INTERNATIONAL TELECOMMUNICATION UNION

ITU-T Z.105

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

(11/99)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

Formal description techniques (FDT) – Specification and Description Language (SDL)

SDL Combined with ASN.1 modules (SDL/ASN.1)

ITU-T Recommendation Z.105

(Previously CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of Formal Description Techniques	Z.110–Z.119
Message Sequence Chart	Z.120–Z.129
PROGRAMMING LANGUAGES	
CHILL: The ITU-T high level language	Z.200–Z.209
MAN-MACHINE LANGUAGE	
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

For further details, please refer to ITU-T List of Recommendations.

ITU-T RECOMMENDATION Z.105

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

Summary

Objective

This Recommendation defines how Abstract Syntax Notation One (ASN.1) modules can be used in combination with Specification and Description Language (SDL). The intention is that the structure and the behaviour of systems are described with SDL, while parameters of exchanged messages are described with ASN.1. This Recommendation defines a mapping of ASN.1 constructs to already existing SDL constructs and contains only a small extension to Recommendation Z.100 to allow ASN.1 modules to be used.

Coverage

This Recommendation presents a semantic definition for the combination of SDL and ASN.1 modules. A mapping of the ASN.1 data defined in a module to the corresponding SDL constructs defined in Recommendation Z.100 [1] is given, including the operators that can be applied to the ASN.1 data. The ASN.1 data items can then be used within SDL (using SDL notation).

The use of ASN.1 notation embedded in SDL is defined in Recommendation Z.107 [2].

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.

NOTE – "Specification" in this Recommendation includes definition of requirements in a standard, Recommendation, or procurement document, and description of an implementation.

A specification conforms to this Recommendation if and only if it conforms to the syntactic and semantic grammar rules for the formal technical language defined by the Recommendation (which includes the referenced ASN.1 and SDL languages). Conformance implies that every possibly dynamic interpretation of the specification conforms to the language rules. A specification that uses extensions of the language does not conform.

A tool does not fully support the language if it rejects some constructs of the language or that has a static or dynamic interpretation of a specification in the language that does not conform to language semantics.

Status/stability

This Recommendation replaces the semantic mappings from ASN.1 to SDL defined in Recommendation Z.105 (1995). The use of ASN.1 notation embedded in SDL previously defined in Recommendation Z.105 (1995) is not defined by this Recommendation.

Changes to Recommendations X.680 [3], X.681 [4], X.682 [5] and X.683 [6] or Z.100 [1] may require modifications to this Recommendation.

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

Associated work

ITU-T Recommendation Z.100 (1999), *Specification and Description Language (SDL)*.

ITU-T Recommendation X.680 (1997), *Specification of basic notation*.

ITU-T Recommendation X.681 (1997), *Information object specification.*

ITU-T Recommendation X.682 (1997), *Constraint specification*.

ITU-T Recommendation X.683 (1997), *Parameterization of ASN.1 specifications*.

ITU-T Recommendation Z.107 (1999), *SDL with embedded ASN.1*.

Source

ITU-T Recommendation Z.105 was revised by ITU-T Study Group 10 (1997-2000) and was approved under the WTSC Resolution No. 1 procedure on 19 November 1999.

FOREWORD

ITU (International Telecommunication Union) is the United Nations Specialized Agency in the field of telecommunications. The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the ITU. 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.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

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

INTELLECTUAL PROPERTY RIGHTS

The ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. The ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, the ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

		Page
1	Introduction	1
1.1	Objective	1
1.2	The characteristics of the combination of SDL and ASN.1 modules	1
1.3	ASN.1 that can be used in combination with SDL	1
1.4	The structure of this Recommendation	2
1.5	Conventions used in this Recommendation	2
2	References	2
3	Package	3
4	Definition and use of data	4
4.1	Name mapping	4
4.2	Variable and data definitions	4
4.2.1	Type assignment	4
4.2.2	Value assignment	5
4.3	Type expressions	5
4.3.1	Sequence	5
4.3.2	Sequenceof	6
4.3.3	Choice	7
4.3.4	Enumerated	7
4.3.5	Integer and Bit Naming	8
4.3.6	Subrange	9
4.3.7	BitString	9
4.3.8	OctetString	9
4.4	Range condition	9
4.5	Value Expressions	11
4.5.1	Choice Primary	11
4.5.2	Composite primary	11
4.5.3	String primary	14
4.5.4	Element set specification	14
5	Mapping of ASN.1 types defined in ASN.1 modules using information objects, classes and sets	14
5.1	Introduction	14
5.2	Object class field value	15
5.3	Objects and object sets	16
6	Mapping of parameterized ASN.1 types	18
6.1	Parameterized type assignment	19

Page

6.2 Parameterized value assignment 19
6.3 Referencing ASN.1 parameterized definitions 20
6.4 Parameterized object definition together with an information object class 20
7 Additions to package Predefined 21

Recommendation Z.105

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

(revised in 1999)

1 Introduction

This Recommendation defines how ASN.1 modules can be used in combination with SDL. ASN.1 modules are imported in SDL descriptions so that ASN.1 data definitions are mapped to internal SDL representation using equivalent SDL constructs and forming together with the rest of the SDL description a complete specification.

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 can be described using SDL, and data and messages using ASN.1. Encoding of these messages can be described by reference to the relevant encoding rules that are defined for ASN.1.

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

1.2 The characteristics of the combination of SDL and ASN.1 modules

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

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

1.3 ASN.1 that can be used in combination with SDL

The use of ASN.1 as defined in Recommendations X.680, X.681, X.682 and X.683 is supported in combination with SDL, with a recognition that some ASN.1 constructs cannot be successfully mapped to SDL (or at least the mapping has not been identified and specified in this Recommendation). The constructs that cannot be mapped to SDL will exist in ASN.1 packages used as a source of transformation. During the transformation to SDL they are effectively treated as if not present and should not cause any problems for successful transformation of other constructs. Such constructs are the extension marker and exception marker defined in Recommendation X.680, which may be present in ASN.1 but are ignored in the transformation to SDL. Parts of the ASN.1 grammar

(1997) related to extension and exception markers are therefore not used in this Recommendation. Some constructs of ASN.1 are never transformed to SDL as such, but contain information that can direct or be used in the transformation. The prominent examples of such constructs are relational constraints as defined in Recommendation X.682, object classes and object sets.

The use of SDL as defined in Recommendation Z.100 [1] is supported.

ASN.1 modules that are used in the transformation to SDL can also be used for generation of encoders and decoders, provided that encoding rules are defined. The SDL data specification derived from ASN.1 modules should not be used for such a purpose since some information that is relevant for encoding may be lost in the transformation to SDL.

1.4 The structure of this Recommendation

This Recommendation is not self-contained: the mapping defined in this Recommendation is based on Recommendation Z.100 and Recommendations X.680, X.681, X.682 and X.683. The language as defined in Recommendation Z.100 applies, except that the <package> production rule is extended to allow direct use of ASN.1 modules. This Recommendation is structured in the following manner:

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

Clause 4 defines the mapping of X.680 ASN.1 types and values to Recommendation Z.100 data in order to incorporate ASN.1 data types and values.

Clause 5 defines the mapping of ASN.1 types defined using information objects, classes and information object sets. The use of X.682 constructs is also treated in this clause.

Clause 6 defines the mapping of parameterized ASN.1 types to Recommendation Z.100 data in order to incorporate parameterized ASN.1 data types.

Clause 7 defines the additions to the package Predefined needed to support the use of ASN.1.

1.5 Conventions used in this Recommendation

The conventions of Recommendation Z.100 normally apply: for example, keywords appear in lowercase boldface, and predefined names start with a capital. However, in ASN.1 examples, the ASN.1 conventions are used in order to respect ASN.1 rules and improve readability for ASN.1 users: for example, keywords are in capitals (no boldface).

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; all users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published.

[1] ITU-T Recommendation Z.100 (1999), *Specification and Description Language (SDL-2000)*.

[2] ITU-T Recommendation Z.107 (1999), *SDL with embedded ASN.1*.

[3] ITU-T Recommendation X.680 (1997) | ISO/IEC 8824-1:1998, *Information Technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*, plus Amendments 1 and 2 (1999), and Corrigendum 1 (1999).

[4] ITU-T Recommendation X.681 (1997) | ISO/IEC 8824-2:1998, *Information Technology – Abstract Syntax Notation One (ASN.1): Information object specification*, plus Amendment 1 (1999), and Corrigendum 1 (1999).

[5] ITU-T Recommendation X.682 (1997) | ISO/IEC 8824-3:1998, *Information Technology – Abstract Syntax Notation One (ASN.1): Constraint Specification*.

[6] ITU-T Recommendation X.683 (1997) | ISO/IEC 8824-4:1998, *Information Technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications,* plus Amendment 1 (1999).

[7] ITU-T Recommendation X.690 (1997) | ISO/IEC 8825-1:1998, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*.

3 Package

The production <package> is extended as follows:

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

where ModuleDefinition is a non-terminal defined in ITU-T X.680: 1997.

Model

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

- ModuleIdentifier (without any DefinitiveIdentifier) corresponds to the <package name>;
- Imports corresponds to the <package reference clause>s;
- Exports corresponds to the <interface>.

An ASN.1 package is transformed into the equivalent SDL, before it is considered as a package, and before any Z.100 transformations. In this transformation, names are transformed into fully qualified identifiers where SDL requires or allows an identifier rather than a name. However, for conciseness this is often omitted from the examples in this Recommendation.

Example

The ASN.1 module definition

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

is the same as

```
package myway;
public synonym yes, synonym no;
synonym    yes <<package Predefined>>Boolean = true;
synonym    no <<package Predefined>>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;

NOTE – Because SDL does not support object identifier values for package identification, ASN.1 modules with the same modulereference but different DefinitiveIdentifiers will potentially cause name resolution problems.

4 Definition and use of data

The different definitions of the use of data are described the following way:

ASN.1 Grammar	Defining the grammar production rules representing the construction to be represented in SDL.
Model	Describing the transformations of the different parts of the ASN.1 grammar into SDL productions. This part is referencing both the SDL grammar, represented as <SDL grammar rule>, and the ASN.1 grammar, represented as **ASN1GrammarRule.**

4.1 Name mapping

ASN.1 Grammar

ASN.1 names are allowed to contain dash characters ("-"). If this is used in SDL this would be interpreted as minus operator.

Model

ASN.1 names containing dash characters are mapped to lexically same SDL names except that dash characters are converted to underline characters.

Example

The ASN.1 name my-example-name is mapped to my_example_name in SDL.

4.2 Variable and data definitions

4.2.1 Type assignment

ASN.1 Grammar

TypeAssignment ::= typereference "::=" Type

Model

If the **Type** is a **typereference**, then the **TypeAssignment** is the same as a <syntype definition> containing only the SDL equivalent of the **Type**.

If the **Type** is a **constrainedType**, then the **TypeAssignment** is the same as a <syntype definition> containing only the SDL equivalent of the **Constraint**.

If the **Type** is a neither a **typereference** nor a **constrainedType** the **TypeAssignment** is represented by a <partial type definition> where <properties expression> is empty and where <formal context parameters> is omitted.

Example

The ASN.1 type assignment

Mytype ::= AnotherType -- *typereference*

is the same as

syntype Mytype = AnotherType **endsyntype** Mytype; /* full qualification omitted here. */

The ASN.1 type assignment

S ::= INTEGER (0..5 | 10)

is the same as

syntype S = <<**package** Predefined>>Integer **constants** 0:5,10 **endsyntype** S;

The ASN.1 type assignment

Integerlist ::= SEQUENCE OF INTEGER

is the same as

value type Integerlist **inherits** <<**package** Predefined>>String
< <<**package** Predefined>> Integer> (" = <<**package** Predefined>>Emptystring) **endvalue type** Integerlist;

4.2.2 Value assignment

ASN.1 Grammar

ValueAssignment ::= valuereference Type "::=" Value

Model

A **ValueAssignment** is represented by a <synonym definition item>.

Example

The ASN.1 definition

yes BOOLEAN ::= TRUE

is the same as

synonym yes <<**package** Predefined>>Boolean = <<**package** Predefined>>true;

4.3 Type expressions

4.3.1 Sequence

ASN.1 Grammar

SequenceType ::= SEQUENCE "{" ComponentTypeList "}" |
SEQUENCE "{" "}"

ComponentTypeList ::= ComponentType |
ComponentTypeList "," ComponentType

ComponentType **::=** **NamedType |**
NamedType OPTIONAL |
NamedType DEFAULT Value |
COMPONENTS OF Type
NamedType **::=** **identifier Type**

Model

A **SequenceType** is represented as a <structure definition> containing a <field> for each **NamedType** of the **SequenceType**. The <field> contains one <field name>, which is the same as the ASN.1 **identifier** of the **NamedType,** and a <field sort>, which is the **Type** transformed to an SDL <sort identifier>.

If the **ComponentType** containing the **NamedType** is **OPTIONAL**, the SDL field has the keyword **optional**.

If the **ComponentType** containing the **NamedType** has a **DEFAULT Value**, the SDL field has the keyword **default** and the value is transformed into the <ground expression> after **default**.

A **ComponentType** that is **COMPONENTS OF Type** is represented as a list of ordered <field>s, one for each field associated to **Type**. These fields are inserted in the position of the **COMPONENTS OF Type** in the order that the fields exist in the **Type**.

Example

The ASN.1 type:

```
S     ::=   SEQUENCE {
a     INTEGER,
b     IA5String OPTIONAL,
c     PrintableString DEFAULT "d"}
```

is the same as

```
value type S struct
      a <<package Predefined>> Integer;
      b <<package Predefined>> IA5String optional;
      c <<package Predefined>> PrintableString default 'd';
endvalue type S;
```

NOTE 1 – There is no distinction between use of keyword SEQUENCE and SET. This is a relaxation compared to Recommendation X.680.

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

4.3.2 Sequenceof

ASN.1 Grammar

SequenceOfType **::=** **SEQUENCE OF Type**
SetOfType **::=** **SET OF Type**

Model

Specifying a **SequenceOfType/SetOfType** is the same as specifying the predefined abstract String or Bag sort having the SDL transform of Type as the first <actual context parameter> and the name Emptystring defined as the literal name for the empty string. The abstract sort is the <<**package** Predefined>> String for **SEQUENCE OF**, and <<**package** Predefined>> Bag for **SET OF**.

If an ASN.1 size constraint is specified for **Type**, the **SequenceOfType** (or **SetOfType**) is a syntype having the transformed size constraint as a <range condition> (see 4.4). The parent sort of the

syntype is the **SequenceOfType** (or **SetOfType** respectively) without the ASN.1 size constraint. This parent sort has an implicit and unique name and is defined in the nearest scope unit enclosing the occurrence of the **SequenceOfType** (or **SetOfType** respectively).

Example

The ASN.1definition:

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

is the same as the three SDL definitions:

```
value type S1 inherits <<package Predefined>> String <S2> ( " = Emptystring ) endvalue type S1;
syntype S2 = <<package Predefined>> Integer constants 0:9 endsyntype;
syntype phonenumber = S1 constants size (8) endsyntype phonenumber;
```

4.3.3 Choice

ASN.1 Grammar

```
ChoiceType          ::=  CHOICE "{" AlternativeTypeList "}"

AlternativeTypeList ::=
                         NamedType                |
                         AlternativeTypeList "," NamedType
```

Model

A **ChoiceType** is represented as a <choice definition> containing a <field> for each **NamedType** of the **ChoiceType**.

Example

The ASN.1 choice type

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

is the same as

```
value type C choice
        a <<package Predefined>> Integer;
        b <<package Predefined>> Real;
endvalue type C;
```

4.3.4 Enumerated

ASN.1 Grammar

```
EnumeratedType      ::=  ENUMERATED "{" Enumeration "}"

Enumeration         ::=  EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem     ::=  identifier | NamedNumber

NamedNumber         ::=  identifier "(" SignedNumber ")"|
                         identifier "(" DefinedValue ")"
```

Model

For each **EnumerationItem**, the **identifier** is transformed into a <literal signature> that has the same name as the **EnumerationItem**. If the **EnumerationItem** contains a **SignedNumber** (or **DefinedValue**), the <literal name> of the <literal signature> is followed by the SDL transform of the **SignedNumber** (or **DefinedValue** respectively).

The definition:

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

is the same as

```
value type colours
    literals blue = 3, red, yellow = 0-
end value type colours;
```

4.3.5 Integer and Bit Naming

ASN.1 Grammar

```
IntegerType        ::=  INTEGER                    |
                        INTEGER "{" NamedNumberList "}"

NamedNumberList    ::=  NamedNumber                |
                        NamedNumberList "," NamedNumber

NamedNumber        ::=  identifier "(" SignedNumber ")"    |
                        identifier "(" DefinedValue ")"

BitStringType      ::=  BIT STRING | BIT STRING "{" NamedBitList "}"

NamedBitList       ::=  NamedBit | NamedBitList "," NamedBit

NamedBit           ::=  identifier "(" number ")"          |
                        identifier "(" DefinedValue ")"
```

Model

Specifying an **IntegerType** with a **NamedNumberList** (or **BitStringType** with a **NamedBitList**) is the same as specifying a <synonym definition> in the nearest enclosing scope unit with one <synonym definition item> for each **NamedNumber** (or **NamedBitList** respectively). The **identifier** of the **NamedNumber** (or **NamedBit** respectively), is transformed into the <synonym name>. The <sort> of the <synonym definition item> is <<**package** Predefined>>Integer in the case of a **NamedNumber**, and <<**package** Predefined>>Bit in the case of a **NamedBit**. The **SignedNumber** or **DefinedValue** or **number** of the **NamedNumber** or **NamedBitList** is used as the <ground expression> of the <synonym definition item>.

Example

The ASN.1 definition:

```
Standards    ::=    SEQUENCE OF INTEGER{z100(0),x680(1),z10x(2)}
```

is the same as

```
value type standards inherits
    << package Predefined >> String < <<package Predefined>> Integer> ("= EmptyString)
endvalue type standards;
synonym    z100        Integer = 0;
```

```
synonym   x680      Integer = 1,
synonym   z10x     Integer = 2;
```

4.3.6 Subrange

Model

Specifying an ASN.1 subrange restriction is represented as specifying the contained <sort> and adding the representation of the ASN.1 subrange restriction after the **constants** keyword in the <syntype> (if specified, otherwise the construct is introduced).

Example

The ASN.1 definition:

```
S     ::=   INTEGER(0..5 | 10)
```

is equivalent to

```
syntype S = <<package Predefined>> Integer constants 0:5, 10 endsyntype S;
```

How the <range condition> is derived is described below.

4.3.7 BitString

ASN.1 Grammar

```
BitStringType ::=
        BIT STRING
        BIT STRING "{" NamedBitList "}"
```

Model

The ASN.1 BitStringType is mapped to SDL <<**package** Predefined>> Bitstring.

4.3.8 OctetString

ASN.1 Grammar

OctetStringType ::= OCTET STRING

Model

The ASN.1 type **OctetStringType** is mapped to SDL <<**package** Predefined >>Octetstring.

4.4 Range condition

Model

A range condition defines a set of values. It is used for defining a syntype. It has an associated parent sort, which is the sort specified in the syntype definition. 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:

```
value type A
operators o: S -> Boolean;
/* where o is derived from the ASN.1 concrete syntax as explained below */
endvalue type A;
```

Each **Range** in the ASN.1 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.

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 **ClosedRange,** a **ClosedRange** contributes with:

  E1 rel1 V and V rel2 E2

  where E1 is **Value** of **LowerEndValue** and E2 is **Value** 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** contributes with:

  V rel2 E2

  If the keyword **MAX** is specified and the keyword **MIN** is not specified, **Range** contributes with:

  E1 rel1 V

  If both keywords **MIN** and **MAX** are specified, the operator always yields true.

- A **ContainedSubType** contributes with:

  o1(V)

  where o1 is the implicit operator defining the value set for the **Type** mentioned in the **ContainedSubType**.

- 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**.

- **InnerTypeConstraints** 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 **InnerTypeConstraints** contributes to and o1 is the implicit operator for **Range** specified in **InnerTypeConstraints**.

  InnerTypeConstraints has a contribution for each contained **NamedConstraint** that specifies constraints of the field (see 4.2.1) denoted by **Identifier** of the parent sort.

  The keyword **PRESENT** is added to the **NamedConstraint**s that have no ending keyword (**PRESENT**, **ABSENT** or **OPTIONAL**) and **NamedConstraint**s of the form **Identifier ABSENT** are added for all fields (i.e. **Identifier**s) not mentioned explicitly in a **NamedConstraint**. The **NamedConstraint**s are added to the **InnerTypeConstraints** before the contributions of each **NamedConstraint** are derived.

If a **Range** is specified for a **NamedConstraint**, 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**. If the **Range** is omitted, the contribution is only the present constraint E.

The present constraint for a field F is:

FPresent(V)

in case the **NamedConstraint** for the field contains the keyword **PRESENT**; and

not FPresent(V)

in case the **NamedConstraint** for the field contains the keyword **ABSENT**. In all other cases, the present constraint is true.

4.5 Value Expressions

4.5.1 Choice Primary

ASN.1 Grammar

ChoiceValue ::= identifier ":" Value

Model

A **ChoiceValue** is represented as an <operator application> having the **Value** as argument. The <operator identifier> in the <operator application> contains a <qualifier> representing the **Type** and an operator name being the **identifier**.

Example

The **ChoiceValue**:

myvalue : Mychoice

is represented as:

myvalue(Mychoice)

In case that a **ChoiceValue** can denote one of several operator applications (i.e. a field of more than one choice sort), a qualifier is used:

MyType ::= CHOICE …

myvalue : Mychoice

which is then represented as:

<< **type** Mytype >> myvalue(Mychoice)

4.5.2 Composite primary

A composite primary is built up of the values for the SDL-representation of respective composite types.

4.5.2.1 Sequence value

ASN.1 Grammar

SequenceValue ::= "{" ComponentValueList "}" | "{" "}"
ComponentValueList ::= NamedValue |
ComponentValueList "," NamedValue

NOTE – There is no distinction between SetValue and SequenceValue. This is a relaxation compared to Recommendation X.680.

Model

The **SequenceValue** specification in ASN.1 is mapped to SDL **synonym** definition. In the mapping **ComponentValueList** is provided to structure data type constructor in SDL. The SDL data type constructor requires that all the fields are given as input so that fields that are omitted in **ComponentValueList** have to be provided empty in the SDL The application of structure data type constructor will have the same effects in SDL as it would in ASN.1.

Example

```
MYTYPE ::= SEQUENCE{
        a       INTEGER,
        b       INTEGER OPTIONAL,
        c       INTEGER DEFAULT 0,
        d       INTEGER,
        e       INTEGER OPTIONAL,
        f       INTEGER DEFAULT 0.
}
myValue MYTYPE ::= {a 1, b 1, c 1, d 1}
```

In this example fields a, b and c have a specified value and fields c, d and e are omitted.

```
synonym myValue MYTYPE = (. 1, 1, 1, 1, , .);
```

The consequences would be that fields a, b, c and d would be set to 1, e would be absent and f would get the default value 0.

4.5.2.2 Sequence of value

ASN.1 Grammar

SequenceOfValue ::= "{" ValueList "}" | "{" "}"

ValueList ::= Value | ValueList "," Value

NOTE – There is no distinction between SetOfValue and SequenceOfValue. This is a relaxation compared to Recommendation X.680.

Model

A **SequenceOfValue** is represented as:

```
MkString(E1) // MkString(E2)// ... // MkString(En)
```

where E1, E2, ..., En are the **Values** of the **SequenceOfValue** in the order of appearance. If no **Values** are specified, the **SequenceOfValue** is represented as the name Emptystring.

The **Type** qualifier of the Composite Primary that contains the SequenceOfValue precedes each MkString operator or the Emptystring literal respectively.

4.5.2.3 Object identifier value

ASN.1 Grammar

```
ObjectIdentifierValue   ::=   "{" ObjIdComponentList "}"      |
                              "{" DefinedValue ObjIdComponentList "}"

ObjIdComponentList      ::=   ObjIdComponent|
                              ObjIdComponent ObjIdComponentList

ObjIdComponent          ::=   NameForm        |
                              NumberForm      |
                              NameAndNumberForm

NameForm                ::=   identifier

NumberForm              ::=   number | DefinedValue

NameAndNumberForm       ::=   identifier "(" NumberForm ")"
```

Model

Object identifier value is in ASN.1 used to distinguish between the modules that have same names but different object identifiers. Because the module names and object identifiers cannot uniquely be mapped to a package identifier that is used in package use clauses, the object identifier component is ignored in the transformation to SDL. The identification of appropriate module is thus open to manual or tool specific solutions.

4.5.2.4 Real value

ASN.1 Grammar

The value of a real type is in ASN.1 defined by the notation "RealValue":

```
RealValue ::=
        NumericRealValue | SpecialRealValue
NumericRealValue ::= 0    |
        SequenceValue     -- Value of the associated sequence type
SpecialRealValue ::=
        PLUS-INFINITY | MINUS-INFINITY
```

The form **0** is used for zero values; the alternate form for **NumericRealValue** shall not be used for zero values.

The associated type for value definition and subtyping purposes is:

```
SEQUENCE {
    mantissa    INTEGER,
    base        INTEGER (2|10),
    exponent    INTEGER
        -- The associated mathematical real number is "mantissa"
        -- multiplied by "base" raised to the power "exponent"
    }
```

Model

An ASN.1 **NumericalRealValue** is mapped to an SDL **real** sort value with the actual value calculated in the transformation. The **SpecialRealValue** shall be transformed to the largest possible positive or negative value respectfully.

NOTE – The transformation of **SpecialRealValue** is not in accordance with the intended ASN.1 semantics because this is a directive to the encoder/decoder to use a special code indicating the - infinite values. Since encoding is not related to data in SDL transformed from ASN.1data, such relaxation should be acceptable.

Example

The ASN.1 definition:

```
r50    REAL        ::= { mantissa 5, base 10, exponent 1}
```

is the same as:

```
synonym r50 Real = 50.0;
```

4.5.3 String primary

Model

An ASN.1 **StringValue** containing a **cstring** (ASN.1 name for character string delimited by " at both beginning and end) represents a <character string literal identifier> consisting of the **Type** and a <character string literal> with the same <text> as the ASN.1 String **Text**. The **Type** for **cstring** is an IA5Type as defined by this Recommendation.

A **StringValue** containing a **BitStringValue** or **HexStringValue** are mapped to SDL <<**package** Predefined>> Bitstring operators with the same syntax.

4.5.4 Element set specification

ASN.1 Grammar

```
ElementSetSpec ::= Unions |
                   ALL Exclusions
Unions ::= Intersections |
                   UElems UnionMark Intersections
UElems ::= Unions
Intersections ::= IntersectionElements |
                   IElems IntersectionMark IntersectionElements
IElems ::= Intersections
IntersectionElements ::= Elements | Elems Exclusions
Elems ::= Elements
Exclusions ::= EXCEPT Elements
UnionMark ::=  "|"   |   UNION
        IntersectionMark ::=  "^"   |   INTERSECTION
```

Model

Two or more value sets can be combined using this notation. The resulting set is evaluated in the transformation and the result is mapped to SDL.

5 Mapping of ASN.1 types defined in ASN.1 modules using information objects, classes and sets

5.1 Introduction

Recommendation X.681 provides the ASN.1 notation that allows information object classes as well as individual information objects and sets thereof to be defined and given reference names. An information object class is a template for a collection of information that makes up the attributes of any members of that class. Information objects provide a generic table mechanism within the ASN.1 language. Such a generic table defines the association of specific sets of field values or types. This

feature replaces the earlier MACRO construct (available in ASN.1: 1990) and is primarily used to fill-in gaps in a type definition dependant on one or more key fields.

This clause assumes that all ASN.1 constructs defined in Recommendations X.681, X.682 and X.683 can be used in ASN.1 modules. It then identifies what information contained in ASN.1 information object classes, information objects and information object sets can be useful when mapped to appropriate SDL targets. The mappings that are possible and useful are defined. It has to be noted that some information will not be represented in SDL because of the differences in nature of the two languages.

5.2 Object class field value

ASN.1 Grammar

```
ObjectClassFieldValue ::=
        OpenTypeFieldVal |
        FixedTypeFieldVal
OpenTypeFieldVal ::= Type ":" Value
FixedTypeFieldVal ::= BuiltinValue | ReferencedValue
```

Model

The ASN.1 object class specification is never mapped to SDL. However, information contained therein is essential for mapping those elements that are defined by reference to the class.

In specification of single ASN.1 type whose fields are defined by reference to a class, only Fixed type value and value set fields can be used. The mapping to SDL is done so that the field name is mapped from the type that is being transformed and the field type can be found in the referenced field of the class specification. Open type field values cannot be mapped to SDL. However, the classes were not primarily designed for this kind of use (single ASN.1 type specification) and this should not pose a problem in practice. Specification of optional or default values is mapped to SDL as specified in this specification.

Example

If the ASN.1 contains the following specification:

```
EXAMPLE-CLASS ::= CLASS {
        &TypeField                              OPTIONAL,     -- class field 1
        &fixedTypeValueField        INTEGER     OPTIONAL,     -- class field 2
        &variableTypeValueField     &TypeField  OPTIONAL,     -- class field 3
        &FixedTypeValueSetField     INTEGER     OPTIONAL,     -- class field 4
        &VariableTypeValueSetField  &TypeField  OPTIONAL      -- class field 5
}
WITH SYNTAX {
        [TYPE-FIELD     &TypeField]
        [FIXED-TYPE-VALUE-FIELD                 &fixedTypeValueField]
        [VARIABLE-TYPE-VALUE-FIELD              &variableTypeValueField]
        [FIXED-TYPE-VALUE-SET-FIELD             &FixedTypeValueSetField]
        [VARIABLE-TYPE-VALUE-SET-FIELD          &VariableTypeValueSetField]
}
ExampleType ::= SEQUENCE {
        integerComponent1       EXAMPLE-CLASS.&fixedTypeValueField,       -- field 1
        integerComponent2       EXAMPLE-CLASS.&FixedTypeValueSetField     -- field 2
}
```

```
exampleValue ExampleType ::= {
        integerComponent1       123,                    -- field 1
        integerComponent2       456                     -- field 2
}
```

Things that can be mapped to SDL are ExampleType and exampleValue:

```
value type ExampleType
        integerComponent1       <<package Predefined>> Integer,     /* field 1 */
        integerComponent2       <<package Predefined>> Integer      /* field 2 */
endvalue type ExampleType;
synonym exampleValue ExampleType = (. 123, 456 .);
```

5.3 Objects and object sets

Model

Classes are dominantly used in the specification of objects based on the class and sets of such objects. All these constructs are then used to define a type that is a generic description of a set of types, provided that a constraint specification is naming the object set by which the type is constrained.

In the transformation to SDL the following steps are performed:

1) the name of the constraining object set is used to identify the objects that shall be mapped to SDL,
2) for each object a value type in SDL is created,
3) each value type has as many fields as there are fields in the object specification,
4) the names of fields are derived from names of the matching fields in constrained type specification,
5) the type specification for each field is derived in the following way: if the field is fixed type value or type set field, the SDL type is derived from the type of the matching field in the referenced object class with a subrange specification derived from the matching field of the object specification. If the field type is open in the class specification, the SDL type is derived from the field type given in the object specification. If the field is not mentioned in the object specification the field is not mapped to the SDL type.

Several levels of indirection are possible in doing this because field types can be specified by reference to some other object class. Also parameterisation can be used in the ASN.1 specification and has to be resolved before mapping to SDL is undertaken.

Example

Suppose that the following information object class definition is given in an ASN.1 module.

```
ADDRESS-CLASS-TEMPLATE::= CLASS {
&whichType    INTEGER(0..3),
&OptType
}
WITH SYNTAX {
        WHICH &whichType
        OPT &OptType
}
```

Suppose also that following information objects are given in the ASN.1 module.

```
gssi-object          ADDRESS-CLASS-TEMPLATE::=
{
        WHICH                   0
        OPT             Group-Short-Subscriber-Identity
}
gssi-ae-object       ADDRESS-CLASS-TEMPLATE::=
{
        WHICH                   1
        OPT             GSSI-AE
}
vgssi-object         ADDRESS-CLASS-TEMPLATE::=
{
        WHICH                   2
        OPT             Visitor-Group-Short-Subscriber-Identity
}
all-object           ADDRESS-CLASS-TEMPLATE::=
{
        WHICH                   3
        OPT             ALL-TYPE
}
```

For completeness the types referenced in object definitions are also specified.

```
GSSI-AE        ::= SEQUENCE
{
        gssi            Group-Short-Subscriber-Identity,
        ae              Address-Extension
}
ALL-TYPE       ::= SEQUENCE
{
        gssi            Group-Short-Subscriber-Identity,
        ae              Address-Extension,
        vgssi           Visitor-Group-Short-Subscriber-Identity
}
```

Suppose also that the objects are specified to form an object set.

```
Address-Class-Instance-Set      ADDRESS-CLASS-TEMPLATE::=
{ gssi-object | gssi-ae-object | vgssi-object | all-object }
```

The definitions above are used to specify a sequence that follows.

```
Group-Identity-Uplink ::= SEQUENCE
{
group-Identity-Address-Type    ADDRESS-CLASS-TEMPLATE.&whichType
({Address-Class-Instance-Set }),
opt             ADDRESS-CLASS-TEMPLATE.&OptType
({Address-Class-Instance-Set }{@.group-Identity-Address-Type })
}
```

Because the object set is mentioned in the sequence, this actually means that an SDL data type can be derived from the ASN.1 module for each object in the set.

```
value type gssi_object STRUCT
        group_Identity_Address_Type          <<package Predefined>> Integer constants (0);
        opt                  Group_Short_Subscriber_Identity;
endvalue type gssi_object;
/* */

value type gssi_ae_object STRUCT
        group_Identity_Address_Type          <<package Predefined>> Integer constants (1);
        opt                  GSSI_AE;
endvalue type gssi_ae_object;
/* */

value type vgssi_object STRUCT
        group_Identity_Address_Type          <<package Predefined>> Integer constants (2);
        opt                  Visitor_Group_Short_Subscriber_Identity;
endvalue type vgssi_object;
/* */

value type all_object STRUCT
        group_Identity_Address_Type          <<package Predefined>> Integer constants (3);
        opt                  ALL_TYPE;
endvalue type all_object;
```

6 Mapping of parameterized ASN.1 types

Recommendation X.683 [6] defines the way to parameterize ASN.1 types. All ASN.1: 1997 concepts can be parameterized. This feature allows the partial specification of types or values within an ASN.1 module with the specification being completed by the addition of the actual parameters at instantiation time.

Recommendation Z.100 defines an equivalent concept of formal context parameters.

The approach is that parameterized ASN.1 types are mapped to Z.100 types with formal context parameters allowing partial specifications to exist without actual parameters and formally analysed.

There are parameterized assignment statements corresponding to each of the assignment statements specified in Recommendations X.680 and X.681. The "ParameterizedAssignment" construct is:

```
ParameterizedAssignment ::=
        ParameterizedTypeAssignment |
        ParameterizedValueAssignment|
        ParameterizedValueSetTypeAssignment      |
        ParameterizedObjectClassAssignment|
        ParameterizedObjectAssignment        |
        ParameterizedObjectSetAssignment
```

The use of all parameterized assignments is supported within ASN.1 modules.

Parameterized types and values can be mapped to SDL as defined in 6.1 and 6.2.

The parameterized assignments that cannot be mapped to SDL types or values with context parameters may be used in ASN.1 modules to define other ASN.1 types or values that can be mapped to SDL as defined in clause 4.

6.1 Parameterized type assignment

ASN.1 Grammar

ParameterizedTypeAssignment ::=
typereference
ParameterList
"::="
Type

Model

The difference between ordinary and parameterized ASN.1 types is that **ParameterList** follows the **typereference** and formal parameters contained in **ParameterList** are used in the **Type** definition.

A **Type** defined in ASN.1 using parameters from the **ParameterList** is mapped to the appropriate SDL type (as defined in 4.2.1) provided that ASN.1 parameters are either value or type parameters. Such parameters are mapped to <formal context parameters> of the SDL type. ASN.1 type parameter is mapped to SDL <sort context parameter> and ASN.1 value parameter is mapped to SDL <synonym context parameter>. ASN.1 parameterized types having different parameters have to be instantiated in ASN.1 modules after which the resulting type or value can be mapped to SDL.

Example

The ASN.1 type definition

```
TemplateMessage {INTEGER : minSize, INTEGER : maxSize, IndicatorType } ::= SEQUENCE
{
        asp         INTEGER,
        pdu         OCTET STRING(SIZE(minSize..maxSize)),
        indicator   IndicatorType
}
```

is mapped to SDL type

value type TemplateMessage
<**synonym** minSize <<**package** Predefined>> Integer; **synonym** maxSize <<**package** Predefined>> Integer; **value type** IndicatorType>
asp Integer;
pdu <<**package** Predefined>>Octetstring (SIZE(minSize:maxSize));
indicator IndicatorType;
endvalue type;

6.2 Parameterized value assignment

ASN.1 Grammar

ParameterizedValueAssignment ::=
valuereference
ParameterList
Type
"::="
Value

Model

A **ParameterizedValueAssignment** is represented by a <synonym definition item> with items from the **ParameterList** mapped to its <formal context parameters>. For the parameters, the conditions mentioned in 6.1 apply.

Example

The ASN.1 value assignment

genericBirthdayGreeting { IA5String : name } IA5String ::= { "Happy birthday, ", name, "!!" }

is mapped to

synonym genericBirthdayGreeting <**synonym** name <<**package** Predefined>> IA5String >

<<**package** Predefined>> IA5String = 'Happy birthday,'//name//'!!';

6.3 Referencing ASN.1 parameterized definitions

Model

Parameterized types and values are used in ASN.1 to define simple ASN.1 types and values by providing an **ActualPameterList**. The resulting types and values can be mapped to SDL as defined in clause 3. If the parameterized definition was such that it was possible to map it to SDL, ASN.1 references to such definitions can be mapped to SDL instantiations of the type with context parameters so that elements of **ActualPameterList** are mapped to <actual context parameters>.

Example

The parameterized type used in the example in 6.1 can be used to define a simple ASN.1 as follows:

ActualMessage ::= TemplateMessage{10, 20, BOOLEAN}

This can be mapped to SDL type

value type **ActualMessage : TemplateMessage < 10, 20, <<**package **Predefined>> Boolean >**

The parameterized value genericBirthdayGreeting can be instantiated in ASN.1 in the following way:

greeting1 IA5String ::= genericBirthdayGreeting { "John" }, which can be mapped to SDL as

synonym greeting1 <<**package** Predefined>> IA5String = 'John'

6.4 Parameterized object definition together with an information object class

What follows is an example of the parameterized object definition together with an information object class and its mapping to SDL.

```
MESSAGE-PARAMETERS ::= CLASS {
        &maximum-priority-level             INTEGER,
        &maximum-message-buffer-size        INTEGER
}
WITH SYNTAX {
        THE MAXIMUM PRIORITY LEVEL IS           &maximum-priority-level
        THE MAXIMUM MESSAGE BUFFER SIZE IS  &maximum-message-buffer-size
}
Message-PDU { MESSAGE-PARAMETERS : param } ::= SEQUENCE {
        priority-level      INTEGER (0..param.&maximum-priority-level),
        message             BMPString (SIZE (0..param.&maximum-message-buffer-size))
}
```

```
my-message-parameters MESSAGE-PARAMETERS ::= {
        THE MAXIMUM PRIORITY LEVEL IS 10
        THE MAXIMUM MESSAGE BUFFER SIZE IS  2000
}
MY-Message-PDU ::= Message-PDU { my-message-parameters }
```

The resulting SDL type would be:

```
value type MY_Message_PDU STRUCT
        priority_level      <<package Predefined>> INTEGER (0..10);
        message     <<package Predefined>> BMPString (SIZE (0..2000));
end value type;
```

7 Additions to package Predefined

The following definitions shall be added to the package Predefined in order to support the combination of ASN.1 modules with SDL.

```
syntype NumericChar = Character constants
' ',   '0',  '1',  '2',  '3',  '4',  '5',  '6',
'7',   '8',  '9' endsyntype;
/* */

/*          NumericString sort          */
/*          Definition          */
value type NumericString
     inherits String < NumericChar > ( " = emptystring )
     adding
          operators ocs in nameclass
               "" ( ('0':'9') or """ or (' ') )+ "" -> this NumericString;
/* character strings of any length of any characters from a space ' ' to a '9' */
axioms
     for all c in NumericChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                                   ==> cs   == mkstring(c);
/* string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                                    ==> cs   == cs1 // cs2;
     ));
endvalue type NumericString;
/* */
syntype PrintableChar = Character constants
 ' ',   '0',  '1',  '2',  '3',  '4',  '5',  '6',
 '7',   '8',  '9',  'A',  'B',  'C',  'D',  'E',
 'F',   'G',  'H',  'I',   'J',  'K',  'L',  'M',
 'N',  'O',  'P',  'Q',  'R',  'S',  'T',  'U',
 'V',  'W', 'X',  'Y',  'Z',  'a',  'b',  'c',
 'd',  'e',  'f',  'g',  'h',  'i',   'j',   'k',
 'l',   'm',  'n',  'o',  'p',  'q',  'r',   's',
 't',   'u',  'v',  'w',  'x',  'y',  'z',  "",
 '(',   ')',   '+',  ',',   '-',   '.',   '/',   ':',
 '=',  '?'
 constants;
 /* */
```

```
/*        PrintableString sort          */
    /*    Definition              */
    value type PrintableString
        inherits String < PrintableChar > ( " = emptystring )
        adding
            operators ocs in nameclass
                                "" ( (' ':'&') or """ or ('(': '?') )+ "" -> this PrintableString;
    /* character strings of any length of any characters from a space ' ' to a '?' */
    axioms
        for all c in PrintableChar nameclass (
        for all cs, cs1, cs2 in ocs nameclass (
        spelling(cs) == spelling(c)                                    ==> cs   == mkstring(c);
    /* string 'A' is formed from character 'A' etc. */
        spelling(cs) == spelling(cs1) // spelling(cs2),
        length(spelling(cs2)) == 1                                     ==> cs   == cs1 // cs2;
        ));
    endvalue type PrintableString;
/* */
    syntype TeletexChar = Character constants
/* characters specified in X.680 clause 34.1 table 3 */ endsyntype;
/* */

/*        TeletexString sort            */
    /*     Definition             */
    value type TeletexString
        inherits String < TeletexChar > ( " = emptystring )
        adding
            operators ocs in nameclass
                /* characters specified in X.680 clause 34.1 table 3  */ -> this TeletexString;
    axioms
        for all c in TeletexChar nameclass (
        for all cs, cs1, cs2 in ocs nameclass (
        spelling(cs) == spelling(c)                                    ==> cs   == mkstring(c);
    /* string 'A' is formed from character 'A' etc. */
        spelling(cs) == spelling(cs1) // spelling(cs2),
        length(spelling(cs2)) == 1                                     ==> cs   == cs1 // cs2;
        ));
    endvalue type TeletexString;
    syntype VideotexChar = Character constants
/* characters specified in X.680 clause 34.1 table 3 */ endsyntype;
/* */

/*        VideotexString sort           */
    /*     Definition             */
    value type VideotexString
        inherits String < VideotexChar > ( " = emptystring )
        adding
            operators ocs in nameclass
            /* characters specified in X.680 clause 34.1 table 3 */ -> this VideotexString;
    axioms
        for all c in VideotexChar nameclass (
        for all cs, cs1, cs2 in ocs nameclass (
        spelling(cs) == spelling(c)                                    ==> cs   == mkstring(c);
    /* string 'A' is formed from character 'A' etc. */
        spelling(cs) == spelling(cs1) // spelling(cs2),
        length(spelling(cs2)) == 1                                     ==> cs   == cs1 // cs2;
        ));
    endvalue type VideotexString;
```

```
syntype IA5Char = Character endsyntype;

syntype IA5String = Charstring endsyntype;

value type GeneralChar
  literals /* All G and all C sets + SPACE + DELETE X.680 clause 34.1 table 3*/
     operators
          gchr      ( Integer ) -> this GeneralChar;
endvalue type;

value type UniversalChar
  literals /* see X.680 clause 34.6 */
     operators
          uchr      ( Integer ) -> this UniversalChar;
endvalue type;
/* */

/*        UniversalCharString sort              */

     /*     Definition              */

     value type UniversalCharString
          inherits String < UniversalChar > ( " = emptystring )
          adding
               operators ocs in nameclass
                                   /* see X.680 clause 34.6 */ -> this UniversalCharString;
     axioms
          for all c in UniversalChar nameclass (
          for all cs, cs1, cs2 in ocs nameclass (
          spelling(cs) == spelling(c)                                   ==> cs   == mkstring(c);
     /* string 'A' is formed from character 'A' etc. */
          spelling(cs) == spelling(cs1) // spelling(cs2),
          length(spelling(cs2)) == 1                                    ==> cs   == cs1 // cs2;
          ));
     endvalue type UniversalCharString;
/* */

/*        UTF8String sort                 */
     syntype UTF8String = UniversalCharString endsyntype;
/* */

/*        GeneralCharString sort                */
     /*      Definition              */
     value type GeneralCharString
          inherits String < GeneralChar > ( " = emptystring )
          adding
               operators ocs in nameclass
               /* All G and all C sets + SPACE + DELETE X.680 clause 34.1 table 3 */
                         -> this GeneralCharString;
     /* character strings of any length of any characters from a space ' ' to a '?' */
    axioms
          for all c in GeneralChar nameclass (
          for all cs, cs1, cs2 in ocs nameclass (
          spelling(cs) == spelling(c)                                   ==> cs   == mkstring(c);
     /* string 'A' is formed from character 'A' etc. */
          spelling(cs) == spelling(cs1) // spelling(cs2),
          length(spelling(cs2)) == 1                                    ==> cs   == cs1 // cs2;
          ));
```

```
    endvalue type GeneralCharString;
/* */
syntype GraphicChar = GeneralChar constants
/*All G+SPACE+DELETE as specified in X.680 clause 34.1 table 3 */
endsyntype;
/* */

/*          GraphicCharString sort                  */
    /*      Definition              */
    value type GraphicCharString
        inherits String < GraphicChar > ( " = emptystring )
        adding
            operators ocs in nameclass
            /* All G + SPACE + DELETE as specified in X.680 clause 34.1 table 3*/
     -> this GraphicCharString;
    axioms
        for all c in GraphicChar nameclass (
        for all cs, cs1, cs2 in ocs nameclass (
        spelling(cs) == spelling(c)                                 ==> cs   == mkstring(c);
        spelling(cs) == spelling(cs1) // spelling(cs2),
        length(spelling(cs2)) == 1                                  ==> cs   == cs1 // cs2;
        ));
    endvalue type GraphicCharString;

syntype VisibleChar = Character constants
/* characters specified in X.680 clause 34.1 table 3 */
endsyntype;
/* */

/*      VisibleString sort              */
    /*      Definition              */
    value type VisibleString
        inherits String < VisibleChar > ( " = emptystring )
        adding
            operators ocs in nameclass
                /* characters specified in X.680 clause 34.1 table 3 */
         -> this VisibleString;
    axioms
        for all c in VisibleChar nameclass (
        for all cs, cs1, cs2 in ocs nameclass (
        spelling(cs) == spelling(c)                                 ==> cs   == mkstring(c);
    /* string 'A' is formed from character 'A' etc. */
        spelling(cs) == spelling(cs1) // spelling(cs2),
        length(spelling(cs2)) == 1                                  ==> cs   == cs1 // cs2;
        ));
    endvalue type VisibleString;

syntype BMPChar = UniversalChar CONSTANTS /* see X.680 clause 34.12 */
endsyntype;
/* */

/*      BMPCharString sort                  */
    /*   Definition             */
    value type BMPCharString
        inherits String < BMPChar > ( " = emptystring )
        adding
            operators ocs in nameclass
                        /* see X.680 clause 34.12 */ -> this BMPCharString;
```

```
    axioms
        for all c in BMPChar nameclass (
        for all cs, cs1, cs2 in ocs nameclass (
        spelling(cs) == spelling(c)                                  ==> cs   == mkstring(c);
    /* string 'A' is formed from character 'A' etc. */
        spelling(cs) == spelling(cs1) // spelling(cs2),
        length(spelling(cs2)) == 1                                   ==> cs   == cs1 // cs2;
        ));
    endvalue type BMPCharString;
/* */

    value type NULL
    literals NULL

    endvalue type;
```

ITU-T RECOMMENDATIONS SERIES

Series A	Organization of the work of the ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks and open system communications
Series Y	Global information infrastructure
Series Z	**Languages and general software aspects for telecommunication systems**

18097

Printed in Switzerland
Geneva, 2000