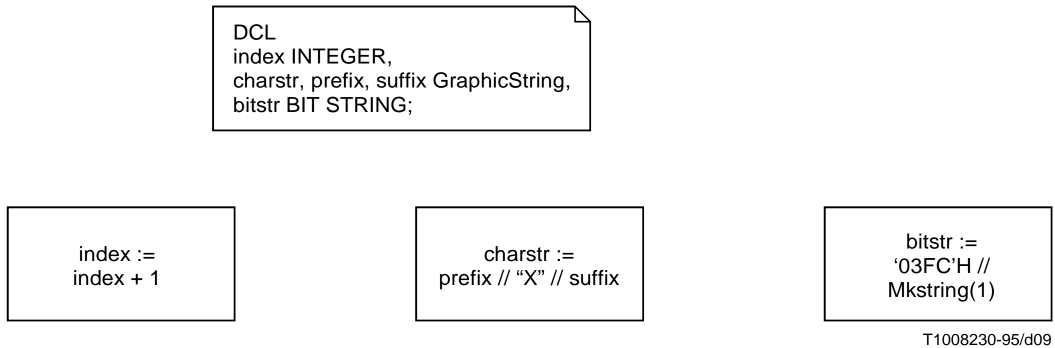


FIGURE II.3.1.2/Z.105

Examples of operators on simple types

Annex A defines all the operators on simple types.

II.3.2 SEQUENCE

For types defined with SEQUENCE, an operator is available to access the value of one component. As an example type ConReqData is taken, that is defined in the partial ASN.1 specification below. The component "Receiver" of variable v of ConReqData can be assigned a value, and accessed using v!Receiver, as illustrated in Figure II.3.2.1.

```
Class ::= ENUMERATED {
        class0(0), class1(1), class2(2) };

ConReqData ::= SEQUENCE {
        sender          GraphicString,
        receiver        GraphicString,
        class           Class DEFAULT class0,
        specialOptions  SpecialOptions OPTIONAL };
```

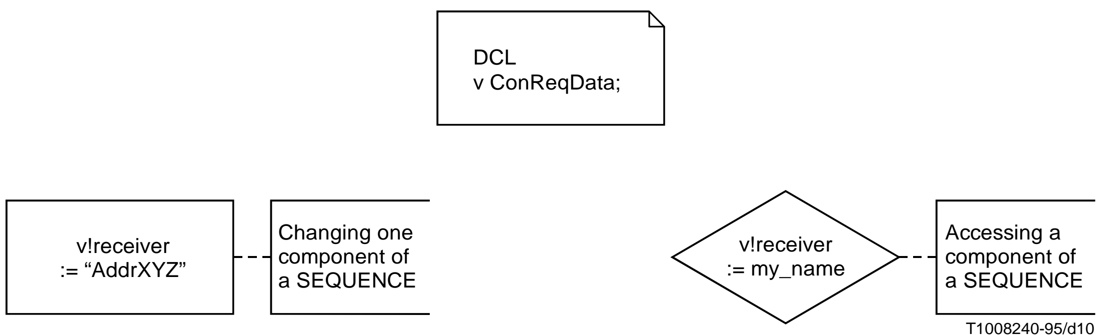


FIGURE II.3.2.1/Z.105

Accessing a component of a SEQUENCE

Reading a default component is always possible, even if its value is absent: in that case the default value is returned.

Reading an optional component requires special care: a dynamic error results if the component is absent. Before accessing an optional component it should always be investigated whether the component is present or not. For this purpose the operator '<identifier>Present' can be used, as illustrated in Figure II.3.2.2.

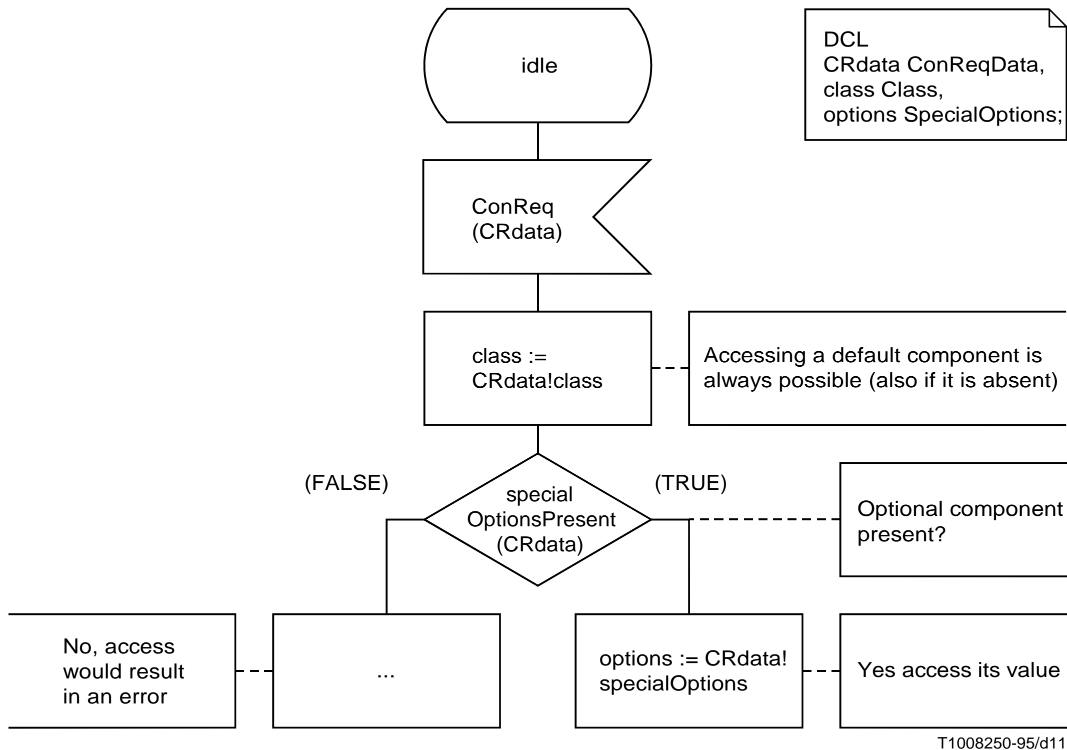


FIGURE II.3.2.2/Z.105

Accessing default and optional components

II.3.3 SEQUENCE OF

Figure II.3.3 shows an example of operators on SEQUENCE OF-types. Variable 'list' contains an ordered sequence of integers. On reception of signal 'nr', a new integer is inserted in the list.

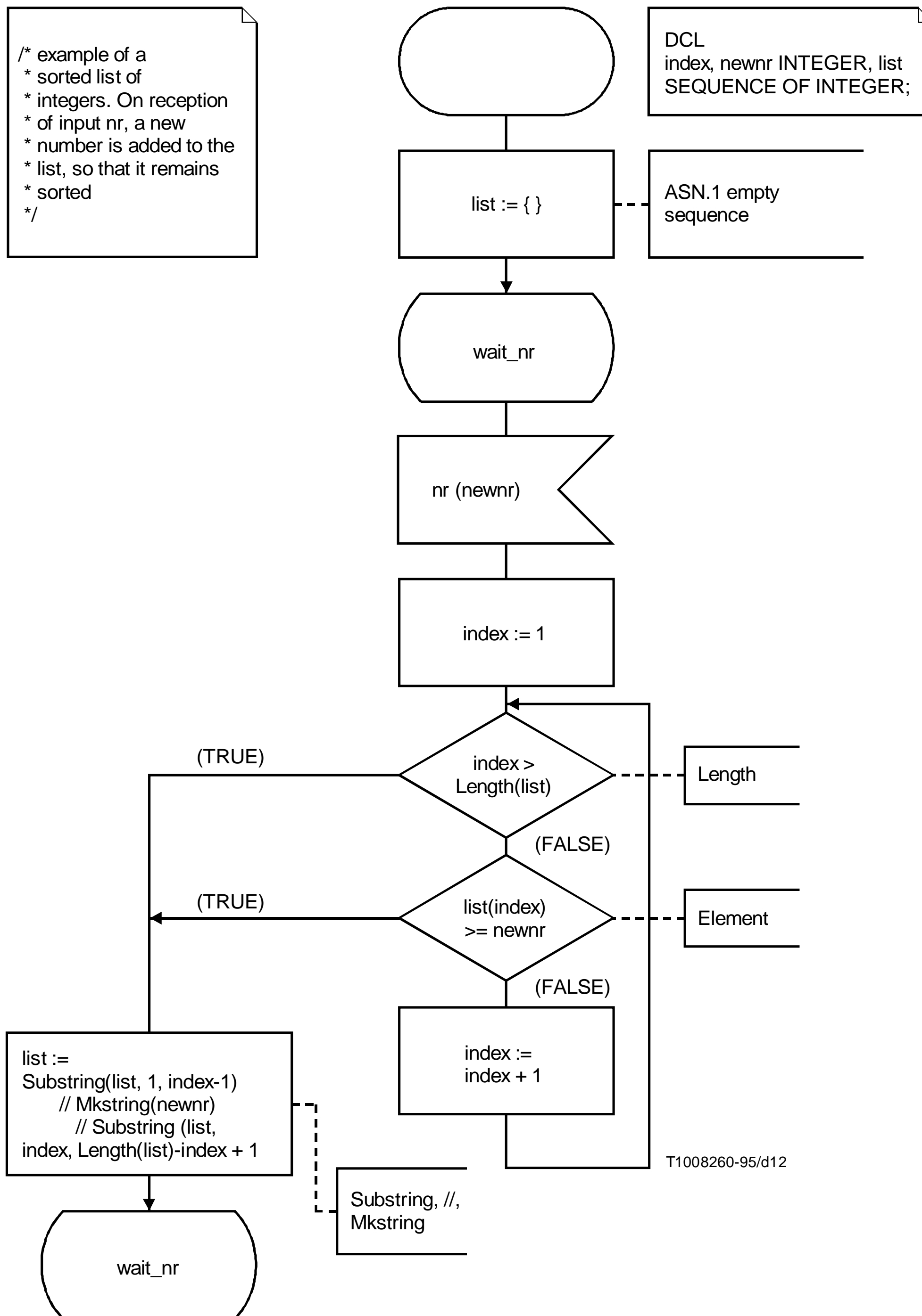


FIGURE II.3.3/Z.105

The use of the predefined operators for SEQUENCE OF

In this example, several operators on SEQUENCE OF are used:

- list(index) indexes one element of a SEQUENCE OF;
- Length(list) returns the length of a string;
- Mkstring(newnr) makes a string containing one item;
- Substring(list, 1, index-1) returns a substring of list, starting at position 1 (the first position) and with length index-1;
- ... // ... returns the concatenation of two strings.

II.3.4 SET OF

The ASN.1 SET OF type can be used to represent sets. This example illustrates the use of SET OF in modelling telephony call processing. It is assumed that a subscriber can activate several so-called supplementary services.

The data types for these supplementary services are defined with the ASN.1 types that follow below. SupplementaryService is an enumerated type that has all services as values. ServiceSet is the type that has sets of supplementary services as values.

```
SupplementaryService ::=  ENUMERATED {
                    HotLine (0), CallWaiting (1), CallForwardUnconditional (2) };

ServiceSet ::= SET OF SupplementaryService;
```

Figure II.3.4 shows a fragment of an SDL process that does the telephone call processing for one subscriber. Variable ActiveServ of type ServiceSet is used to keep track of which services are active for a given subscriber. By sending signal Activate, a service can be activated, by sending signal Deactivate, a service can be activated.

If the subscriber does offhook, it is checked whether service hotline is active. If this is the case, a predefined number is immediately called, without waiting for the subscriber to dial a number. If hotline is not active, a dialling tone is sent to the subscriber, after which the subscriber can dial a number.

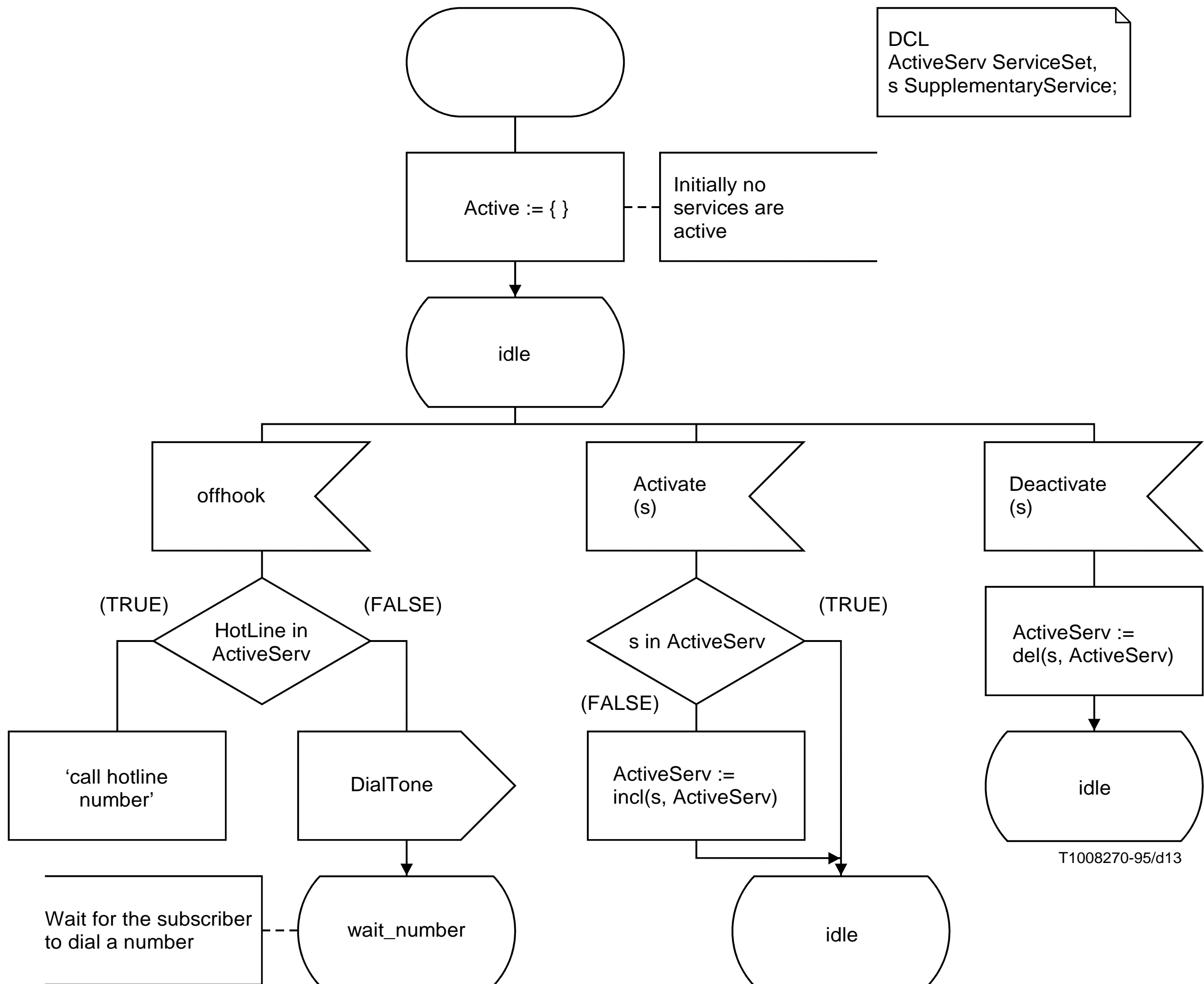


FIGURE II.3.4/Z.105

Use of predefined operators for SET OF

In the example, several operators on SET OF are illustrated:

- s in ActiveServ: gives TRUE if service s is member of set ActiveServ.
- incl (s, ActiveServ): gives the set ActiveServ + element s. Note that in ASN.1, it is significant how many times an element appears in a SET OF: { HotLine, HotLine } is not the same as { HotLine }. In the example, it makes no sense that a service is activated more than once. Therefore it is checked whether s is already member of Active before s is actually added to the set.
- del (s, ActiveServ): gives the set ActiveServ from which element s is removed.

II.3.5 CHOICE

A value can be assigned to a variable of a CHOICE type by using the ASN.1 value notation for CHOICE. Note that the identifier has to be supplied, and that a colon shall be present between the identifier and the value. A value can also be assigned to a variable using an SDL operator (<variable>!<identifier>). These two ways are illustrated in Figure II.3.5.

Accessing a non-selected component of a CHOICE value results in a dynamic error. Such errors can be avoided by using the !present-operator to find out which alternative was selected before accessing a component. This is illustrated in Figure II.3.5.

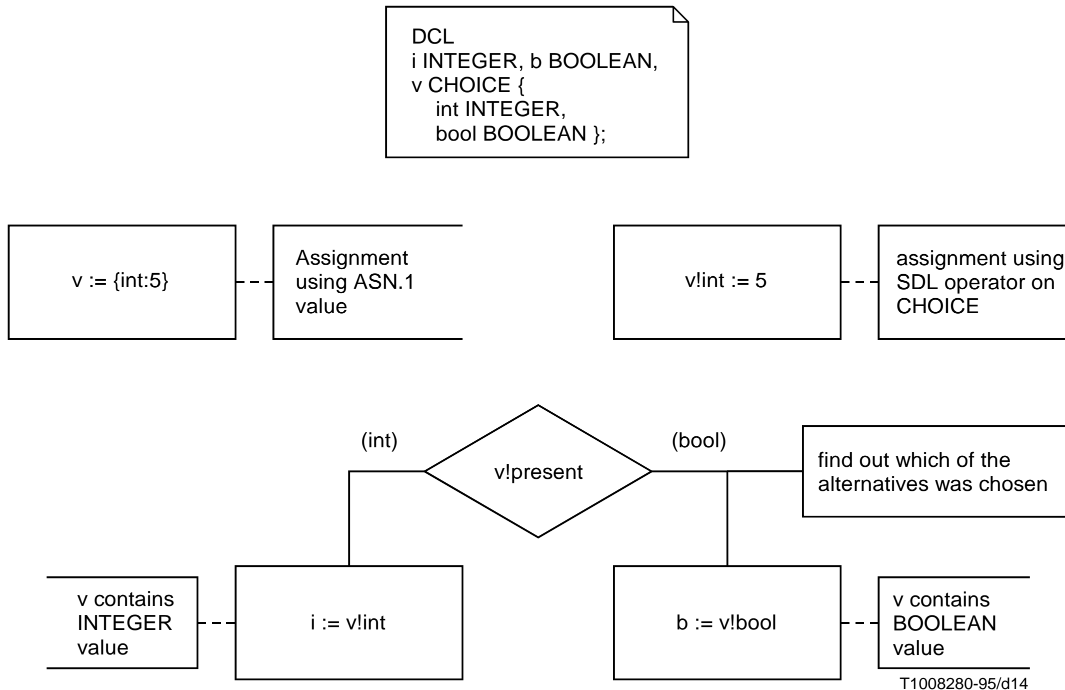


FIGURE II.3.5/Z.105

The use of CHOICE in combination with SDL

II.4 Guidelines to circumvent restrictions on ASN.1

In this subclause, some guidelines are given how to deal with some ASN.1 constructs that are not supported in combination with SDL. This subclause does not aim to give all possible solutions.

II.4.1 ANY DEFINED BY / TYPE-IDENTIFIER information object class

ANY DEFINED BY (Recommendation X.208, in Recommendation X.681 replaced by the TYPE-IDENTIFIER useful information object class) is often used to specify that the type of a parameter is dependent on the value of some other parameter, for example, suppose a process has a number of attributes, possibly of different types. An attribute is addressed by its identifier. It is possible to have a general operation to read an attribute. The type of the response of the read operation depends on the attribute that was read.

```
ReadArg ::= OBJECT IDENTIFIER;

ReadResult ::= SEQUENCE {
        attribute       OBJECT IDENTIFIER,
        result          ANY DEFINED BY attribute };
```

In this Recommendation, no operations are supported for ANY DEFINED BY, except = and /=. This makes it impossible to give a specification of the read operation in SDL.

Specification of the read operation is possible if the different attribute types are known. In that case, the ANY DEFINED BY can be replaced by a CHOICE type over the different attributes. Suppose that there are totally two attributes, one of type INTEGER, one of type BOOLEAN. The ReadResult type can then be rewritten as below:

```
ResultType ::= CHOICE {
        attr1           INTEGER,
        attr2           BOOLEAN };

ReadResult ::= SEQUENCE {
        attribute       OBJECT IDENTIFIER,
        result          ResultType };
```

Using the redefined ReadResult, the read operation can be specified in SDL as shown in Figure II.4.1.

II.4.2 The OPERATION macro / information object class

The ASN.1 macro mechanism is not supported for use in combination with SDL. Neither is the mechanism for information object specification (see Recommendation X.681). This means that the often used OPERATION information object cannot be used in combination with SDL.

In this section, two ways are shown how to model an operation in SDL:

1) by exchange of SDL signals

2) by an SDL remote procedure

As an example, a read operation is taken. Its definition in ASN.1 is:

```
Read    OPERATION
        ARGUMENT        ReadArg
        RESULT          ReadResult
        ERRORS          { noSuchAttribute, accessDenied,
                        processingFailure }
    ::=  {ccitt recommendation z 105 operations(2) 0}
```

(ReadArg and ReadResult are defined in II.4.1.)

II.4.2.1 Modelling an operation by signal exchange

The read operation can be modelled by the introduction of three different signals:

1) ReadInvoke, with parameter of type ReadArg to model the invocation of the read operation.

2) ReadResult, with parameter of type ReadResult to model the return of the result of the read operation.

3) ReadError, with parameter of type “Enumerated” for the different read errors to model the failure of the read operation.

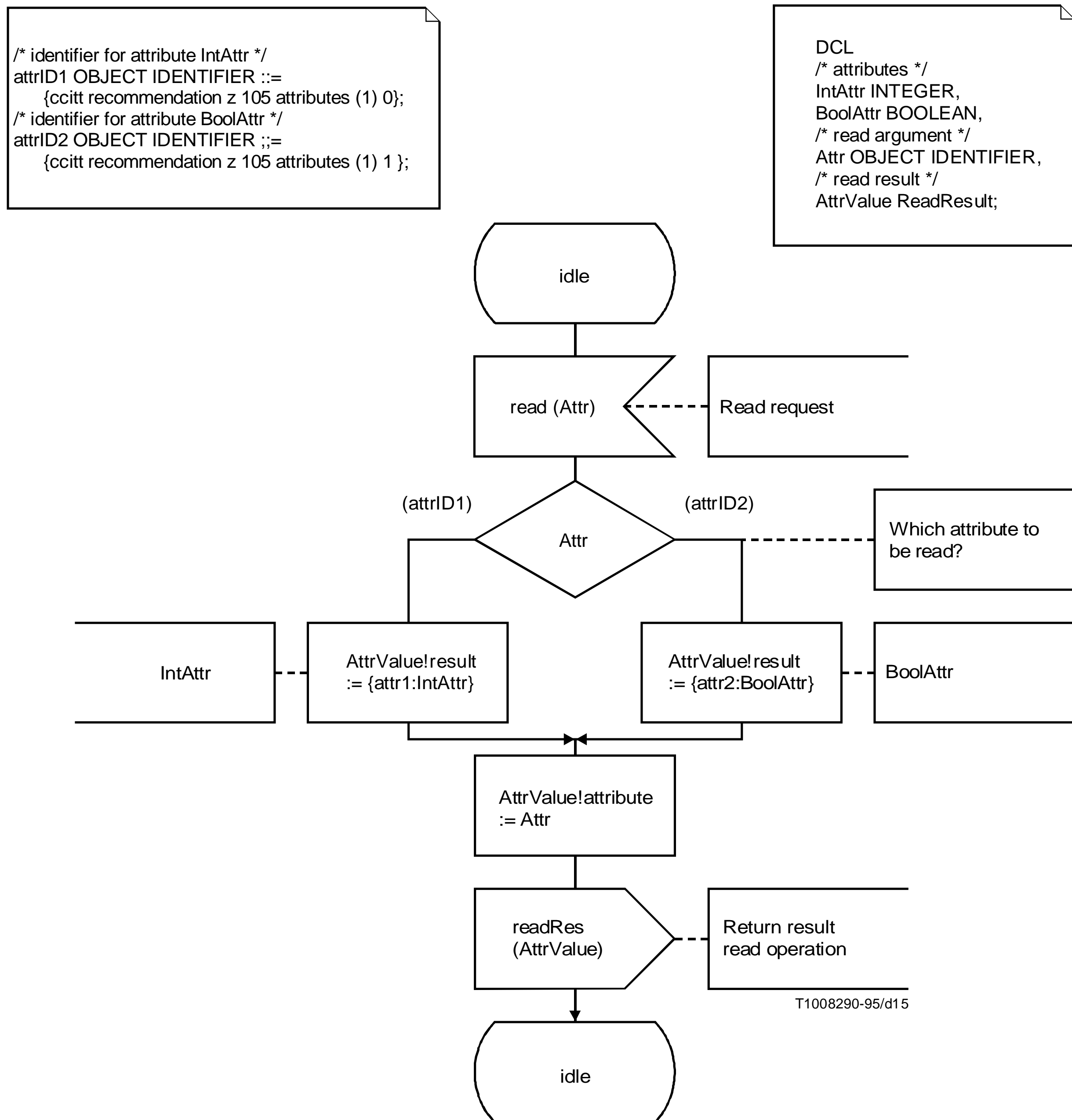


FIGURE II.4.1/Z.105

Specification of the read operation

For ReadError, an enumerated type needs to be defined for the different read errors:

```
ReadErrors ::= ENUMERATED {
        noSuchAttribute(0), accessDenied(1), processingFailure(2) };
```

The operation code does not have to be present in the SDL model: the name of the signal is used instead to identify the operation.

The actual operation can be modelled as a process transition, as is shown in Figure II.4.2.1. The diagram contains informal text. In II.4.1, it is shown how the informal text can be replaced by formal SDL.

The operation is invoked by sending signal ReadInvoke to the process that can perform the operation, and wait for the result, as illustrated in Figure II.4.2.2.

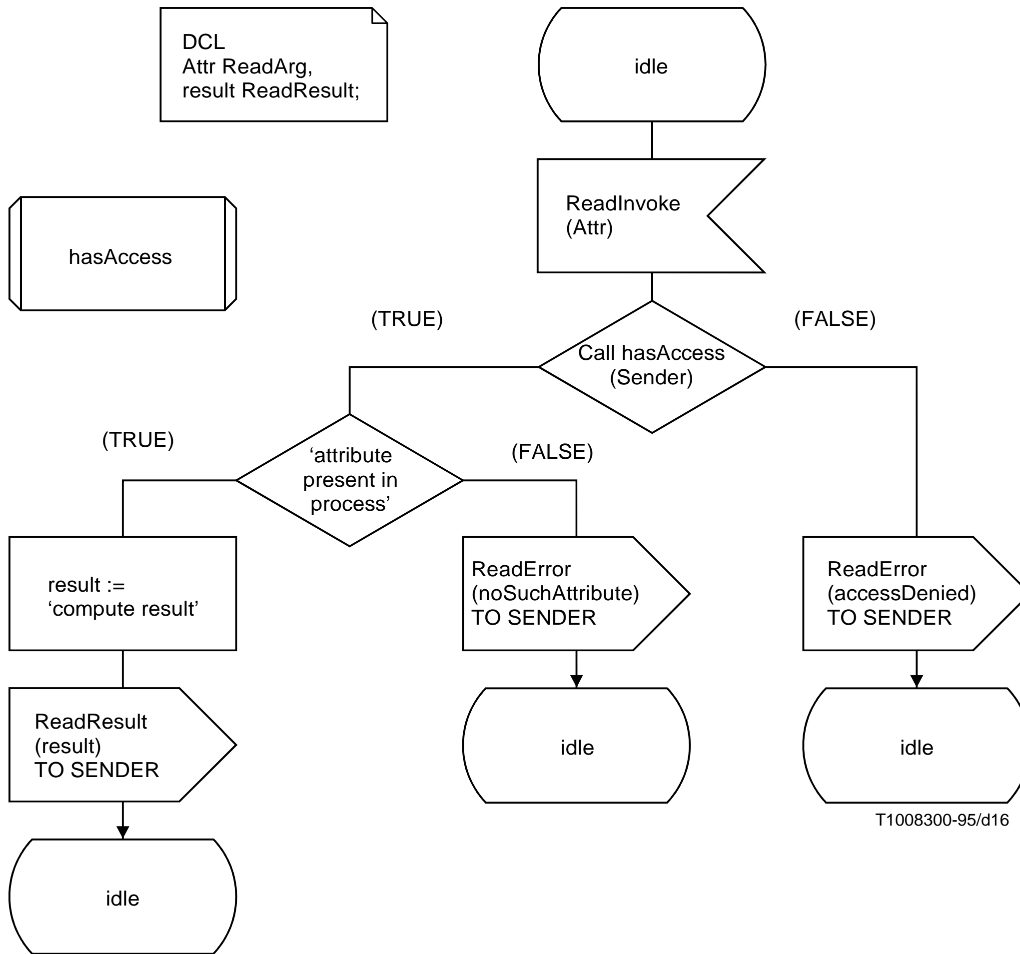


FIGURE II.4.2.1/Z.105

An SDL transition modelling the read operation

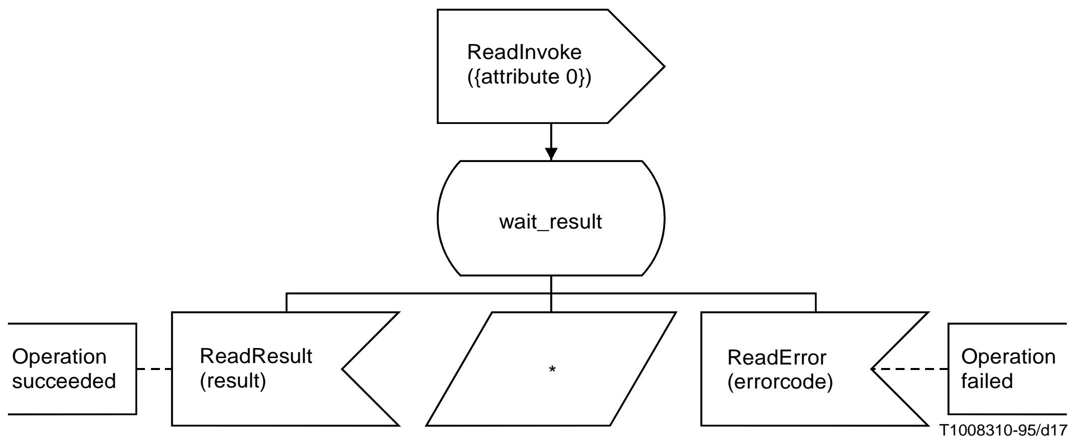


FIGURE II.4.2.2/Z.105

Invoking the read operation

II.4.2.2 Modelling an operation by a remote procedure

An alternative way is to introduce an SDL remote procedure for the read operation. The remote procedure gets two parameters:

- an **in** parameter for the argument of the read operation (type ReadArg);
- an **in/out** parameter for the result of the operation. The result can be either that the operation succeeded, in which case it is of type ReadResult, or the operation failed, in which case it is of type ReadError as defined in II.4.2.1.

For the **in/out** parameter, a new type is introduced:

```
ReadReply ::= CHOICE {
        success     ReadResult,
        failed      ReadError };
```

The procedure that models the Read operation is shown in Figure II.4.2.3.

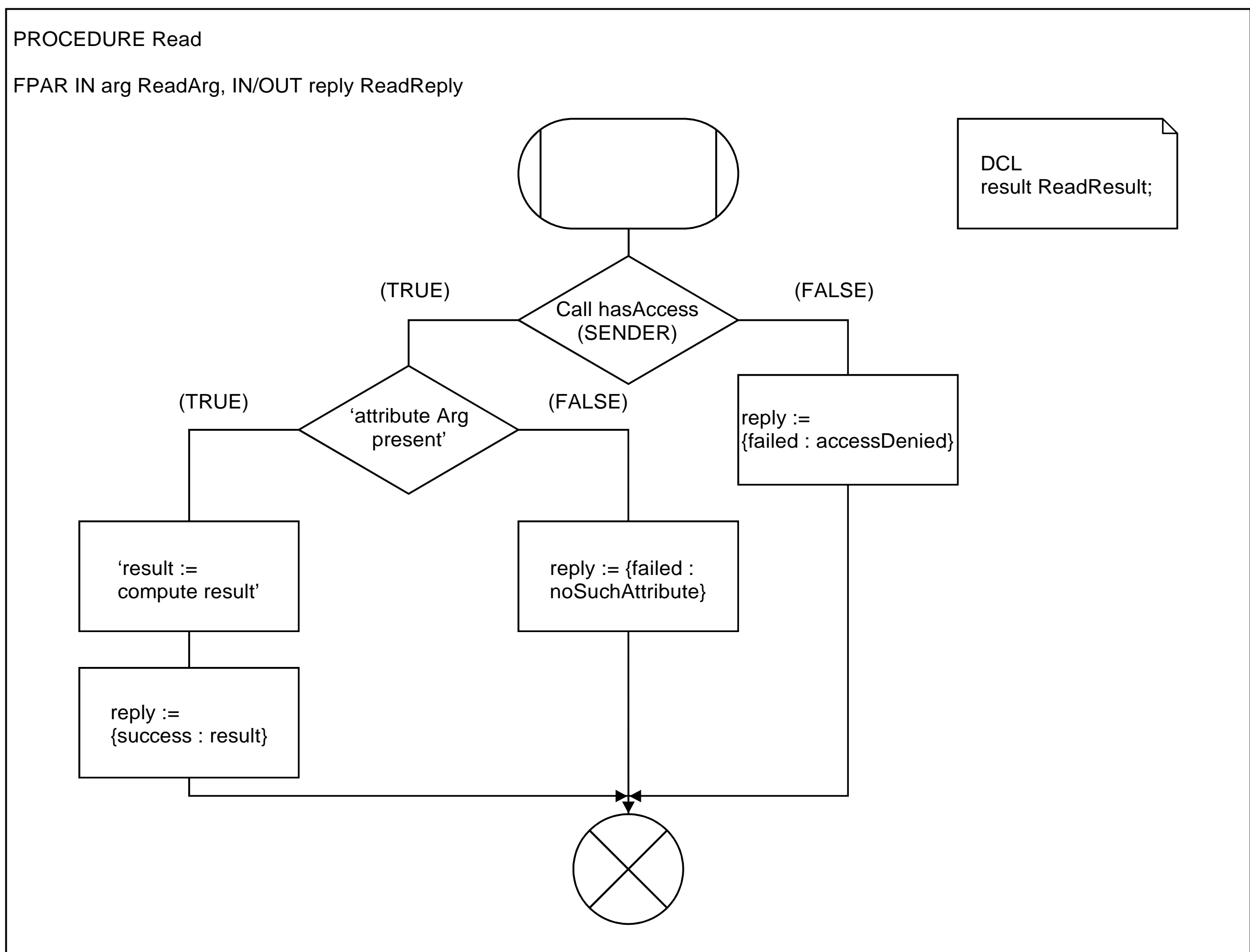


FIGURE II.4.2.3/Z.105

A remote procedure modelling the read operation

The operation is invoked by calling the remote procedure Read, as shown in Figure II.4.2.4. The remote procedure has to be imported by the calling process. This is also shown in the figure.

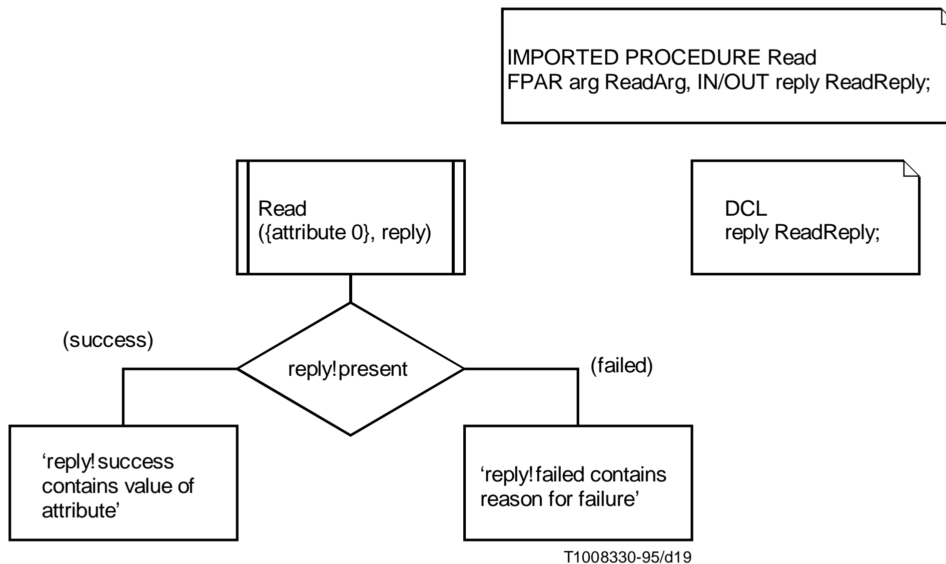


FIGURE II.4.2.4/Z.105

Invoking the read operation

II.5 Some guidelines for the combined use of SDL and ASN.1

The language that is defined in this Recommendation is a mixture of SDL and ASN.1. Because the ASN.1 constructs are mapped to corresponding SDL constructs, it is in principle possible to mix up ASN.1 and SDL completely. It is for example possible to define process diagrams in an ASN.1 module, and ASN.1 data in an SDL package. However, this is not considered to be good use of this Recommendation.

This subclause gives some guidelines on how to use the combination of SDL and ASN.1 as it is intended by this Recommendation. The general principle is:

Use ASN.1 syntax for/within ASN.1 constructs, and SDL syntax for/within SDL constructs

The guidelines below follow from this principle:

- Use only ASN.1 type and value notations in modules. SDL constructs should be defined in packages. Similarly, SDL packages should rather not contain ASN.1 type definitions.

  For example the below defined ASN.1 module WrongUse contains SDL constructs. Although correct according to this Recommendation, such use should be avoided.

```
WrongUse DEFINITIONS ::=
BEGIN
    -- example of wrong use of this Recommendation:
    -- SDL-specific syntax is used within an ASN.1 module
    synonym MaxIndex integer = external;

    syntype index = integer
        constants 0 : MaxIndex
    endsyntype index;
END
```

- Use as much as possible the ASN.1 value notation for types defined using ASN.1 type notation, and SDL notation for types defined using SDL notation.

  Exception: for operators on ASN.1 types, SDL syntax must be used, because there are no operators in ASN.1.

  For example:

  – For S ::= SEQUENCE { a INTEGER }: use { a 5 } as value notation, not (. 5 .)

  – For **newtype** s **struct** a : integer **endnewtype**: use (. 5 .) as value notation, not { a 5 }

  – For IA5String: use "abc" as value notation, not 'abc'

  – For Charstring: use 'abc' as value notation, not "abc"

  – For BIT STRING: '10'B // '0A3C'H is correct use, because the // (concatenation) is an operator that is not available in ASN.1

- It is recommended to adhere to the case sensitivity of ASN.1, even though case sensitivity is not supported in this Recommendation.

  For example:

– Do not use c ::= Choice { A Integer, B Boolean },

  but instead C ::= CHOICE { a INTEGER, b BOOLEAN }

- Avoid as much as possible the use of SDL specific types inside the definition of ASN.1 types, and also use of ASN.1 specific types inside the definition of SDL types. Try to avoid types that mix SDL specific types with ASN.1 specific types. Examples of SDL specific types are PId, Time, Duration, Character. ASN.1 specific types are for example NULL, ANY, and BIT STRING.

  For example:

– Do not use S ::= SEQUENCE { p PId, t Time },

  but instead **newtype** S **struct** p: PId; t : Time **endnewtype**

– Do not use newtype S struct a : ANY; n: NULL endnewtype,

  but instead S ::= SEQUENCE { a ANY, n NULL }

– Try to avoid S ::= SEQUENCE { n Null, p PId }, although this may not be possible.

Appendix III

Supported operators for ASN.1 types

(This appendix does not form an integral part of this Recommendation)

This appendix gives an overview of the supported ASN.1 types, and operators that are available in SDL for these types.

For every type, at least two operators are present: = (equality) and /= (unequality).

Any

Operators:

... = ..., ... /= ...

Bit string

For BIT STRING, a new type BIT is introduced with values 0 and 1, and all BOOLEAN operators.

Some operators are illustrated using type:

Bitstr ::= BIT STRING { bit0(0), bit1(1) }

Operators:

... = ..., ... /= ...,

Length (...),

/* returns the length, i.e. Length('100'B) = 3 */

Mkstring(...),

/* BIT to BIT STRING conversion, e.g. Mkstring(1) = '1'B */

... // ... ,

/* concatenates two BIT STRINGS, e.g. '1'B // '00'B = '100'B */

SubString(..., ..., ...),

/* gives a sub-string of a BIT STRING from given start position and given length, e.g. SubString('011'B, 1, 2) = '01'B*/

...(...),

/* indexing of one bit, starting with 0, i.e. if variable v = '10'B,
then v(0) = 1, v(1) = 0. The named bits can also be used for indexing, i.e. v(bit0) indexes bit 0 for v of type Bitstr defined above */

not ..., ... and ..., ... or ..., ... xor ..., ... = > ...

/* bitwise not, and, or, xor, = > operators */

Boolean

Operators:

... = ..., ... /= ...,

not ..., ... and ..., ... or ..., ... xor ..., ... = > ...

/* logical operations. = > is the implication operator */

Character string types

The character strings are strings of the SDL type Char.

Operators:

... = ..., ... /= ...,

Length(...), Mkstring(...), ... // ..., SubString(..., ..., ...), ...(...),

/* operators are similar to those of BIT STRING, but indexes start with 1,
i.e. for variable v of IA5String with value "abc", v(1) = 'a', v(2) = 'b' */

Num (...)

/* returns canonical number of a character */

Choice

The examples are based on:

C ::= CHOICE {field1 INTEGER, field2 BOOLEAN}, c C ::= {field1:5}

Operators:

... = ..., ... /= ...,

...!<identifier>,

/* field selection, e.g. c!field1 = 5, c!field2 results in a dynamic
error */

<identifier>Present,

/* indicates whether the identifier is present,
e.g. field1_present (c) = TRUE, field2Present (c) = FALSE */

...!present,

/* indicates which identifier was chosen in a value
by giving its identifier, e.g. c!present = field1 */

... < ...

/* ASN.1 selection type,
e.g. field1 < C gives type INTEGER */

Enumerated

The examples are based on:

E ::= ENUMERATED { element2 (2), element1 (1), element3 (3) }

Operators:

... = ..., ... /= ..., ... < ..., ... > ..., ... < = ..., ... > = ...,

/* comparison operators (based on supplied numbers),
e.g. element1 < element2 */

num (...),

/* gives the integer value associated with an enumerated value,
num (element1) = 1 */

pred(...), succ(...),

/* give predecessor and successor of an enumerated value (pred
of first element and succ of last element result in error),
e.g. succ(element1) = element2, pred(element1) results in error */

first (...), last (...)

/* first gives the smallest element, i.e. first (element1) = first (element2) = first (element3) = element1;
last gives the biggest element, i.e. last (element1) = last (element2) = last (element3) = element3 */

Integer

Operators:

... = ..., ... /= ..., ... < ..., ... > ..., ... < = ..., ... > = ...,

/* comparison operators */

... + ..., ... – ..., – ..., ... * ..., ... / ...,

/* addition, substraction, unary minus, multiplication, division */

... rem ..., ... mod ...

/* remainder and modulo, e.g. 10 mod 7 = 3, 10 rem 7 = 3, –10 mod 7 = 4, –10 rem 7 = –3 */

Null

Operators:

... = ..., ... /= ...

Object identifier

A special type Object_element is introduced that has as values all positive integers. Only operators = and /= are available for Object_element.

Operators on OBJECT IDENTIFIER:

... = ..., ... /= ...,

Length (...), Mkstring (...), ... // ..., Substring (..., ..., ...), ...(...)

/* see BIT STRING, but indexing starts with 1 */

Octet string

For OCTET STRING, a new type OCTET is introduced, that has all OCTET STRINGs of size 1 as values, and = and /= as operators.

Operators for OCTET STRING:

... = ..., ... /= ...,

Length(...),

/* returns number of octets, i.e. Length(’3FC’H) = 2 */

Mkstring(...),

/* OCTET to OCTET STRING conversion */

... // ...,

/* concatenation, e.g. ’1’B // ’A6’H = ’10A6’H */

SubString(..., ..., ...)

/* gives a sub-string of an OCTET STRING from a given start
position and given length,
e.g. SubString(’FEDCBA’H, 1, 2) = ’FEDC’H */

...(...),

/* indexing of one octet, starting with 1,
i.e. if variable v = ’3A10’H, then v(2) = ’10’H */

BIT STRING (...),

/* OCTET STRING to BIT STRING conversion */

OCTET STRING (...)

/* BIT STRING to OCTET STRING conversion */

Real

Operators:

... = ..., ... /= ..., ... < ..., ... > ..., ... < = ..., ... > = ...,
... + ..., ... – ..., – ..., ... * ..., ... / ...,

/* similar as INTEGER operators */

Power(..., ...),

/* exponential operator. The arguments are integers.
E.g. Power(2, –1) = 0.5 */

float(...),

/* INTEGER to REAL conversion, e.g. float(4) = {4, 10, 0} = 4.0 */

fix(...)

/* REAL to INTEGER conversion. E.g. fix({19, 10, –1}) = 1 */

Sequence

The examples are based on:

S ::= SEQUENCE {

field1 INTEGER,

field2 INTEGER DEFAULT 7,

field3 BOOLEAN OPTIONAL}

s S ::= {field1 105}

Operators:

... = ..., ... /= ...,

...!<identifier>,

/* gives the value of one component, e.g. s!field1 = 105,
s!field2 = 7, s!field3 results in a dynamic error */

<identifier>Present(...)

/* for optional components: indicates whether the identifier is
present or not: field3Present(s) = FALSE. For default components,
present always gives TRUE, i.e. field2Present(s) = TRUE */

Sequence of

Operators:

... = ..., ... /= ...,

Length(...), Mkstring(...), ... // ..., SubString(..., ..., ...), ...(...)

/* operators are similar to those of BIT STRING, but indexing starts with 1 instead of 0 */

Set

See SEQUENCE.

Set of

The examples are based on SET OF INTEGER

Operators:

... = ..., ... /= ..., ... < ... , ... > ..., ... < = ..., ... > = ...,

/* comparison operators. < is subset of, > superset operator,
e.g. {3, 3} /= {3}, {3, 3} > {3}, {1, 2} = {2, 1} */

... and ..., ... or ...,

/* union and intersection of sets, e.g. {1} or {0, 1} = {1, 0},
{1, 1} and {4, 1} = {1} */

... in ...,

/* is element in set, e.g. 7 in {4, 7} = TRUE, 7 in {4, 8} = FALSE */

Makebag (...),

/* makes a set of one element, e.g. Makebag (1) = {1} */

Incl (..., ...)

/* add element to set, e.g. Incl (5, {2, 3}) = {5, 2, 3} */

Del (..., ...)

/* remove element from set, e.g. Del (2, {2, 3, 2}) = {3, 2},
Del (1, {2}) = {2} */

Length (...)

/* number of elements in the set, e.g. Length ({1, 1, 2}) = 3 */

Take (...)

/* gives one element of the set, e.g. Take ({1, 2, 3}) = 1.
Take ({ }) results in a dynamic error */

Subtypes

All subtyping mechanisms are supported. A subtype has exactly the same operators as its parent type.

Tagged types

Tagged types are allowed to be used, but the tags are completely ignored. A tagged type has exactly the same operators as its base type.

Useful types

As for all types, operators = and /= are defined on the useful types. There are no special operators for these types: the useful types are defined with help of other ASN.1 types.

Appendix IV

Summary of the syntax

(This appendix does not form an integral part of this Recommendation)

Listed below are the syntax productions which in this Recommendation are different from the corresponding syntax productions in Recommendation Z.100. Those productions which are referred, but not defined, are identical to those of Recommendation Z.100. All the listed syntax rules originate from changing the following Z.100 syntax production rules:

1) <lexical unit>
2) <special>
3) <national>
4) <quoted operator>
5) <composite special>
6) <package>
7) <data definition>
8) <sort>
9) <extended properties>
10) <range condition>
11) <extended primary>
12) <extended literal identifier>
13) <keyword>

IV.1 <lexical unit>

<lexical unit> ::= <word> |
<string> |
<special> |
<composite special> |
<note> |
<single line note> |
<keyword>
<string> ::= <character string> |
<quoted string> |
<bitstring> |
<hexstring>
<quoted string> ::= **"** <text> **"**
<bitstring> ::= <apostrophe> { **0** | **1** }* <apostrophe> **B**
<hexstring> ::= <apostrophe>
{<decimal digit> | **A** | **B** | **C** | **D** | **E** | **F** }*
<apostrophe> **H**
<single line note> ::= **--** <text> [**--**]

IV.2 <special>

<special> ::= **+** | **–** | **!** | **/** | **>** | ***** | **(** | **)** | **"** | **,** | **;** | **<** | **=** |
: | **[** | **]** | **{** | **}** | **|** | <full stop>

IV.3 <national>

<national> ::= **#** | **‘** | **$** | **@** | **** | <overline> | <upward arrow head>

IV.4 <quoted operator>

<quoted operator> ::= <quote> <infix operator> <quote> |
<quote> **not** <quote>

IV.5 <composite special>

<composite special> ::= **<<** | **>>** | **= =** | **= =>** | **/=** | **< =** | **> =** |
// | **:=** | **= >** | **>** | **(.** | **.)** | **..** | **...** | **::=**

IV.6 <package>

<package> ::= <package definition> |
<package diagram> |
<module definition>
<module definition> ::= <module> **definitions** [<tagdefault>] ::=
begin [<modulebody>] **end**
<module> ::= <package name> [<objectidentifiervalue>]
<tagdefault> ::= **explicit tags** | **implicit tags** | **automatic tags**
<modulebody> ::= [<exports>] [<imports>] <entity in package>*
<exports> ::= **exports** [<definition selection list>] <end>
<imports> ::= **imports** <symbolsfrommodule>* <end>
<symbolsfrommodule> ::= {<definition selection list> **from** <module>}*

IV.7 <data definition>

<data definition> ::= {<partial type definition> |
<generator definition> |
<syntype definition> |
<synonym definition> |
<sort assignment> |
<value assignment>} <end>
<sort assignment> ::= <sort name> **::=** <extended properties>
<value assignment> ::= <synonym name> <sort> **::=** <ground expression>

IV.8 <sort>

<sort> ::= <sort expression>

<sort expression> ::= {<existingsort> | <subrange> | <sort constructor> |
<inheritance rule> | <generator transformations> |
<structure definition>}

<existingsort> ::= [<package name> **.**] {<sort identifier> | <syntype identifier>} |
any [**defined by** <identifier>] |
<selection>

<selection> ::= <name> **<** <sort>

<sort constructor> ::= <tag> <sort expression> |
<sequence> |
<sequenceof> |
<choice> |
<enumerated>|
<integernaming>

<tag> ::= **[** [**universal** | **application** | **private**]
<simple expression> **]** [**implicit** | **explicit**]

<sequence> ::= { **sequence** | **set** } **{** [<elementsort> { **,** <elementsort>}*] **}**

<elementsort> ::= <namedsort> [**optional** | **default** <ground expression>] |
components of <sort>

<namedsort> ::= [<name>] <sort>

<sequenceof> ::= { **sequence** | **set** } [<sizeconstraint> | <asn1 range condition>]
of <sort>

<choice> ::= **choice** **{** [<namedsort> { **,** <namedsort>}*] **}**

<enumerated> ::= **enumerated** **{** <named number> { **,** <named number>}* **}**

<named number> ::= <named value> | <name>

<integernaming> ::= <identifier> **{** <named value> { **,** <named value>}* **}**

<named value> ::= <name> **(** <simple expression> **)**

<subrange> ::= <sort> **(** <range condition> **)**

IV.9 <extended properties>

<extended properties> ::= <sort expression>

IV.10 <range condition>

<range condition> ::= <range> { { **,** | **|** } <range>}*

<range> ::= <closed range> |
<open range> |
<contained subrange> |
<sizeconstraint> |
<innercomponent> |
<innercomponents>

<closed range> ::= <lowerendvalue> { **:** | **..** } <upperendvalue>

<lowerendvalue> ::= {<ground expression> | **min** } [**<**]

<upperendvalue> ::= [**<**] {<ground expression> | **max** }

<open range> ::= [**=** | **/=** | **<** | **>** | **<=** | **>=**] <ground expression>

<contained subrange>::= **includes** <sort>

<sizeconstraint> ::= **size** **(** <range condition> **)**

<innercomponent> ::= { **from** | **with component** } **(** <range condition> **)**

<innercomponents> ::= **with components**
{ [**... ,**] <named constraint> { **,** <named constraint>}* **}**

<named constraint> ::= <name> [<asn1 range condition>]
[**present** | **absent** | **optional**]

<asn1 range condition> ::= **(** <range condition> **)**

IV.11 <extended primary>

<extended primary> ::= <synonym> |
<indexed primary> |
<field primary> |
<structure primary> |
<choice primary> |
<composite primary>

<choice primary> ::= <identifier> **:** <primary>

<composite primary> ::= [<qualifier>]
{<sequencevalue> |
<sequenceofvalue> |
<objectidentifiervalue> |
<realvalue>}

<sequencevalue> ::= **{** [<namedvalue> { **,** <namedvalue> }*] **}**

<namedvalue> ::= <name> <expression>

<sequenceofvalue> ::= **{** [<expression> { **,** <expression> }*] **}**

<objectidentifiervalue> ::= **{** <objidcomponent>+ **}**

<objidcomponent> ::= <identifier> [(<ground expression>)]

<realvalue> ::= **{** <mantissa> **,** <base> **,** <exponent> **}**

<mantissa> ::= <expression>

<base> ::= <simple expression>

<exponent> ::= <expression>

IV.12 <extended literal identifier>

<extended literal identifier> ::= <character string literal identifier> |
<generator formal name> |
<string primary>

<string primary> ::= [<qualifier>] {<bitstring> | <hexstring> | <quoted string>}

Index

..., 18
<base>, 22
<bitstring>, 4
<choice primary>, 19
<choice>, 13
<closed range>, 17
<composite primary>, 20
<composite special> (Z.100), 4
<contained subrange>, 17
<data definition> (Z.100), 6
<elementsort>, 9
<enumerated>, 14
<existingsort>, 7
<exponent>, 22
<exports>, 5
<extended primary> (Z.100), 19
<extended properties> (Z.100), 7
<hexstring>, 4
<imports>, 5
<innercomponent>, 17
<innercomponents>, 17
<integernaming>, 15
<lexical unit> (Z.100), 3
<lowerendvalue>, 17
<mantissa>, 22
<module definition>, 5
<module>, 5
<modulebody>, 5
<named constraint>, 17
<named number>, 14
<named value>, 15
<namedsort>, 9
<namedvalue>, 20
<national> (Z.100), 3
<objectidentifiervalue>, 21
<objidcomponent>, 21
<open range>, 17
<package> (Z.100), 5
<quoted operator> (Z.100), 4
<quoted string>, 4
<range condition>, 17
<range>, 17
<realvalue>, 22
<selection>, 7
<sequence>, 9
<sequenceof>, 12
<sequenceofvalue>, 21
<sequencevalue>, 20
<single line note>, 4
<sizeconstraint>, 17
<sort assignment>, 6
<sort constructor>, 7
<sort expression>, 7
<sort> (Z.100), 7
<special> (Z.100), 3
<string primary> (Z.100), 22
<string>, 4
<subrange>, 16
<symbolsfrommodule>, 5
<tag>, 7
<tagdefault>, 5
<upperendvalue>, 17
<value assignment>, 7
absent, 18, 19
any, 8, 29, 30, 47
ANY DEFINED BY / TYPE-IDENTIFIER information object class, 42
Any_type, 29
ASN.1 data type, 1
bag, 12, 25
bit, 27
Bit string, 4, 15, 22, 27, 47
boolean, 23, 48
case sensitivity, 3, 31
character, 23
character string types, 48
characteristics of the combination of SDL and ASN.1, 1
CHOICE, 41
Choice, 13
choice, 48
combination of ASN.1 and SDL, 1
combination of SDL and ASN.1, 1
Comparing data values, 36
components of, 10
Composite primary, 20
constants, 16
dash in ASN.1 names, 31
data definitions, 6
data type, 1
default, 9, 10
defined by, 8
Defining ASN.1 data types in SDL, 33
Defining ASN.1 types directly in SDL, 33
Definition and use of data, 6
Differences between SDL data and ASN.1, 1
dot in a name, 32
encoding rules, 1, 30
Enumerated, 14
enumerated, 49
enumeration, 15, 25
external, 30
external type reference, 31

external value reference, 31
EXTERNAL_Type, 30
field, 8
Fields and optionality, 8
from, 17
GeneralizedTime, 30
GraphicString, 24
Guidelines to circumvent restrictions on ASN.1, 42
Illustration of the use of some ASN.1 types in SDL, 36
Importing a type definition from an ASN.1 module, 33
imports, 5
inner component, 18
integer, 15, 49
Integer Naming, 15
Keywords, 3
Lexical units, 3
macro, 30
Manipulating values, 36
max, 18
min, 18
MINUS_INFINITY, 25
Modelling an operation by a remote procedure, 45
Modelling an operation by signal exchange, 43
module, 5
nameclass, 10
named bits, 28
named numbers, 31
Names, 3
null, 29, 49
object identifier, 29, 49
Object identifier value, 21
object_element, 29
ObjectDescriptor, 30
octet, 28
Octet string, 4, 22, 28, 50
Operators on simple ASN.1 types, 36
optional, 10, 18
optional field, 17, 19
Package (Z.100), 5
package Predefined, 23
PLUS_INFINITY, 25
power operator, 22
predefined package, 5, 23
present, 18
present operator, 10, 13
productions, 3
quoted operator, 32
Quoted operators, 4
range, 17
Range condition, 16
range condition, 16
real, 23, 31, 50
Real value, 22
Restrictions on ASN.1 and SDL, 30
Restrictions on SDL and ASN.1, 2
SDL in Combination with ASN.1 Predefined Data, 23
selection, 8
SEQUENCE, 37
Sequence, 9
sequence, 51
SEQUENCE OF, 38
sequence of, 12, 51
Sequence value, 20
Sequenceof value, 21
set, 9, 51
SET OF, 40
set of, 51
Single line notes, 4
size, 12
Some guidelines for the combined use of SDL and ASN.1, 46
Sort assignments, 6
Sort expressions, 7
sorts, 1
spaces in names, 5
Special symbols, 4
String primary, 22
string0 generator, 23
Strings, 4
Subrange, 16
subtypes, 52
Summary of the supported subset of ASN.1, 30
Summary of the supported subset of SDL, 32
Summary of the syntax, 52
Supported operators for ASN.1 types, 47
synonym, 21
syntax productions, 52
syntype, 16
tag, 5, 30
tagged types, 52
The OPERATION macro / information object class, 43
UniversalString, 24
Use of space characters in names, 5
useful types, 30, 52
User defined operators on ASN.1 types, 34
Using ASN.1 values in SDL, 35
UTCTime, 30
Value assignments, 7
Value expressions, 19
Variable and data definitions, 6
with component, 17