UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.105

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(10/2001)

SÉRIE Z: LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS DES SYSTÈMES DE TÉLÉCOMMUNICATION

Techniques de description formelle – Langage de description et de spécification (SDL)

Langage SDL combiné avec des modules ASN.1 (SDL/ASN.1)

Recommandation UIT-T Z.105

RECOMMANDATIONS UIT-T DE LA SÉRIE Z

LANGAGES ET ASPECTS GÉNÉRAUX LOGICIELS DES SYSTÈMES DE TÉLÉCOMMUNICATION

TECHNIQUES DE DESCRIPTION FORMELLE	
Langage de description et de spécification (SDL)	**Z.100–Z.109**
Application des techniques de description formelle	Z.110–Z.119
Diagrammes des séquences de messages	Z.120–Z.129
LANGAGES DE PROGRAMMATION	
CHILL: le langage de programmation de l'UIT-T	Z.200–Z.209
LANGAGE HOMME-MACHINE	
Principes généraux	Z.300–Z.309
Syntaxe de base et procédures de dialogue	Z.310–Z.319
LHM étendu pour terminaux à écrans de visualisation	Z.320–Z.329
Spécification de l'interface homme-machine	Z.330–Z.399
QUALITÉ DES LOGICIELS DE TÉLÉCOMMUNICATION	Z.400–Z.499
MÉTHODES DE VALIDATION ET D'ESSAI	Z.500–Z.599

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

Recommandation UIT-T Z.105

Langage SDL combiné avec des modules ASN.1 (SDL/ASN.1)

Résumé

La présente Recommandation définit la façon dont des modules ASN.1 (*abstract syntax notation one*) peuvent être utilisés en combinaison avec le langage SDL (*specification and description language*). Le texte remplace les mappages sémantiques ASN.1 sur SDL définis dans la Rec. UIT-T Z.105 (1999). L'emploi de la notation ASN.1 intégrée dans le langage SDL précédemment définie dans la Rec. UIT-T Z.105 (1995) n'est pas défini dans la présente Recommandation.

La principale application de la présente Recommandation est la spécification de systèmes de télécommunication. L'emploi combiné SDL/ASN.1 offre un moyen cohérent de spécifier la structure et le comportement de systèmes de télécommunication avec les données, les messages et le codage de messages qu'utilisent ces systèmes.

Source

La Recommandation Z.105 de l'UIT-T, révisée par la Commission d'études 10 (2001-2004) de l'UIT-T, a été approuvée le 29 octobre 2001 selon la procédure définie dans la Résolution 1 de l'AMNT.

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'UIT. Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

L'Assemblée mondiale de normalisation des télécommunications (AMNT), qui se réunit tous les quatre ans, détermine les thèmes d'étude à traiter par les Commissions d'études de l'UIT-T, lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution 1 de l'AMNT.

Dans certains secteurs des technologies de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

NOTE

Dans la présente Recommandation, l'expression "Administration" est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

TABLE DES MATIÈRES

Introduction

- **Objectif**

La présente Recommandation définit la façon dont des modules en notation de systaxe abstracte numéro un (ASN.1, *abstract syntax notation number one*) peuvent être utilisés en combinaison avec le langage de description et de spécification (SDL, *specification and description language*). L'objectif est de permettre de décrire la structure et le comportement des systèmes en langage SDL et les paramètres des messages échangés en notation ASN.1. La présente Recommandation définit un mappage de constructions ASN.1 dans des constructions SDL préexistantes et ne contient qu'une légère extension de la Rec. UIT-T Z.100 de manière à pouvoir utiliser des modules ASN.1.

- **Portée**

La présente Recommandation donne une définition sémantique pour la combinaison du langage SDL et de modules ASN.1. Elle donne un mappage des données ASN.1 définies dans un module sur les constructions SDL correspondantes définies dans la Rec. UIT-T Z.100 [1], y compris les opérateurs qui peuvent être appliqués aux données ASN.1. Les items de données ASN.1 peuvent alors être utilisés dans le langage SDL (en utilisant la notation SDL).

L'utilisation du langage SDL avec notation ASN.1 incorporée est définie dans la Rec. UIT-T Z.107 [2].

- **Application**

Le principal domaine d'application de la présente Recommandation est la spécification de systèmes de télécommunication. L'usage combiné du langage SDL et de la notation ASN.1 permet de spécifier de manière cohérente la structure et le comportement de systèmes de télécommunication, ainsi que les données, les messages et le codage de messages que ces systèmes utilisent.

NOTE – Dans la présente Recommandation, le terme "spécification" inclut la définition des prescriptions dans une norme, dans une Recommandation ou un document d'approvisionnement ainsi que la description d'une implémentation.

Une spécification est conforme à la présente Recommandation si et seulement si elle respecte les règles grammaticales syntaxiques et sémantiques relatives au langage technique formel défini dans la Recommandation (qui inclut les langages ASN.1 et SDL cités en référence). La conformité implique que toute interprétation éventuellement dynamique de la spécification soit conforme aux règles du langage. Une spécification qui utilise des extensions du langage n'est pas conforme.

Un outil ne prend pas entièrement en charge le langage s'il rejette certaines constructions du langage ou si une interprétation statique ou dynamique d'une spécification n'est pas conforme à la sémantique du langage.

- **Etat/Stabilité**

La présente Recommandation remplace les mappages sémantiques de la notation ASN.1 dans le langage SDL définis dans la Rec. UIT-T Z.105 (1999). L'utilisation du langage SDL avec notation ASN.1 incorporée qui était définie dans la Rec. UIT-T Z.105 (1995) n'est pas définie dans la présente Recommandation.

En cas de modification des Recs. UIT-T X.680 [3], X.681 [4], X.682 [5] et X.683 [6] ou Z.100 [1], il sera peut-être nécessaire d'apporter des modifications à la présente Recommandation.

La présente Recommandation constitue un manuel de référence complet qui décrit la combinaison du langage SDL et de modules ASN.1.

- **Travaux associés**

– Rec. UIT-T Z.100 (1999), *langage de description et de spécification (SDL)*.

– Rec. UIT-T X.680 (1997), *ASN.1: notation de syntaxe abstraite numéro un*.

– Rec. UIT-T X.681 (1997), *ASN.1: spécification des objets informationnels*.

– Rec. UIT-T X.682 (1997), *ASN.1: spécification des contraintes*.

– Rec. UIT-T X.683 (1997), *ASN.1: paramétrage des spécifications de la notation de syntaxe abstraite numéro 1*.

– Rec. UIT-T Z.107 (1999), *langage SDL avec notation ASN.1 incorporée*.

Recommandation UIT-T Z.105

Langage SDL combiné avec des modules ASN.1 (SDL/ASN.1)

1 Domaine d'application

La présente Recommandation définit la façon dont des modules ASN.1 peuvent être utilisés en combinaison avec le langage SDL. Des modules ASN.1 sont importés dans des descriptions SDL de telle manière que des définitions de données ASN.1 soient mappées sur une représentation SDL interne utilisant des constructions SDL équivalentes et formant avec le reste de la description SDL une spécification complète.

Le langage SDL permet de spécifier et de décrire des systèmes de télécommunication. Il possède des concepts pour:

- structurer des systèmes;
- définir le comportement de systèmes;
- définir les données utilisées par les systèmes.

La notation ASN.1 est un langage qui permet de définir des données. A cette notation sont associées des règles de codage qui définissent la façon dont les valeurs ASN.1 sont transférées sous forme de flux binaires en cours de communication.

1.1 Objectif

La combinaison du langage SDL et de la notation ASN.1 constitue un moyen cohérent pour spécifier la structure et le comportement de systèmes de télécommunication, ainsi que les données, les messages et le codage des messages que ces systèmes utilisent. La structure et le comportement peuvent être décrits au moyen du langage SDL; les données et les messages peuvent l'être au moyen de la notation ASN.1. Le codage de ces messages peut être décrit par référence aux règles de codage applicables définies pour la notation ASN.1.

La présente Recommandation prend en compte la pleine utilisation du langage SDL (y compris des types de données).

1.2 Caractéristiques de la combinaison du langage SDL et de modules ASN.1

Les systèmes décrits en langage SDL combiné avec des modules ASN.1 ont les caractéristiques suivantes:

- la structure et le comportement sont définis au moyen de concepts SDL;
- les paramètres des signaux sont définis par des types ASN.1;
- les données utilisées dans des signaux sont définies par des définitions de types ASN.1;
- les données internes peuvent être définies par des types ASN.1 ou par des sortes SDL;
- le codage des valeurs de données définies en ASN.1 peut être défini par référence aux règles de codage applicables. Le codage est hors du domaine d'application de la présente Recommandation.

1.3 Notation ASN.1 pouvant être utilisée en combinaison avec le langage SDL

L'utilisation de la notation ASN.1 telle que définie dans les Recs. UIT-T X.680, X.681, X.682 et X.683 est prise en compte en combinaison avec le langage SDL. Toutefois, certaines constructions ASN.1 n'ont pas pu être mappées en langage SDL (ou, tout du moins, le mappage en langage SDL

n'a pas été identifié et spécifié dans la présente Recommandation). Celles-ci apparaîtront dans des paquetages ASN.1 utilisés comme source de transformation. Au cours de la transformation en langage SDL, ces constructions, qui sont traitées en fait comme si elles n'étaient pas présentes, ne devraient pas empêcher la transformation d'autres constructions. Constituées du marqueur d'extension et du marqueur d'exception, ces constructions, définies dans la Rec. UIT-T X.680, peuvent être présentes dans la notation ASN.1 mais sont ignorées lors de la transformation en langage SDL. Des parties de la grammaire ASN.1 (1997) concernant les marqueurs d'extension et d'exception ne sont plus utilisées dans la présente Recommandation. Certaines constructions ASN.1 ne sont jamais transformées telles quelles en langage SDL, mais contiennent des informations pouvant être utiles pour leur transformation. Comme exemples importants de ces constructions, on peut citer les contraintes relationnelles définies dans la Rec. UIT-T X.682, les classes d'objets ou les ensembles d'objets.

L'utilisation du langage SDL tel que défini dans la Rec. UIT-T Z.100 [1] est prise en compte.

Les modules ASN.1, utilisés pour la transformation en langage SDL, peuvent aussi être employés pour la production de codeurs et de décodeurs, à condition que les règles de codage soient définies. La spécification des données SDL dérivée des modules ASN.1 ne doit pas être utilisée à cette fin, des informations importantes relatives au codage pouvant être perdues au cours de la transformation en langage SDL.

1.4 Structure de la présente Recommandation

La présente Recommandation n'est pas autonome: le mappage qu'elle définit est fondé sur la Rec. UIT-T Z.100 et les Recs. UIT-T X.680, X.681, X.682 et X.683. Le langage défini dans la Rec. UIT-T Z.100 est applicable, sauf que la règle de production <package> est étendue afin de pouvoir utiliser directement des modules ASN.1. La présente Recommandation est structurée comme suit:

Le paragraphe 3 définit les modifications apportées à la Rec. UIT-T Z.100 afin d'incorporer des modules ASN.1.

Le paragraphe 4 définit le mappage des types et valeurs ASN.1 selon la Rec. UIT-T X.680 sur la Rec. UIT-T Z.100 afin d'incorporer des types de données et des valeurs ASN.1.

Le paragraphe 5 définit le mappage des types ASN.1 définis qui utilisent des objets d'information, des classes d'objets d'information et des ensembles d'objets d'information. Il aborde également l'utilisation des constructions conformes à la Rec. UIT-T X.682.

Le paragraphe 6 définit le mappage de types ASN.1 paramétrés sur des données de la Rec. UIT-T Z.100 afin d'incorporer des types de données ASN.1 paramétrés.

Le paragraphe 7 définit les adjonctions du paquetage "Predefined" nécessaires pour prendre en charge l'utilisation de la notation ASN.1.

1.5 Conventions utilisées dans la présente Recommandation

Fondamentalement, les conventions de la Rec. UIT-T Z.100 sont applicables: par exemple, les mots clés apparaissent en caractères gras minuscules et les noms prédéfinis commencent par une majuscule. Dans les exemples reposant sur l'ASN.1 toutefois, les conventions de l'ASN.1 sont utilisées afin de respecter les règles ASN.1 et faciliter la lecture par des utilisateurs de l'ASN.1: par exemple les mots clés sont écrits en majuscules.

Des références à des documents originaux sont indiquées pour les productions grammaticales ASN.1, mais certaines d'entre elles sont reproduites, par souci de clarté, aux endroits où cela a été jugé nécessaire. En cas de divergence, ce sont les productions ASN.1 originales qui prévalent.

2 Références normatives

La présente Recommandation se réfère à certaines dispositions des Recommandations UIT-T et textes suivants qui, de ce fait, en sont partie intégrante. Les versions indiquées étaient en vigueur au moment de la publication de la présente Recommandation. Toute Recommandation ou tout texte étant sujet à révision, les utilisateurs de la présente Recommandation sont invités à se reporter, si possible, aux versions les plus récentes des références normatives suivantes. La liste des Recommandations de l'UIT-T en vigueur est régulièrement publiée.

[1] Recommandation UIT-T Z.100 (1999), SDL: *langage de description et de spécification*.

[2] Recommandation UIT-T Z.107 (1999), *Langage SDL avec notation ASN.1 incorporée*.

[3] Recommandation UIT-T X.680 (1997) | ISO/CEI 8824-1:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification de la notation de base,* comprenant les Amendements 1 et 2 (1999) et le Corrigendum 1 (1999).

[4] Recommandation UIT-T X.681 (1997) | ISO/CEI 8824-2:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des objets informationnels,* comprenant l'Amendement 1 (1999) et le Corrigendum 1 (1999).

[5] Recommandation UIT-T X.682 (1997) | ISO/CEI 8824-3:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: spécification des contraintes*.

[6] Recommandation UIT-T X.683 (1997) | ISO/CEI 8824-4:1998, *Technologies de l'information – Notation de syntaxe abstraite numéro un: paramétrage des spécifications de la notation de syntaxe abstraite numéro un,* comprenant l'Amendement 1 (1999).

[7] Recommandation UIT-T X.690 (1997) | ISO/CEI 8825-1:1998, *Technologies de l'information – Règles de codage ASN.1: spécification des règles de codage de base, des règles de codage canoniques et des règles de codage distinctives*.

3 Paquetage *(package)*

Grammaire ASN.1

`ModuleDefinition` est défini dans le § 12.1 de [3].

```
ModuleDefinition ::=
              ModuleIdentifier
              DEFINITIONS
              TagDefault
              "::="
              BEGIN
              ModuleBody
              END
ModuleIdentifier ::=
              modulereference
              DefinitiveIdentifier
DefinitiveIdentifier ::=
              "{" DefinitiveObjIdComponentList "}" | empty
```

Modèle

La production <package> est étendue comme suit:

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

où ModuleDefinition est un non-terminal défini dans la Rec. UIT-T X.680:1997.

Une définition de module <module definition> a la même signification qu'une définition de paquetage <package definition> où:

- l'identificateur de module **`ModuleIdentifier`** (sans aucun identificateur définitif **`DefinitiveIdentifier`**) correspond au nom de paquetage <package name>;
- l'importation **`Imports`** correspond aux clauses de référence de paquetage <package use clause>;
- l'exportation **`Exports`** correspond à l'interface <interface>.

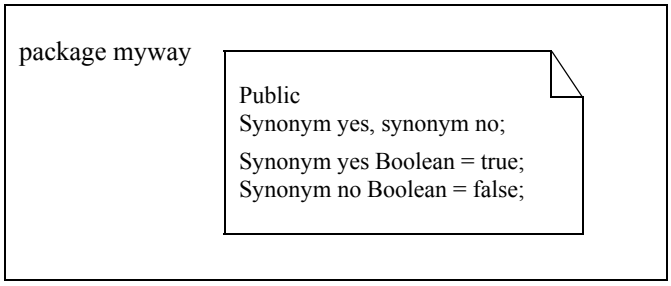
Un paquetage ASN.1 est transformé en son équivalent SDL, avant d'être considéré comme un paquetage et avant toutes transformations Z.100. Dans cette transformation, les noms sont transformés en identificateurs entièrement qualifiés lorsque le SDL nécessite ou autorise un identificateur plutôt qu'un nom. Toutefois, par souci de concision, cette qualification complète est souvent omise des exemples donnés dans la présente Recommandation.

Exemple

La définition de module ASN.1:

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

équivaut à:



De même, lorsque le paquetage est utilisé dans l'importation **`imports`** d'un autre paquetage:

```
IMPORTS yes FROM myway;
```

Cela équivaut à la clause <package reference clause> suivante:

use myway/yes;

NOTE – Etant donné que le langage SDL ne prend pas en charge de valeurs d'identificateur d'objet pour l'identification des paquetages, les modules ASN.1 ayant la même référence de module **`modulereference`**, mais des identificateurs définitifs **`DefinitiveIdentifiers`**, différents seront susceptibles de causer des problèmes de résolution de nom.

4 Définition et utilisation de données

Les différentes définitions de l'utilisation de données sont décrites comme suit:

Grammaire ASN.1	Définition des règles de production grammaticales représentant la construction à représenter en langage SDL
Modèle	Description des transformations des différentes parties de la grammaire ASN.1 en productions SDL. Cette partie fait référence à la grammaire SDL, représentée par <SDL grammar rule>, et à la grammaire ASN.1, représentée par **`ASN1GrammarRule`**.

4.1 Mappage de noms

Grammaire ASN.1

Le tiret ("-") est admis dans les noms ASN.1. S'il était utilisé en langage SDL, il serait considéré comme l'opérateur de soustraction.

Modèle

Les noms ASN.1 contenant des tirets sont mappés sur des noms SDL lexicalement similaires, les tirets étant toutefois convertis en caractères de soulignement.

Exemple

Le nom ASN.1 **`my-example-name`** est mappé sur le nom my_example_name en langage SDL.

4.2 Définitions de variables et de données

4.2.1 Affectation de type (TypeAssignment)

Grammaire ASN.1

`TypeAssignment` est défini dans le § 15.1 de [3].

```
TypeAssignment     ::=  typereference "::=" Type
```

Modèle

Si le type **`Type`** est une référence de type **`typereference`**, l'affectation **`TypeAssignment`** équivaut à une définition <syntype definition> contenant uniquement l'équivalent SDL du type **`Type`**.

Si le type **`Type`** est un type contraint **`constrainedType`**, l'affectation **`TypeAssignment`** équivaut à une définition <syntype definition> contenant uniquement l'équivalent SDL de la contrainte **`Constraint`**.

Si le type **`Type`** n'est ni une référence **`typereference`** ni un type **`constrainedType`**, l'affectation **`TypeAssignment`** est représentée par une définition <partial type definition> où l'expression <properties expression> est vide et où les paramètres <formal context parameters> sont omis.

Exemple

L'affectation de type ASN.1:

```
Mytype ::= AnotherType -- typereference
```

équivaut à:

syntype Mytype = AnotherType **endsyntype** Mytype; /* la qualification complète est omise ici. */

L'affectation de type ASN.1:

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

équivaut à:

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

L'affectation de type ASN.1:

```
Integerlist ::= SEQUENCE OF INTEGER
```

équivaut à:

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

}

4.2.2 Affectation de valeur (ValueAssignment)

Grammaire ASN.1

ValueAssignment est défini dans le § 15.2 de [3].

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

Modèle

Une affectation **ValueAssignment** est représentée par un item <synonym definition item>.

Exemple

La définition ASN.1:

```
yes BOOLEAN ::= TRUE
```

équivaut à:

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

4.3 Expressions de type

4.3.1 Séquence (Sequence)

Grammaire ASN.1

SequenceType est défini dans le § 24.1 de [3]. **SetType** est défini dans le § 26.1 de [3].

```
SequenceType ::=
                  SEQUENCE "{" "}" |
                  SEQUENCE "{" ExtensionAndException "}" |
                  SEQUENCE "{" ComponentTypeLists "}"

ExtensionAndException ::=   "..." | "..." ExceptionSpec

ComponentTypeLists ::=
              RootComponentTypeList |
              RootComponentTypeList   ","  ExtensionAndException |
              RootComponentTypeList   ","  ExtensionAndException  ","
              AdditionalComponentTypeList |
              ExtensionAndException   ","  AdditionalComponentTypeList
RootComponentTypeList       ::=  ComponentTypeList
AdditionalComponentTypeList ::=  ComponentTypeList
ComponentTypeList  ::=
              ComponentType   |
              ComponentTypeList "," ComponentType
ComponentType ::=
              NamedType    |
              NamedType OPTIONAL   |
              NamedType DEFAULT Value   |
              COMPONENTS OF Type
NamedType     ::=  identifier Type
```

Modèle

Un type **SequenceType** est représenté par une définition <structure definition> contenant un champ <field> pour chaque type **NamedType** du type **SequenceType**. Le champ <field> contient un nom <field name>, qui équivaut à l'identificateur **identifier** ASN.1 du type **NamedType**, et une sorte <field sort>, qui correspond au type **Type** transformé en identificateur <sort identifier> SDL.

Si le type **ComponentType** contenant le type **NamedType** est facultatif **OPTIONAL**, le champ SDL comporte le mot clé **optional**.

Si le type **ComponentType** contenant le type **NamedType** a une valeur par défaut **DEFAULT Value**, le champ SDL comporte le mot clé **default** et la valeur est transformée en expression <constant expression> après **default**.

Un type **ComponentType** qui est **COMPONENTS OF Type** est représenté par une liste de champs <field> ordonnés, un pour chaque champ associé au type **Type**. Ces champs sont insérés dans la position de **COMPONENTS OF Type** dans l'ordre dans lequel les champs figurent dans le type **Type**.

Les occurrences de **ExtensionAndException** dans **SequenceType** sont ignorées dans la transformation.

Exemple

Le type ASN.1:

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

équivaut à:

value type S
{
    struct
    a <<**package** Predefined>> Integer;
    b <<**package** Predefined>> IA5String optional;
    c <<**package** Predefined>> PrintableString default 'd';
}

NOTE 1 – Il n'y a aucune distinction entre l'utilisation du mot clé **SEQUENCE** et celle du mot clé **SET**, ce qui constitue un assouplissement par rapport à la Rec. UIT-T X.680.

NOTE 2 – Dans la présente Recommandation, il n'est pas nécessaire de prévoir des étiquettes pour distinguer les différentes composantes d'un même type: on suppose qu'il se produit un étiquetage automatique ASN.1.

4.3.2 Séquence-de (Sequenceof)

Grammaire ASN.1

SequenceOfType est défini dans le § 25.1 de [3].

```
SequenceOfType::= SEQUENCE OF Type
```

Modèle

Le fait de spécifier un type **SequenceOfType** revient à spécifier la sorte prédéfinie String comportant la transformée SDL du type Type comme premier paramètre <actual context parameter> et le nom Emptystring défini comme le nom littéral de la chaîne vide.

Si une contrainte de taille ASN.1 est spécifiée pour le type **Type**, le type **SequenceOfType** est un syntype comportant la contrainte de taille transformée comme condition <range condition> (voir § 4.4). La sorte mère du syntype est le type **SequenceOfType** sans la contrainte de taille ASN.1. Cette sorte mère a un nom implicite et unique et elle est définie dans la plus proche unité de portée englobant l'occurrence du type **SequenceOfType**.

Exemple

La définition ASN.1:

```
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)
}
syntype S2 = <<**package** Predefined>> Integer **constants** (0..9) endsyntype;
syntype phonenumber = S1 **constants** size (8) endsyntype phonenumber;

4.3.3 Choix (Choice)

Grammaire ASN.1

ChoiceType est défini dans le § 28.1 de [3].

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

AlternativeTypeLists          ::=
                    RootAlternativeTypeList |
                    RootAlternativeTypeList "," ExtensionAndException |
                    RootAlternativeTypeList "," ExtensionAndException  ","
                                AdditionalAlternativeTypeList
RootAlternativeTypeList       ::=  AlternativeTypeList
AdditionalAlternativeTypeList ::=  AlternativeTypeList
AlternativeTypeList           ::=
                    NamedType |
                    AlternativeTypeList "," NamedType
```

Modèle

Un type **ChoiceType** est représenté par une définition <choice definition> contenant un champ <field> pour chaque type **NamedType** du type **ChoiceType**.

Les occurrences de **ExtensionAndException** dans **ChoiceType** sont ignorées dans la transformation.

Exemple

Le type choix ASN.1:

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

équivaut à:

value type C Choice
{
a <<**package** Predefined>> Integer;
b <<**package** Predefined>> Real;
}

4.3.4 Enuméré (Enumerated)

Grammaire ASN.1

EnumeratedType est défini dans le § 19.1 de [3].

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

Enumerations          ::=
              RootEnumeration |
              RootEnumeration  ","   "..." |
              RootEnumeration  ","   "..."   ","  AdditionalEnumeration

RootEnumeration       ::=   Enumeration

AdditionalEnumeration ::=   Enumeration

Enumeration           ::=   EnumerationItem | EnumerationItem "," Enumeration

EnumerationItem       ::=   identifier | NamedNumber

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

Modèle

Un type **EnumeratedType** est représenté par une définition <partial type definition> dans laquelle <properties expression> est vide et <formal context parameters> est omis. Pour chaque item **EnumerationItem**, l'identificateur **identifier** est transformé en une signature <literal signature> qui comporte le même nom que l'item **EnumerationItem**. Si l'item **EnumerationItem** contient un nombre **SignedNumber** (ou une valeur **DefinedValue**), le nom <literal name> de la signature <literal signature> est suivi de la transformée SDL du nombre **SignedNumber** (respectivement de la valeur **DefinedValue**).

Les instances de "…" dans **EnumeratedType** sont ignorées dans la transformation en SDL.

La définition:

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

équivaut à:

value type colours {
    literals blue = 3, red, yellow = 0
}

4.3.5 Dénomination d'entier et de bit (Integer and BitNaming)

Grammaire ASN.1

IntegerType est défini dans le § 18.1 de [3]. **BitStringType** est défini dans le § 21.1 de [3].

```
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 ")"
```

Modèle

Le fait de spécifier un type **`IntegerType`** avec une liste **`NamedNumberList`** (ou un type **`BitStringType`** avec une liste **`NamedBitList`**) revient à spécifier une définition <synonym definition> dans la plus proche unité de portée englobante avec un item <synonym definition item> pour chaque nombre **`NamedNumber`** (respectivement pour chaque liste **`NamedBitList`**). L'identificateur **`identifier`** du nombre **`NamedNumber`** (respectivement du bit **`NamedBit`**) est transformé en nom <synonym name>. La sorte <sort> de l'item <synonym definition item> est l'entier <<**package** Predefined>>Integer dans le cas d'un nombre **`NamedNumber`** et le bit <<**package** Predefined>>Bit dans le cas d'un bit **`NamedBit`**. Le nombre **`SignedNumber`** ou la valeur **`DefinedValue`** ou le nombre **`number`** du nombre **`NamedNumber`** ou de la liste **`NamedBitList`** est utilisé comme expression <constant expression> de l'item <synonym definition item>.

Exemple

La définition ASN.1:

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

équivaut à:

value type standards {
inherits
<< **package** Predefined >> String < <<**package** Predefined>> Integer> ("= EmptyString)
}
synonym z100 Integer = 0;
synonym x680 Integer = 1,
synonym z10x Integer = 2;

4.3.6 Intervalle de Valeurs (ValueRange)

Grammaire ASN.1

`Subrange` est défini dans le § 48.4.1 de [3].

```
ValueRange ::= LowerEndpoint ".." UpperEndpoint
```

Modèle

Le fait de spécifier une restriction d'intervalle de valeurs ASN.1 revient à spécifier la sorte <sort> contenue et à ajouter la représentation de la restriction d'intervalle de valeurs ASN.1 après le mot clé **constants** dans la production <syntype>.

Exemple

La définition ASN.1:

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

équivaut à:
syntype S = <<**package** Predefined>> Integer **constants** (0..5, 10) endsyntype S;

La procédure de détermination de la condition d'intervalle <range condition> est décrite ci-dessous.

4.3.7 Type Chaine de Bits (BitString)

Grammaire ASN.1

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

Modèle

Le type ASN.1 **`BitStringType`** est mappé en langage SDL sur la chaîne <<**package** Predefined>> Bitstring.

4.3.8 Type Chaine d'octets (OctetString)

Grammaire ASN.1

```
OctetStringType ::= OCTET STRING
```

Modèle

Le type ASN.1 **`OctetStringType`** est mappé en langage SDL sur la chaîne **<<package** Predefined**>>** Octetstring.

4.3.9 Type Ensemble de (Setof)

Grammaire ASN.1

`SetOfType` est défini dans le § 27.1 de [3].

```
SetOfType          ::=  SET OF Type
```

Modèle

Le fait de spécifier un type **`SetOfType`** revient à spécifier la sorte abstraite <<**package** Predefined>> comportant la transformée SDL du type **`Type`** comme premier paramètre <actual context parameter> et le nom Emptybag défini comme le nom de littéral du sac vide.

Si une contrainte de taille ASN.1 est spécifiée pour le type **`Type`**, le type **`SetOfType`** est un syntype comportant la contrainte de taille transformée comme condition <range condition> (voir § 4.4). La sorte mère du syntype est le type **`SetOfType`** sans la contrainte de taille ASN.1. Cette sorte mère a un nom implicite et unique et elle est définie dans la plus proche unité de portée englobant l'occurrence du type **`SetOfType`**.

4.4 Condition d'intervalle

Modèle

Une condition d'intervalle définit un ensemble de valeurs. Elle sert à définir un syntype. Une sorte mère lui est associée: c'est la sorte qui est spécifiée dans la définition du syntype. Une valeur fait partie de l'ensemble de valeurs si l'opérateur désigné par l'identificateur d'opérateur renvoie la valeur "true" lorsqu'il est appliqué à la valeur.

L'identificateur d'opérateur pour une condition d'intervalle donnée est alors défini par:

value type A

operators o: S -> Boolean;

/* oú o est derivé de la syntaxe concrète ASN.1 comme cele est expliqué ci-dessous */

endvalue type A;

Chaque intervalle Range de la condition d'intervalle ASN.1 contribue aux propriétés de l'opérateur définissant l'ensemble de valeurs:

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

Si un syntype est spécifié sans condition d'intervalle, l'opérateur renvoie "true".

Dans l'explication suivante de la manière dont chaque intervalle Range contribue au résultat de l'opérateur, V désigne la valeur de l'argument. Chaque contribution doit être bien formée, ce qui signifie que les opérateurs utilisés doivent exister avec une signature appropriée pour le contexte.

- Si aucun des mots clés **MIN** et **MAX** n'est spécifié dans un intervalle **ClosedRange**, la contribution de celui-ci est la suivante:

  E1 rel1 V and V rel2 E2

  où E1 est la valeur de **LowerEndValue** et E2 est la valeur de **UpperEndValue**.

  Si "<" est spécifié pour la valeur **LowerEndValue**, rel1 est l'opérateur "<", sinon c'est l'opérateur "<=".

  Si "<" est spécifié pour la valeur **UpperEndValue**, rel2 est l'opérateur "<", sinon c'est l'opérateur "<=".

  Si le mot clé **MIN** est spécifié et le mot clé **MAX** n'est pas spécifié, la contribution de l'intervalle **Range** est la suivante:

  V rel2 E2

  Si le mot clé **MAX** est spécifié et le mot clé **MIN** n'est pas spécifié, la contribution de l'intervalle **Range** est la suivante:

  E1 rel1 V

  Si les deux mots clés **MIN** et **MAX** sont spécifiés, l'opérateur renvoie toujours "true".

- La contribution d'un type **ContainedSubType** est la suivante:

  o1(V)

  où o1 est l'opérateur implicite définissant l'ensemble de valeurs pour le type **Type** mentionné dans le type **ContainedSubType.**

- La contribution d'une contrainte **SizeConstraint** est la suivante:

  o1(length(V))

  où o1 est l'opérateur implicite définissant l'ensemble de valeurs pour la condition <range condition> mentionnée dans la contrainte **SizeConstraint**.

- La contribution des contraintes **InnerTypeConstraints** est l'une des suivantes:

  if length(V) = 0 **then** true **else** o1(first(V)) **and** o(Substring(V,2,length(V)-1)) **fi**; ou

  if length(V) = 0 **then** true **else** o1(take(V)) **and** o(del(take(V), V)) **fi**

  selon ce qui est approprié pour la sorte de V. o est l'opérateur implicite auquel les contraintes **InnerTypeConstraints** contribuent et o1 est l'opérateur implicite pour l'intervalle **Range** spécifié dans les contraintes **InnerTypeConstraints**.

  Les contraintes **InnerTypeConstraints** ont une contribution pour chaque contrainte **NamedConstraint** contenue qui spécifie des contraintes du champ (voir § 4.2.1) désigné par l'identificateur **Identifier** de la sorte mère.

  On ajoute le mot clé **PRESENT** aux contraintes **NamedConstraints** qui n'ont pas de mot clé de fin (**PRESENT**, **ABSENT** ou **OPTIONAL**) et on ajoute des contraintes **NamedConstraints** de la forme **Identifier ABSENT** pour tous les champs (c'est-à-dire les identificateurs **Identifiers**) non mentionnés explicitement dans une contrainte **NamedConstraint**. Les contraintes **NamedConstraints** sont ajoutées aux contraintes **InnerTypeConstraints** avant de déterminer les contributions de chaque contrainte **NamedConstraint**.

  Si un intervalle **Range** est spécifié pour une contrainte **NamedConstraint**, la contribution est la suivante:

  E and if FPresent(V) then o1(V) else true fi

où E est la contrainte de présence pour le champ, F (tiré du nom d'opérateur FPresent) est le nom du champ facultatif et o1 est l'opérateur implicite pour l'intervalle **`Range`**. Si l'intervalle **`Range`** est omis, la contribution correspond uniquement à la contrainte de présence E.

La contrainte de présence pour un champ F est:

FPresent(V)

dans le cas où la contrainte **`NamedConstraint`** pour le champ contient le mot clé **`PRESENT`**;

not FPresent(V)

dans le cas où la contrainte **`NamedConstraint`** pour le champ contient le mot clé **`ABSENT`**. Dans tous les autres cas, la contrainte de présence a la valeur "true".

4.5 Expressions de valeur

4.5.1 Valeur de choix (ChoiceValue)

Grammaire ASN.1

`ChoiceValue` est défini dans le § 28.8 de [3].

```
ChoiceValue   ::=  identifier ":" Value
```

Modèle

Une valeur **`ChoiceValue`** est représentée par une application <operator application> ayant la valeur **`Value`** comme argument. L'identificateur <operator identifier> de l'application <operator application> contient un qualificateur <qualifier> représentant le type **`Type`** et un nom d'opérateur comme identificateur **`identifier`**.

Exemple

La valeur **`ChoiceValue`**:

```
myvalue : Mychoice
```

est représentée par:

myvalue(Mychoice)

Au cas où une valeur **`ChoiceValue`** particulière pourrait désigner une application d'opérateur parmi d'autres (c'est-à-dire un champ comportant plusieurs sortes de choix), un qualificateur est utilisé:

```
MyType ::= CHOICE …

myvalue : Mychoice
```

et la représentation est alors la suivante:

<<type Mytype>> myvalue(Mychoice)

4.5.2 Primaire composite

Un primaire composite est constitué des valeurs correspondant à la représentation SDL des types composites successifs.

4.5.2.1 Valeur SequenceValue

Grammaire ASN.1

SequenceValue est défini dans le § 24.16 de [3].

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

NOTE – Il n'y a pas de distinction entre **SetValue** et **SequenceValue**. C'est un assouplissement par rapport à la Rec. UIT-T X.680.

Modèle

La spécification **SequenceValue** en ASN.1 est mappée sur la définition **synonym** en langage SDL. Lors du mappage, la liste **ComponentValueList** est fournie au constructeur de type de données de structure en langage SDL. Le constructeur de type de données SDL exige que tous les champs soient fournis de façon que les champs qui sont omis dans la liste **ComponentValueList** soient fournis vides en langage SDL. L'application du constructeur de type de données de structure aura les mêmes effets en langage SDL qu'en notation ASN.1.

Exemple

```
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}
```

Dans cet Exemple, les champs a, b, c et d de myValue ont une valeur d'affectation et les champs e et f n'ont pas d'affectation.

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

En conséquence, les champs a, b, c et d de myValue seraient mis à 1, e serait absent et f serait affecté de la valeur par défaut 0.

4.5.2.2 Valeur SequenceOfValue

Grammaire ASN.1

SequenceOfValue est défini dans le § 25.3 de [3].

```
SequenceOfValue    ::=  "{" ValueList "}" | "{"  "}"
ValueList          ::=  Value | ValueList "," Value
```

Modèle

Une valeur **SequenceOfValue** est représentée par:

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

où E1, E2, ..., En sont les valeurs **Values** de la valeur **SequenceOfValue** dans l'ordre d'apparition. Si aucune valeur **Values** n'est spécifiée, la valeur **SequenceOfValue** est représentée par le nom Emptystring.

Le qualificateur de type **Type** du primaire composite qui contient la valeur **SequenceOfValue** précède chaque opérateur MkString ou le littéral Emptystring respectivement.

4.5.2.3 Valeur ObjectIdentifierValue

Grammaire ASN.1

`ObjectIdentifierValue` est défini dans le § 31.3 de [3].

Modèle

`ObjectIdentifierValue` est ignoré dans la transformation en SDL.

On utilise **`ObjectIdentifierValue`** en ASN.1 pour identifier les modules qui ont le même nom mais des identificateurs différents. Etant donné que les noms de module et les identificateurs d'objet ne peuvent pas être mappés de façon univoque sur un identificateur de paquetage utilisé dans des clauses d'utilisation de paquetages, la composante d'identificateur d'objet est ignorée lors du passage en langage SDL. L'identification de modules appropriés sera donc effectuée manuellement ou au moyen d'outils spécifiques.

4.5.2.4 Valeur RealValue

Grammaire ASN.1

`RealValue` est défini dans le § 20.6 de [3].

```
RealValue          ::=
         NumericRealValue | SpecialRealValue
NumericRealValue   ::=
         0    |
         SequenceValue     -- Valeur du type de séquence associé
SpecialRealValue   ::=
         PLUS-INFINITY | MINUS-INFINITY
```

La valeur **`0`** est utilisée pour des valeurs nulles; toute autre forme pour **`NumericRealValue`** ne doit pas être utilisée pour des valeurs nulles.

Le type associé utilisé à des fins de définition de valeurs et de sous-typage est:

```
SEQUENCE {
     mantissa  INTEGER,
     base      INTEGER (2|10),
     exponent  INTEGER
          -- Le numéro réel mathématique associé est "mantissa"
          -- multiplié par "base" élevé à la puissance "exponent"
     }
```

Modèle

Une valeur **`NumericalRealValue`** ASN.1 est mappée sur une valeur de sorte **`real`** en SDL au moyen de la valeur réelle calculée lors de la transformation. La valeur **`SpecialRealValue`** doit donc être transformée en la plus grande valeur positive ou négative possible.

NOTE – La transformation de la valeur **`SpecialRealValue`** n'est pas conforme à la syntaxe ASN.1 prévue étant donné que le codeur/décodeur doit utiliser un code spécial pour designer la valeur –∞ (moins l'infini). Etant donné que le codage ne concerne pas les données en SDL transformées à partir des données en ASN.1, cet assouplissement devrait être autorisé.

Exemple

La définition ASN.1:

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

équivaut à:

synonym r50 Real = 50.0;

4.5.3 Primaire de chaîne

Grammaire ASN.1

Le chaîne caractères en ASN.1 est définie dans le § 11.11 de [3].

`BitStringValue` est défini dans le § 21.9 de [3].

```
BitStringValue ::=
         bstring                |
         hstring                |
         "{" IdentifierList "}"  |
         "{" "}"
IdentifierList ::=
         identifier    |
         IdentifierList "," identifier
```

Modèle

Une valeur **`StringValue`** ASN.1 contenant une chaîne **`cstring`** (nom ASN.1 associé à la chaîne de caractères délimitée par **"** au début et à la fin de celle-ci) représente un identificateur <character string literal identifier> constitué du type **`Type`** et d'un littéral <character string literal> comportant le même texte <text> que le texte **`Text`** de la chaîne String ASN.1. Le type **`Type`** de la chaîne **`cstring`** est un type **`IA5Type`**, tel que défini par la présente Recommandation.

Une valeur **`StringValue`** contenant une valeur **`BitStringValue`** ou **`HexStringValue`** est mappée sur les opérateurs Bitstring <<**package** Predefined>> en SDL avec la même syntaxe.

4.5.4 Spécification d'ensembles d'éléments (ElementSetSpecs)

Grammaire ASN.1

`ElementSetSpecs` est défini dans le § 46.1 de la référence [3].

```
ElementSetSpecs             ::=
         RootElementSetSpec |
         RootElementSetSpec  ","   "..."  |
         "..."  ","  AdditionalElementSetSpec |
         RootElementSetSpec  ","  "..."  ","  AdditionalElementSetSpec

RootElementSetSpec          ::=  ElementSetSpec
AdditionalElementSetSpec    ::=  ElementSetSpec
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
```

Modèle

Deux ensembles de valeurs ou plus peuvent être combinés au moyen de cette notation. L'ensemble qui en résulte est évalué lors de la transformation et le résultat est mappé en langage SDL.

Les instances de "..." de **`ElementSetSpecs`** sont ignorées dans la transformation en SDL.

5 Mappage de types ASN.1 définis dans des modules ASN.1 utilisant des objets, des classes et des ensembles d'information

5.1 Introduction

La Rec. UIT-T X.681 spécifie la notation ASN.1 qui permet de définir les classes d'objets d'information ainsi que les objets d'information proprement dits et leurs ensembles et de leur donner des noms de référence. Une classe d'objets d'information est un modèle utilisé pour représenter un ensemble d'informations qui définit les propriétés de tous les membres de cette classe. Les objets d'information fournissent un mécanisme de table générique en langage ASN.1. Cette table définit l'association d'ensembles spécifiques de valeurs ou de types de champs. Cette fonction, qui remplace la construction MACRO antérieure (formulée en notation ASN.1:1990), est principalement utilisée pour remplir les espaces contenus dans une définition de type qui dépend d'un ou de plusieurs champs de clés.

Dans le présent paragraphe, on suppose que toutes les constructions ASN.1 définies dans les Recs. UIT-T X.681, X.682 et X.683 peuvent être utilisées dans des modules ASN.1. On y donne ensuite les informations contenues dans les classes d'objets d'information, dans les objets d'information et dans les ensembles d'objet d'information ASN.1, qui peuvent être utiles lors de leur mappage sur les cibles SDL appropriées. On définit les mappages qui sont possibles et utiles. Il convient de noter que certaines informations ne seront pas représentées en langage SDL en raison de la nature différente des deux langages.

5.2 Définition et affectation de classe d'objets d'information

Grammaire ASN.1

`ObjectClassAssignment` est défini dans le § 9.1 de [4].

Les définitions de **`ObjectClass`** en ASN.1 n'ont pas de correspondance directe en SDL.

5.3 Type champ de classe d'objets

Grammaire ASN.1

`ObjectClassFieldType` est défini dans le § 14.1 de [4].

```
ObjectClassFieldType ::= DefinedObjectClass "." FieldName

FieldSpec ::=
     TypeFieldSpec |
     FixedTypeValueFieldSpec |
     VariableTypeValueFieldSpec |
     FixedTypeValueSetFieldSpec |
     VariableTypeValueSetFieldSpec |
     ObjectFieldSpec |
     ObjectSetFieldSpec
```

Modèle

Les types ASN.1 peuvent être définis en utilisant la notation **`ObjectClassFieldType`** pour extraire les informations des champs de spécification de classe en l'absence de contraintes de tableau. De tels types ASN.1 peuvent être mappés sur le SDL à condition que l'on utilise, dans leur définition, uniquement **`FixedTypeValueFieldSpec`** ou **`FixedTypeValueSetFieldSpec`**. Le mappage sur un type valeur SDL s'effectue comme indiqué au § 4.3 dès que le sens de **`FixedTypeValueFieldSpec`** ou de **`FixedTypeValueSetFieldSpec`** a été déterminé à partir des spécifications de la classe référencée.

La notation **`ObjectClassFieldType`** est également utilisée en relation avec les contraintes tabulaires définies dans le § 5.5.2.

Exemple

Si la notation ASN.1 contient la spécification suivante:

```
EXAMPLE-CLASS ::= CLASS {
     &TypeField                                  OPTIONAL,      -- champ classe 1
     &fixedTypeValueField          INTEGER       OPTIONAL,      -- champ classe 2
     &variableTypeValueField       &TypeField    OPTIONAL,      -- champ classe 3
     &FixedTypeValueSetField       INTEGER       OPTIONAL,      -- champ classe 4
     &VariableTypeValueSetField    &TypeField    OPTIONAL       -- champ classe 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,    -- champ 1
     integerComponent2  EXAMPLE-CLASS.&FixedTypeValueSetField  -- champ 2
}

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

Peuvent être mappés en langage SDL le type **`ExampleType`** et la valeur **`exampleValue`**:

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

5.4 Définition et affectation d'objet d'information

Grammaire ASN.1

`ObjectAssignment` est défini au § 11.1 de [4].

Modèle

Les définitions de **`Object`** dans le module ASN.1 n'ont pas de mappage équivalent en SDL.

5.5 Information à partir d'objets

Grammaire ASN.1

`InformationFromObjects` est défini au § 15.1 de [4].

Modèle

L'information de la colonne du tableau associé pour un objet ou un ensemble d'objets peut être mentionnée par les divers cas de la notation **`InformationFromObjects`**.

Dans le module ASN.1, on peut spécifier un type ASN.1 avec des champs définis au moyen de la notation **`InformationFromObjects`**. Un tel type ASN.1 peut être mappé sur le SDL à condition que toutes les occurrences de la notation **`InformationFromObjects`** puissent être étendues à une valeur ou à un type. Le type ASN.1 en tant que tel est mappé comme indiqué dans le § 4.3 alors que la sémantique de l'expansion **`InformationFromObjects`** suit la sémantique ASN.1.

5.6 Spécification des contraintes

Grammaire ASN.1

`GeneralConstraint` est défini dans le § 8.1 de [5].

```
GeneralConstraint ::=
     UserDefinedConstraint |
     TableConstraint
```

Modèle

Les types spécifiés utilisant **`TableConstraint`** sont mappés sur le SDL conformément aux règles du § 5.5.2. Les types spécifiés au moyen de **`UserDefinedConstraint`** ne peuvent pas être mappés sur le SDL.

5.6.1 Contraintes définies par l'utilisateur

Grammaire ASN.1

`UserDefinedConstraint` est défini dans le § 9.1 de [5].

```
UserDefinedConstraint ::=
     CONSTRAINED BY "{" UserDefinedConstraintParameter ","  *  "}"
```

Modèle

Cette forme de spécification de contrainte peut être considérée comme une forme spéciale de commentaire ASN.1, étant donné qu'elle n'est pas entièrement exploitable par machine. Pour cette raison, les spécifications de types utilisant **`UserDefinedConstraint`** ne peuvent pas être mappées sur le SDL.

5.6.2 Contraintes tabulaires

Grammaire ASN.1

`TableConstraint` est défini dans le § 10.3 de [5].

```
TableConstraint ::=
     SimpleTableConstraint |
     ComponentRelationConstraint
SimpleTableConstraint ::= ObjectSet
ComponentRelationConstraint ::=
     "{" DefinedObjectSet "}"   "{" AtNotation ","  +  "}"
AtNotation ::=
     "@" ComponentIdList |
     "@." ComponentIdList
ComponentIdList             ::= identifier  "."  +
```

Modèle

Une notation de contrainte peut apparaître (entre parenthèses) après toute utilisation de la structure syntaxique "Type". Les concepteurs d'applications peuvent utiliser cette notation pour définir un type données structurées avec d'autres contraintes dans leurs valeurs de champs. Des exemples de telles contraintes sont la limitation de la portée d'un ou de plusieurs composants ou la spécification d'une relation entre composants. Le premier est une **`SimpleTableConstraint`**, la seconde une **`ComponentRelationConstraint`**.

Les règles de transformation suivantes s'appliquent aux types à contrainte **`SimpleTableConstraint`**.

Avant que le type contraint puisse être mappé sur le SDL, il faut construire quelques types valeur SDL de la manière suivante à partir de la spécification de classe et de la spécification d'ensemble de contraintes:

a) pour chaque ensemble d'objets sont créés un certain nombre de types valeur SDL. Ces types sont créés de manière que, pour chaque champ de la CLASS associée à l'ensemble d'objets un type valeur SDL soit généré. Le nom du type est la concaténation du nom de l'ensemble d'objets, un soulignement ('_') et le nom du champ de la classe assortie;

b) si le champ de la classe est une **`FixedTypeValueFieldSpec`**, un syntype SDL est construit. Celui-ci a une contrainte de portée qui est une association de valeurs spécifiées par le champ correspondant de chaque objet de l'ensemble d'objets;

c) si le champ de la classe est une **`VariableTypeValueFieldSpec`**, un type choix SDL est construit de telle manière que tous les types se trouvant dans le champ correspondant de tous les objets appartenant à l'ensemble d'objets contraignants sont inclus dans le choix. Les noms des champs de choix sont les équivalents en minuscules des types correspondants.

Le type ASN.1 contraint peut maintenant être mappé sur le SDL. Le type en tant que tel est mappé comme indiqué au § 4.3. Les noms de champs SDL sont les mêmes que les noms de champs ASN.1. La spécification ASN.1 du caractère optionnel est préservée dans la transformation. Pour chaque champ ASN.1 contraint par un ensemble d'objets, le type SDL est spécifié comme un type construit à partir de la spécification de classe et de la spécification d'ensemble contraignant (points a) à c) qui précèdent).

Pour les spécifications de types ASN.1 utilisant **`ComponentRelationConstraint`**, les mêmes règles de transformation des types s'appliquent. En plus de cela, pour chaque type ASN.1 ayant une contrainte **`ComponentRelationConstraint`** est également générée une méthode de contrôle qui traverse l'objet et vérifie les contraintes. Elle accepte si toutes les contraintes relationnelles sont respectées et refuse si l'une quelconque de ces contraintes n'est pas suivie.

Les étapes pour la construction de la méthode de contrôle sont:

pour chaque élément qui intervient dans la contrainte relationnelle (autrement dit qui est accompagné d'une contrainte **`ComponentRelationConstraint`** ou qui est mentionné dans une contrainte **`ComponentRelationConstraint`** quelconque), une déclaration de variable d'essai locale est générée, à savoir:

'dcl <test var name> <field type>; <test var name> := <field ref>;'

où <test var name> est un nom unique pour chaque variable d'essai, <field type> est le type de l'élément, <field ref> est une référence à l'élément. Si l'élément est présent, la variable est initialisée à la valeur du champ correspondant de l'objet.

Pour chaque contrainte relationnelle est généré un test s'appliquant à chaque combinaison de valeurs ou de types contraignants dans la définition de l'ensemble d'objets. Chaque test est généré au moyen de:

'si (<test expr> et non (<value test>) alors { return False; }

où le terme <test expr> résulte de la combinaison de tests individuels pour chaque valeur ou type contraignant utilisant l'opérateur 'and'. Pour les valeurs contraignantes, le test est défini comme étant

'<test var name> = <test value>'

où <test var name> est le nom de la variable de test telle que décrite ci-dessus et <test value> est la valeur correspondante de la définition de l'ensemble d'objets.

Pour les types contraignants, le test est défini comme étant:

'<test var name>.<ispresent method>'

où <test var name> est le nom de la variable de test telle que décrite ci-dessus et <ispresent method> est la méthode qui vérifie la présence du type correspondant.

Le terme <value test> résulte de la combinaison de tests individuels pour chaque valeur ou type de l'élément contraignant dans la définition de l'ensemble d'objets qui correspond aux valeurs et types dans l'expression <test expr> ci-dessus, avec utilisation de l'opérateur ‘or’. Pour les valeurs, chaque test est défini comme étant:

'<test var name> = <value>'

où <test var name> est le nom de la variable correspondant au champ contraint et <value> est une valeur de la définition d'ensemble d'objets.

Pour les types, le test est défini comme étant:

'<test var name>.<ispresent method>'

où <test var name> est le nom de la variable correspondant au champ contraint et où <ispresent method> est la méthode qui vérifie la présence du type correspondant.

Pour chaque champ String du type est générée une boucle conformément à:

'loop(dcl <loop var> Integer := 1; <loop var> <= length(<string field>);
<loop var> := <loop var> + 1) { <loop body> }'

où <loop var> est un nom variable unique, <string field> une référence au champ de la chaîne traitée et <loop body> le résultat de l'application des étapes de transformation du présent paragraphe aux éléments de la chaîne.

Exemple 1

Exemple d'un type avec **`SimpleTableConstraint`**:

```
ErrorReturn ::= SEQUENCE
{
  errorCategory ERROR-CLASS.&category({ErrorSet}) OPTIONAL,
  errors SEQUENCE OF SEQUENCE
  {
    errorCode ERROR-CLASS.&code
                          ({ErrorSet}),
    errorInfo ERROR-CLASS.&Type
                          ({ErrorSet})
  } OPTIONAL
}
```

Si les spécifications de classe et d'ensemble d'objets étaient:

```
ERROR-CLASS ::= CLASS
{
  &category PrintableString (SIZE(1)),
  &code     INTEGER,
  &Type
}
WITH SYNTAX {&category &code &Type }
```

```
ErrorSet ERROR-CLASS ::=
{
  { "A" 1 INTEGER } |
  { "A" 2 REAL } |
  { "B" 1 CHARACTER STRING } |
  { "B" 2 GeneralString }
}
```

les types SDL obtenus à partir de la spécification de contrainte seraient:

```
syntype ErrorSet_category = PrintableString (SIZE(1))
  constants 'A', 'B'
endsyntype;

syntype ErrorSet_code = <<package Predefined>> Integer
  constants 1, 2
endsyntype;

value type ErrorSet_Type { choice
  integer          <<package Predefined>> Integer;
  real             <<package Predefined>> Real;
  characterString  <<package Predefined>> CharacterString;
  generalString    <<package Predefined>> GeneralString;
}
```

le type SDL construit serait:

```
value type ErrorReturn { struct
  errorCategory ErrorSet_category optional;
  errors String <
    { struct
      errorCode ErrorSet_code,
      errorInfo ErrorSet_Type } > optional;
}
```

Aucune méthode de contrôle ne serait générée.

Exemple 2

Exemple d'un type avec **ComponentRelationConstraint**.

```
ErrorReturn ::= SEQUENCE
{
  errorCategory ERROR-CLASS.&category({ErrorSet}) OPTIONAL,
  errors SEQUENCE OF SEQUENCE
  {
    errorCode ERROR-CLASS.&code
                        ({ErrorSet}{@errorCategory}),
    errorInfo ERROR-CLASS.&Type
                        ({ErrorSet}{@errorCategory,@.errorCode})
  } OPTIONAL
}
```

le type SDL correspondant serait:

```
value type ErrorReturn {
struct
  errorCategory ErrorSet_category optional;
  errors String <
    { struct
      errorCode ErrorSet_code,
      errorInfo ErrorSet_Type } > optional;
  method Check() -> Boolean
  {
     dcl t1 ErrorSet_category;
```

```
     dcl p1 Boolean;
     p1 := this.errorCategoryPresent();
     if (p1 = True)
{
          t1 := this.errorCategory;
     }
     if ((p1 = False) and (this.errorsPresent() =  True))
     {
          return False;
     }
     loop (dcl i1 Integer := 1; I <=length(errors); i1 := i1+1)
     {
          dcl t2 ErrorSet_code, t3 ErrorSet_Type;
          t2 := this.errors[i1].errorCode;
          t3 := this.errors[i1].errorInfo ;
          if (t1="A" and not( t2=1 or t2=2))
          {
               return False;
          }
          if (t1="B" and not( t2=1 or t2=2))
          {
               return False;
          }
          if (t1="A" and t2=1 and not (t3.integerPresent()))
          {
               return False;
          }
          if (t1="A" and t2=2 and not (t3.realPresent()))
          {
               return False;
          }
          if (t1="B" and t2=1 and not (t3.characterStringPresent()))
{
               return False;
          }
          if (t1="B" and t2=2 and not (t3.generalStringPresent))
{
               return False;
          }
           }
    }
}
```

6 Mappage de spécifications ASN.1 paramétrées

La Rec. UIT-T X.683 [6] définit la manière de paramétrer les spécifications UIT-T ASN.1. Tous les concepts ASN.1:1997 peuvent être paramétrés. Cette caractéristique permet de spécifier partiellement des types ou des valeurs dans un module ASN.1, la spécification étant complétée par l'adjonction des paramètres réels au moment de l'instantiation.

La Rec. UIT-T Z.100 définit un concept équivalent de paramètres de contexte formel.

6.1 Affectation paramétrée

Grammaire ASN.1

Des énoncés d'affectation paramétrée correspondent à chacun des énoncés d'affectation spécifiés dans les Recs. UIT-T X.680 et X.681. La construction "ParameterizedAssignment" est:

```
ParameterizedAssignment ::=
     ParameterizedTypeAssignment          |
     ParameterizedValueAssignment         |
```

```
    ParameterizedValueSetTypeAssignment   |
    ParameterizedObjectClassAssignment        |
    ParameterizedObjectAssignment             |
    ParameterizedObjectSetAssignment
```

Modèle

L'emploi de toutes les formes de **ParameterizedAssignment** est pris en charge dans les modules ASN.1.

ParameterizedTypeAssignment peut être mappé sur le SDL comme indiqué au § 6.2 fondée sur les mécanismes des paramètres de contexte formel SDL.

ParameterizedValueSetTypeAssignment, ParameterizedObjectClassAssignment, ParameterizedObjectAssignment et ParameterizedObjectSetAssignment peuvent être utilisés dans les modules ASN.1 afin d'être utilisés dans d'autres spécifications ASN.1 mais ne sont pas eux-mêmes mappés sur le SDL.

6.2 Affectation de types paramétrés

Grammaire ASN.1

```
ParameterizedTypeAssignment ::=
    typereference
    ParameterList
    "::="
    Type
ParameterList ::= "{" Parameter "," + "}"
Parameter ::= ParamGovernor ":" DummyReference | DummyReference
ParamGovernor ::= Governor | DummyGovernor
Governor ::= Type | DefinedObjectClass
DummyGovernor ::= DummyReference
DummyReference ::= Reference
```

Modèle

La différence entre des types ASN.1 ordinaires et des types ASN.1 paramétrés est que la liste **ParameterList** suit **typereference** et que les paramètres formels contenus dans **ParameterList** sont utilisés dans la définition de **Type**.

Un **Type** défini en ASN.1 au moyen de paramètres de la liste **ParameterList** est mappé sur le type SDL approprié (tel que défini dans le § 4.2.1) à condition que les paramètres ASN.1 soient des paramètres de valeur ou de type. De tels paramètres sont mappés sur des paramètres <formal context parameters> du type SDL. Le paramètre du type ASN.1 est mappé sur le paramètre SDL <sort context parameter> et le paramètre de valeur ASN.1 est mappé sur le paramètre SDL <synonym context parameter>.

Les types ASN.1 paramétrés ayant des paramètres qui ne sont pas types ou des valeurs ne peuvent pas être mappés directement sur le SDL. Toutefois, si les paramètres peuvent d'abord être élargis en types ou valeurs, le type ou la valeur ASN.1 résultant peut être mappé sur le SDL comme indiqué au § 6.3.

Exemple

La définition du type ASN.1:

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

```
}
```

est mappée sur le type SDL type:

```
value type TemplateMessage
<synonym minSize <<package Predefined>> Integer; synonym maxSize <<package
                            Predefined>> Integer; value type IndicatorType>
{
struct
     asp           Integer;
     pdu           <<package Predefined>>Octetstring (SIZE(minSize:maxSize));
     indicator     IndicatorType;
}
```

6.3 Désignation de définitions de type ASN.1 paramétrées

Grammaire ASN.1

```
ParameterizedType ::=
          SimpleDefinedType
          ActualParameterList
ActualParameterList ::=
          "{" ActualParameter "," + "}"
ActualParameter ::=
     Type |
     Value    |
     ValueSet  |
     DefinedObjectClass |
     Object    |
     ObjectSet
```

Modèle

Des références à **ParameterizedType** et **ParameterizedValue** sont utilisées en ASN.1 pour définir de nouveaux types et valeurs ASN.1 au moyen d'une liste **ActualParameterList**.

Si la définition de **ParameterizedType** était telle qu'il était possible de la mapper sur le SDL et si la liste **ActualParameterList** contient uniquement des paramètres **Type** et **Value**, les références ASN.1 à de telles définitions peuvent être mappées sur les instantiations SDL du type avec des paramètres de contexte de telle manière que les éléments de la liste **ActualParameterList** sont mappés sur les paramètres <actual context parameters>. L'exemple 1 illustre un tel mappage.

Si, conformément au § 6.2, la définition de **ParameterizedType** n'a pas pu être mappée sur la définition du type SDL avec des paramètres de contexte, les références à de telles définitions de **ParameterizedType** peuvent être mappées sur le SDL de telle manière que le sens d'un tel type est complètement élargi au niveau des types définis dans le paragraphe 4 avant que le mappage sur le langage SDL ne soit effectué.

Si la liste **ActualParameterList** contient **ValueSet**, **DefinedObjectClass**, **Object** ou **ObjectSet**, le mappage d'un tel type sur le SDL s'effectue de telle manière que le sens d'un tel type est entièrement élargi au niveau des types définis dans le paragraphe 4 avant que le mappage sur le langage SDL ne soit effectué. L'exemple 2 illustre un tel mappage.

Les types et valeurs ASN.1 obtenus à partir des définitions paramétrées ASN.1 référencées peuvent être mappés sur le SDL comme indiqué au paragraphe 4.

Exemple 1

Le type paramétré utilisé dans l'exemple du § 6.2 peut être utilisé pour définir un type ASN.1 simple de la manière suivante:

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

Cela peut être mappé sur le type SDL:

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

Exemple 2

Ce qui suit est un exemple de définition de type ASN.1 obtenu au moyen d'un paramètre qui est un objet d'information. Le module ASN.1 doit contenir la définition de classe d'objets d'information appropriée avec une affectation paramétrée ayant l'objet de cette classe comme paramètre fictif, une affectation de valeur d'objet d'information et une référence de définition de type paramétré.

```
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 }
```

Etant donné la sémantique de l'ASN.1, la présence d'une telle définition de classe, d'une définition de type paramétré et d'une définition de valeur d'objet, le type résultant de la transformation du message MY-Message-PDU est équivalent à:

```
MY-Message-PDU ::=      SEQUENCE {
     priority-level     INTEGER  (0..10),
     message            BMPString (SIZE    (0..2000))
}
```

Le type ASN.1 résultant peut être mappé sur le type SDL comme suit:

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

6.4 Désignation de valeurs ASN.1 paramétrées

Grammaire ASN.1

```
ParameterizedValue ::=
     SimpleDefinedValue
     ActualParameterList

SimpleDefinedValue ::=
     Externalvaluereference |
     valuereference

ActualParameterList ::=
          "{" ActualParameter "," + "}"
```

```
ActualParameter ::=
     Type     |
     Value    |
     ValueSet  |
     DefinedObjectClass |
     Object   |
     ObjectSet
```

Modèle

Les références à **ParameterizedValue** sont utilisées en ASN.1 pour définir de nouvelles valeurs ASN.1 au moyen d'une liste **ActualParameterList**.

Les références à **ParameterizedValue** sont mappées sur le SDL de telle manière que le sens d'une telle spécification de valeur est entièrement élargi au niveau des affectations de valeur définies dans le paragraphe 4 avant que le mappage sur le SDL n'ait lieu.

6.5 Désignation d'autres définitions paramétrées ASN.1

```
ParameterizedValueSetType ::=
     SimpleDefinedType
     ActualParameterList

ParameterizedObjectClass ::=
     DefinedObjectClass
     ActualParameterList

ParameterizedObjectSet ::=
     DefinedObjectSet
     ActualParameterList

ParameterizedObject ::=
     DefinedObject
     ActualParameterList

ActualParameterList ::=
          "{" ActualParameter "," + "}"

ActualParameter ::=
     Type |
     Value    |
     ValueSet  |
     DefinedObjectClass |
     Object   |
     ObjectSet
```

Les modules ASN.1 peuvent contenir la spécification d'ensembles de valeurs, de classes d'objets, d'ensembles d'objets et d'objets définis par référence au type **SimpleDefinedType** avec la liste **ActualParameterList**. De telles spécifications ne sont pas mappées sur le SDL.

7 Adjonctions au paquetage Predefined

Les définitions suivantes doivent être ajoutées au paquetage "Predefined" afin de prendre en charge la combinaison de modules ASN.1 en langage SDL.

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

```
/*        Sorte NumericString      */
/*        Définition               */
value type NumericString
     inherits String < NumericChar > ( '' = emptystring )
     adding
          operators ocs in nameclass
               "" ( ('0':'9') or """" or (' ') )+ "" -> this NumericString;
/*   chaînes d'une longueur indéterminée contenant n'importe quels caractères allant de l'espace ' ' à
'9'   */
axioms
     for all c in NumericChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                                   ==> cs  == mkstring(c);
/*   la chaîne 'A' est formée du caractère '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;
/* */

/*   Sorte PrintableString      */
/*   Définition                 */
value type PrintableString
     inherits String < PrintableChar > ( '' = emptystring )
     adding
          operators ocs in nameclass
                         "" ( (' ':'&') or """" or ('(': '?') )+ "" -> this PrintableString;
/*   chaînes d'une longueur indéterminée contenant n'importe quels caractères allant de l'espace ' ' à
'?' */
axioms
     for all c in PrintableChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                                   ==> cs  == mkstring(c);
/*   la chaîne 'A' est formée du caractère 'A' etc.   */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                                    ==> cs  == cs1 // cs2;
     ));
endvalue type PrintableString;
/* */
syntype TeletexChar = Character constants
```

```
/*   caractères spécifiés au § 36.1, Tableau 3 de la Rec. UIT-T X.680 */ endsyntype;
/* */

/*   Sorte TeletexString  */
/*   Définition           */
value type TeletexString
     inherits String < TeletexChar > ( " = emptystring )
     adding
          operators ocs in nameclass
/*   caractères spécifiés au § 36.1, Tableau 3 de la Rec. UIT-T X.680 */ -> this TeletexString;
axioms
     for all c in TeletexChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                                   ==> cs  == mkstring(c);
/*   la chaîne de caractères 'A' est formée du caractère 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                                    ==> cs  == cs1 // cs2;
     ));
endvalue type TeletexString;
syntype VideotexChar = Character constants
/*   caractères spécifiés au § 36.1, Tableau 3 de la Rec. UIT-T X.680 */ endsyntype;
/* */

/*   Sorte VideotexString     */
/*   Définition               */
value type VideotexString
     inherits String < VideotexChar > ( " = emptystring )
     adding
          operators ocs in nameclass
/*   caractères spécifiés au § 36.1, Tableau 3 de la Rec. UIT-T X.680 */ -> this VideotexString;
axioms
     for all c in VideotexChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                                   ==> cs  == mkstring(c);
/*   la chaîne 'A' est formée du caractère '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 /*   Toutes les combinaisons G et C  + SPACE + DELETE sont spécifiées au § 36.1,
Tableau 3 de la Rec. UIT-T X.680 */
operators
     gchr          ( Integer ) -> this GeneralChar;
endvalue type;

value type UniversalChar
     literals /*   voir § 36.6 de la Rec. UIT-T X.680 */
```

```
operators
    uchr          ( Integer ) -> this UniversalChar;
endvalue type;
/* */

/*  Sorte UniversalCharString   */
/*  Définition                  */

value type UniversalCharString
    inherits String < UniversalChar > ( " = emptystring )
    adding
        operators ocs in nameclass
/*  voir § 36.6 de la Rec. UIT-T X.680 */ -> this UniversalCharString;
axioms
    for all c in UniversalChar nameclass (
    for all cs, cs1, cs2 in ocs nameclass (
    spelling(cs) == spelling(c)                          ==> cs  == mkstring(c);
/*  la chaîne 'A' est formée du caractère 'A' etc.   */
    spelling(cs) == spelling(cs1) // spelling(cs2),
    length(spelling(cs2)) == 1                           ==> cs  == cs1 // cs2;
    ));
endvalue type UniversalCharString;
/* */

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

/*  Sorte GeneralCharString */
/*  Définition              */
value type GeneralCharString
    inherits String < GeneralChar > ( " = emptystring )
    adding
        operators ocs in nameclass
/*  Toutes les combinaisons G et C + SPACE + DELETE sont spécifiées au § 36.1, Tableau 3 de la
Rec. UIT-T X.680 */
-> this GeneralCharString;
/*  chaînes d'une longueur indéterminée contenant n'importe quels caractères allant de l'espace ' ' à
'?'  */
axioms
    for all c in GeneralChar nameclass (
    for all cs, cs1, cs2 in ocs nameclass (
    spelling(cs) == spelling(c)                          ==> cs  == mkstring(c);
/*  la chaîne 'A' est formée du caractère 'A' etc.   */
    spelling(cs) == spelling(cs1) // spelling(cs2),
    length(spelling(cs2)) == 1                           ==> cs  == cs1 // cs2;
    ));
endvalue type GeneralCharString;
/* */
syntype GraphicChar = GeneralChar constants
/*  Toutes les combinaisons G+SPACE+DELETE spécifiées au § 36.1, Tableau 3 de la Rec. UIT-T
X.680 */
endsyntype;
```

```
/* */

/*   Sorte GraphicCharString */
/*   Définition                */
value type GraphicCharString
     inherits String < GraphicChar > ( '' = emptystring )
     adding
          operators ocs in nameclass
/*   Toutes les combinaisons G+SPACE+DELETE spécifiées au § 36.1, Tableau 3 de la Rec. UIT-T
X.680 */
-> 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
/* caractères spécifiés au § 36.1, Tableau 3 de la Rec. UIT-T X.680 */
endsyntype;
/* */

/*   Sorte VisibleString  */
/*   Définition           */
value type VisibleString
     inherits String < VisibleChar > ( '' = emptystring )
     adding
          operators ocs in nameclass
/*   caractères spécifiés au § 36.1, Tableau 3 de la Rec. UIT-T X.680 */
-> this VisibleString;
axioms
     for all c in VisibleChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                              ==> cs  == mkstring(c);
/*   la chaîne 'A' est formée du caractère 'A' etc.   */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                               ==> cs  == cs1 // cs2;
     ));
endvalue type VisibleString;

syntype BMPChar = UniversalChar CONSTANTS /* voir § 36.12 de la Rec. UIT-T X.680 */
endsyntype;
/* */
/*   Sorte BMPCharString    */
/*   Définition             */
value type BMPCharString
     inherits String < BMPChar > ( '' = emptystring )
     adding
          operators ocs in nameclass
/*   voir § 36.12 de la Rec. UIT-T X.680 */
```

```
-> this BMPCharString;
axioms
      for all c in BMPChar nameclass (
      for all cs, cs1, cs2 in ocs nameclass (
      spelling(cs) == spelling(c)                                  ==> cs  == mkstring(c);
/* la chaîne 'A' est formée du caractère '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;
```

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Réseaux câblés et transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	RGT et maintenance des réseaux: systèmes de transmission, circuits téléphoniques, télégraphie, télécopie et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	Réseaux de données et communication entre systèmes ouverts
Série Y	Infrastructure mondiale de l'information et protocole Internet
Série Z	**Langages et aspects généraux logiciels des systèmes de télécommunication**

Genève, 2002