Remplacée par une version plus récente

UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T Z.105

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

(03/95)

LANGAGES DE PROGRAMMATION

LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION COMBINÉ AVEC LA NOTATION DE SYNTAXE ABSTRAITE NUMÉRO UN

Recommandation UIT-T Z.105
Remplacée par une version plus récente

(Antérieurement «Recommandation du CCITT»)

Remplacée par une version plus récente

AVANT-PROPOS

L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'Union internationale des télécommunications (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.

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes d'études 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 n° 1 de la CMNT (Helsinki, 1er-12 mars 1993).

La Recommandation UIT-T Z.105, que l'on doit à la Commission d'études 10 (1993-1996) de l'UIT-T, a été approuvée le 6 mars 1995 selon la procédure définie dans la Résolution n° 1 de la CMNT.

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 de télécommunications.

Remplacée par une version plus récente

TABLE DES MATIÈRES

RÉSUMÉ

Objectif

La présente Recommandation définit la façon dont la notation ASN.1 peut être utilisée en combinaison avec le langage SDL. L'objectif visé est que la structure et le comportement des systèmes soient décrits en langage SDL, la notation ASN.1 étant utilisée pour décrire les paramètres des messages échangés et les données à usage interne. La présente Recommandation fait suite à la Recommandation Z.100. Pour les utilisateurs connaissant bien le langage SDL et la notation ASN.1, les extensions sont d'application aisée, en ce sens que la notation ASN.1 pourra être utilisée chaque fois que des données peuvent être exprimées en langage SDL.

Couverture

La présente Recommandation présente une syntaxe et une définition des sémantèmes permettant de combiner le langage SDL et la notation ASN.1. Les appendices présentent plusieurs vues d'ensemble du langage combiné ainsi que des exemples et des directives pour l'emploi de la présente Recommandation.

Application

Le principal champ 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.

Etat/Stabilité

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

Travaux associés

Recommandation UIT-T Z.100: Langage de description et de spécification

Recommandation UIT-T X.680: Notation de syntaxe abstraite numéro un (ASN.1).

Remplacée par une version plus récente

Recommandation Z.105

LANGAGE DE DESCRIPTION ET DE SPÉCIFICATION COMBINÉ AVEC LA NOTATION DE SYNTAXE ABSTRAITE NUMÉRO UN

(Genève, 1995)

1 Introduction

La présente Recommandation définit la combinaison de la notation ASN.1 et du langage SDL. Cette combinaison permet d'utiliser la notation ASN.1 dans des diagrammes SDL et d'importer des modules ASN.1 dans des descriptions SDL.

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. Elle fait l'objet de règles de codage qui définissent la façon dont des 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. Cette combinaison fait appel à des données, à des messages et au codage des messages utilisés par ces systèmes: structure et comportement en langage SDL, données et messages en notation ASN.1, codage par référence aux règles de codage définies expressément pour la notation ASN.1.

La présente Recommandation prend en compte la pleine utilisation des types de données en langage SDL.

1.2 Différences entre données de types SDL et ASN.1

Aussi bien le langage SDL que la notation ASN.1 offrent des éléments permettant de définir les données mais sur la base de concepts différents.

En langage SDL, un type de données se caractérise par des ensembles de valeurs (appelés sortes) et par les opérateurs qui sont définis pour ces sortes. Des éléments sont disponibles pour définir des sortes, des valeurs et des opérateurs agissant sur les valeurs. Le langage SDL possède des sortes prédéfinies et permet de définir de nouvelles sortes, de nouvelles valeurs et de nouveaux opérateurs, soit sur la base de définitions existantes ou au moyen de définitions possédant des propriétés totalement nouvelles. Les propriétés des données sont fondées sur *l'équivalence algébrique des termes*.

La notation ASN.1 est fondée sur des *ensembles*: un type de données ASN.1 définit un ensemble de valeurs. Une notation permet d'écrire des valeurs particulières d'un type de données. Aucun opérateur n'agit sur des types de données (par exemple, il n'existe pas en ASN.1 d'opérateur «+» permettant d'ajouter deux valeurs d'entier). On peut définir de nouveaux types de données, sur la base de types existants. En choisissant des règles de codage appropriées (par exemple les règles de codage définies dans la Recommandation X.690), on peut spécifier la façon dont les valeurs seront codées en flux binaires afin de les transférer à un autre dispositif.

1.3 Caractéristiques de la combinaison SDL/ASN.1

La présente Recommandation est une pure extension de la Recommandation Z.100, à l'exception de quelques restrictions mineures imposées aux règles lexicales (voir l'Appendice II.2). Fondamentalement, la présente Recommandation définit une extension de l'article 5/Z.100: *Données dans le SDL*.

Les systèmes décrits en notation combinée SDL/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 ou par des sortes SDL;
- les données peuvent être définies par des définitions de types ASN.1 ou par des définitions de sortes SDL;
- le codage des valeurs de données ASN.1 peut être défini par référence aux règles de codage applicables. Le codage est hors du champ d'application de la présente Recommandation.

Le Tableau 1 résume les caractéristiques des données SDL, ASN.1 et combinées:

TABLEAU 1/Z.105

	SDL	ASN.1	SDL/ASN.1
Définition des types	X	X	X
Notation pour les valeurs	X	X	X
Définition des opérateurs	X		X
Expressions	X		X
Codage des valeurs		X	X

1.4 Restrictions imposées au langage SDL et à la notation ASN.1

L'utilisation de la notation ASN.1 telle que définie dans la Recommandation X.680 est prise en compte en combinaison avec le langage SDL, au prix de quelques restrictions. L'Appendice I.1 donne un résumé du sous-ensemble ASN.1 supporté. Les caractéristiques définies dans les Recommandations X.681, X.682, X.683 ne sont pas supportées.

Les caractéristiques de la Recommandation X.208 qui ont été remplacées par celles de la Recommandation X.680 ne sont pas prises en compte, sauf pour le type ANY.

L'utilisation du langage SDL tel que défini dans la Recommandation Z.100 est prise en compte, avec quelques restrictions concernant les règles lexicales.

L'Appendice I.2 résume les restrictions imposées au langage SDL lorsqu'il est combiné à la notation ASN.1.

1.5 Structure de la présente Recommandation

La présente Recommandation n'est pas autonome: le langage qu'elle définit est fondé sur la Recommandation Z.100, qui reste applicable sauf pour 13 règles de production syntaxique, qui sont redéfinies dans la présente Recommandation. Celle-ci est structurée comme suit:

L'article 2 définit les modifications apportées aux règles lexicales de la Recommandation Z.100.

L'article 3 définit les modifications apportées à la Recommandation Z.100 afin d'incorporer les modules ASN.1.

L'article 4 définit les modifications apportées à la Recommandation Z.100 afin d'incorporer les types de données et les valeurs ASN.1.

L'Annexe A définit le progiciel avec données prédéfinies.

L'Appendice I résume les restrictions imposées à l'utilisation du langage SDL et de la notation ASN.1.

L'Appendice II donne des exemples d'utilisation du langage SDL en combinaison avec la notation ASN.1, ainsi que quelques directives.

L'Appendice III donne une vue d'ensemble des structures syntaxiques pouvant être créées en notation ASN.1 ainsi que les opérateurs disponibles pour ces structures.

L'Appendice IV résume les modifications syntaxiques apportées aux règles de production de la Recommandation Z.100.

1.6 Conventions utilisées dans la présente Recommandation

Fondamentalement, les conventions de la Recommandation Z.100 sont applicables, c’est-à-dire que les mots clés apparaissent en caractères gras, que les noms prédéfinis commencent par une majuscule, etc. Dans certains exemples toutefois, les conventions de l'ASN.1 sont utilisées afin de faciliter la lecture par des utilisateurs de l'ASN.1, c'est-à-dire que les mots clés sont en majuscules (et non en caractères gras).

L'index n'énumère que les occurrences définissantes. Les non-terminaux qui sont cités dans le texte mais non repris dans l'index sont définis dans la Recommandation Z.100.

2 Compléments à la Recommandation Z.100

2.1 Unités lexicales

La production <lexical unit> est modifiée comme suit:

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

NOTE – Par rapport à la définition donnée dans la Recommandation Z.100, deux modifications sont apportées à la production <lexical unit>:

1) la production <character string> est remplacée par <string> pour tenir compte de l'utilisation des guillemets simples comme délimiteurs de chaîne;

2) la production <single line note> est ajoutée pour tenir compte de l'utilisation de deux tirets avant de commencer un commentaire.

2.2 Mots clés

Les mots clés suivants sont ajoutés en variante à la production <keyword>:

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

NOTES

1 Dans la présente Recommandation, il n'y a pas de distinction entre lettres en minuscules et lettres en majuscules pour écrire les mots clés et les noms. Conformément aux conventions de la Recommandation Z.100, les mots clés sont donc écrits en caractères gras minuscules.

2 Tous les mots clés figurant dans la Recommandation X.680 ne sont pas forcément des mots clés dans la présente Recommandation. Par exemple, le terme BOOLEAN (écrit sous la forme Boolean dans la présente Recommandation) est un mot clé dans la Recommandation X.680 alors qu'il s'agit d'un nom prédéfini dans la présente Recommandation.

2.3 Noms

Les productions <national> et <special> sont remplacées par ce qui suit:

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

Dans une structure de nom <name>, une structure de point <full stop> ne doit pas être immédiatement suivie d'une structure de soulignement <underline> ou d'une autre structure de point <full stop>.

Un soulignement <underline> ne doit jamais être immédiatement suivi d'un point <full stop>.

NOTES

1 Les conditions ci-dessus impliquent que les deux formes a..b et a . b désignent trois unités lexicales <lexical unit> plutôt qu'un seul nom. Un point <full stop> joue le rôle d'une unité lexicale <lexical unit> (c'est-à-dire le rôle d'une variante de la production <special>) dans la règle syntaxique <existingsort>.

2 Les deux productions ci-dessus impliquent que les caractères de parenthèse gauche, de parenthèse droite, de crochet gauche, de crochet droit et de barre verticale ne peuvent pas faire partie de noms dans la présente Recommandation, alors que c'était le cas dans la Recommandation Z.100.

2.4 Chaînes

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

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

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

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

La production <character string> est définie dans la Recommandation Z.100.

Pour représenter un caractère de citation (') à l'intérieur d'une chaîne citée <quoted string>, on utilise une apostrophe double (").

Les chaînes binaires et hexadécimales <bitstring> et <hexstring> sont les littéraux des sortes prédéfinies Bit_string et Octet_string (voir 4.4.3 et l'Annexe A).

2.5 Opérateurs cités

La production <quoted-operator>, qui était une construction syntaxique, devient une unité lexicale:

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

où la production <infix operator> fait partie de l'unité lexicale bien qu'elle soit, dans d'autres productions, une construction syntaxique.

Il y a une ambiguïté lexicale entre les productions <quoted string> et <quoted operator> car la séquence de caractères qui forme un opérateur cité <quoted operator> vaut également pour former une chaîne citée <quoted string>. Dans de tels cas, l'unité lexicale est un opérateur cité <quoted operator>.

NOTES

1 En cas de conflit, il faut utiliser une chaîne de caractères <character string> au lieu d'une chaîne citée <quoted string>. Par exemple, une chaîne de caractères composée d'un seul astérisque devra être écrite comme suit: '*' car la construction "*" est toujours un opérateur cité <quoted operator>.

2 Le fait de faire passer la production <quoted operator> de construction syntaxique à une unité lexicale implique qu'il n'est plus permis d'insérer des séparateurs ou des commentaires à l'intérieur d'une production de type <quoted operator>.

2.6 Notes sur ligne individuelle

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

Les mêmes règles s'appliquent pour la production <single line note> et pour la production <note>, sauf qu'une note sur ligne individuelle commence par deux tirets -- plutôt que par les caractères /* et qu'elle se termine par une coupure de ligne ou par la séquence -- (selon ce qui vient en premier) plutôt que par les caractères */.

NOTE – Les deux formes suivantes:

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

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

sont identiques, tandis que la forme:

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

est un peu différente car la construction comment possède sa propre notation dans la syntaxe graphique (le symbole de commentaire).

2.7 Symboles spéciaux

La production <composite special> est complétée comme suit:

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

NOTE – Les trois dernières variantes sont des adjonctions.

2.8 Utilisation des caractères d'espacement dans des noms

Le paragraphe 2.2.2/Z.100 énumère cinq exceptions à la règle que les caractères d'espacement et de commande peuvent remplacer des caractères de soulignement dans des structures de nom <name>. Des exceptions supplémentaires sont énumérées ci-dessous, qui ne s'appliquent qu'aux nouvelles constructions syntaxiques définies dans la présente Recommandation:

- les caractères de soulignement <underline> dans le nom <name> d'une valeur d'élément nommé <namedvalue> doivent être spécifiés explicitement;
- les caractères de soulignement <underline> dans les noms <name> contenus dans un primaire composite <composite primary> doivent être spécifiés explicitement.

3 Progiciel

La production <package> est complétée comme suit:

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

où la définition de module <module definition> est la suivante:

<module definition> ::= <module> **definitions** [<tagdefault>] **::=**
begin [<modulebody>] **end**

<module> ::= <package name> [<objectidentifiervalue>]

<tagdefault> ::= **explicit tags** | **implicit tags** | **automatic tags**

<modulebody> ::= [<exports>] [<imports>] <entity in package>*

<exports> ::= **exports** [<definition selection list>] <end>

<imports> ::= **imports** <symbolsfrommodule>* <end>

<symbolsfrommodule> ::= {<definition selection list> **from** <module>}*

Dans une valeur d'identificateur d'objet <objectidentifiervalue> d'un <module>, seuls les littéraux et les opérateurs des sortes définies dans le progiciel prédéfini Predefined sont visibles.

La spécification de la production d'étiquetage par défaut <tagdefault> et la façon de la spécifier ne sont pas déterminantes.

Modèle

Si une valeur d'identificateur d'objet <objectidentifiervalue> est présente dans un <module>, celui-ci est transformé en un nouveau <module> dans lequel la valeur <objectidentifiervalue> est omise et dans lequel le nom de progiciel <package name> donne des informations sur la valeur qui était spécifiée dans la production <objectidentifiervalue>. La présente Recommandation ne spécifie pas la façon dont cette transformation s'effectue pratiquement.

Une définition de module <module definition> a la même signification qu'une définition de progiciel <package definition>, comme suit:

- le <module> (sans aucune valeur de type <objectidentifiervalue>) correspond au nom <package name>;
- l'importation <imports> correspond aux clauses de référence du progiciel <package reference clause>;
- l'exportation <exports> correspond à l'interface <interface>.

Par **exemple**, le module:

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

équivaut au module suivant:

package myway;
interface synonym yes, **synonym** no;
synonym yes Boolean = True,
no Boolean = False;
endpackage myway;

De même, lorsque le progiciel est utilisé dans l'importation <imports> d'un autre progiciel:

IMPORTS yes FROM myway;

ce qui est équivalent à la clause de référence de progiciel <package reference clause> suivante:

use myway/yes;

NOTES

1 Une définition de module <module definition> ne se termine pas par une structure de fin <end>, contrairement à une définition de progiciel <package definition>.

2 Lorsque des progiciels/modules sont utilisés dans la définition de système <system definition>, la notation de la Recommandation Z.100 doit être utilisée (c'est-à-dire au moyen de clauses <package reference clause> plutôt que d'importations <imports>).

3 A titre de directive, pour des raisons de compatibilité avec la Recommandation X.680, il y a lieu que les définitions de module <module definition> contiennent, dans la mesure du possible, des définitions en notation ASN.1.

4 Définition et utilisation de données

4.1 Définitions de variables et de données

La production <data definition> est complétée comme suit:

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

Les quatre variantes <partial type definition>, <syntype definition>, <generator definition> et <synonym definition> sont définies dans la Recommandation Z.100 tandis que les variantes <sort assignment> et <value assignment> sont définies ci-dessous.

NOTES

1 Dans la présente Recommandation, la production <data definition> permet d'utiliser la notation X.680 pour définir des sortes (c'est-à-dire des types ASN.1) et des synonymes. Les variantes <sort assignment> et <value assignment> sont nouvelles.

2 Dans la présente Recommandation, la structure <end> est obligatoire, tandis que dans la Recommandation X.680 il n'y a pas de symbole séparateur entre définitions.

4.1.1 Affectations de sortes

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

Modèle

La production <sort assignment> est une forme courte d'une définition partielle de type <partial type definition> ou <syntype definition> dont le nom défini est le nom de la sorte <sort name>.

Si les propriétés étendues <extended properties> constituent une sorte existante <existing sort> ou un sous-intervalle <subrange>, l'affectation de sorte <sort assignment> est la même qu'une définition de syntype <syntype definition> ne contenant que ces propriétés étendues <extended properties>.

Une affectation de sorte <sort assignment> est identique à une définition partielle de type <partial type definition> dans laquelle l'expression des propriétés <properties expression> est vide et dans laquelle les paramètres contextuels formels <formal context parameters> sont omis.

Exemple:

L'affectation de sorte

Integerlist ::= SEQUENCE OF INTEGER;

équivaut à

newtype Integerlist
sequence of Integer
endnewtype Integerlist;

L'affectation de sorte

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

équivaut à

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

NOTE – Il découle de ce qui précède que si des littéraux ou des opérateurs supplémentaires sont requis, il faudra utiliser une définition partielle de type <partial type definition> (c'est-à-dire la forme complète).

4.1.2 Affectations de valeurs

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

La structure d'expression close <ground expression> est définie dans la Recommandation Z.100.

Une affectation de valeur <value assignment> introduit un nom pour une valeur spécifique.

Modèle

Une affectation de valeur <value assignment> équivaut à un item de définition d'un synonyme <synonym definition item>.

Exemple:

La définition

yes BOOLEAN ::= TRUE;

équivaut à

synonym yes Boolean = True;

4.2 Expressions des sortes

Les productions <sort> et <extended properties> sont modifiées comme suit:

<sort> ::= <sort expression>

<extended properties> ::= <sort expression>

où l'expression d'une sorte est définie comme suit:

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

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

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

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

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

Si la structure **defined by** <identifier> est spécifiée, l'identificateur <identifier> doit désigner une sorte ou un syntype. Si ce n'est pas le cas, cet identificateur n'est pas déterminant.

Les variantes <existingsort> et <subrange> ne doivent pas être utilisées en tant que propriétés étendues <extended properties> d'une définition partielle de type <partial type definition> ou <syntype>.

Une expression de sorte <sort expression> définit les propriétés d'une sorte. La présence d'étiquettes de type <tag> n'est pas déterminante. Si des étiquettes sont présentes, l'expression simple qu'elles contiennent (<simple expression>) doit toujours être de la sorte Integer (entier).

Dans une définition d'opérateur <operator definition> référencée (voir la Recommandation Z.100), une sorte dont la position correspond à une sorte <sort> insérée dans une référence <textual operator reference> doit toujours équivaloir syntaxiquement à cette sorte <sort>.

Dans une signature d'opérateur <operator signature> (voir la Recommandation Z.100), une sorte dont la position correspond à une sorte <sort> insérée dans une définition d'opérateur <operator definition> doit toujours équivaloir syntaxiquement à la sorte insérée dans la définition d'opérateur <operator definition>. Si cependant la sorte est du type <existingsort>, ces deux sortes peuvent ne pas être syntaxiquement équivalentes mais elles doivent toujours correspondre à la même sorte ou au même syntype.

Il existe une ambiguïté syntaxique entre une transformation par générateur <generator transformations> et un sous-intervalle <subrange>. Une sorte qui commence par un identificateur <identifier> suivi d'une parenthèse gauche est traitée comme un identificateur de type <sort identifier> ou <syntype identifier> si cela est permis par les règles de visibilité (voir 2.2.2/Z.100) et comme un identificateur de générateur <generator identifier> si ce n'est pas possible.

Modèle

Dans la description ci-après de la façon de transformer les constructions syntaxiques précédentes en leurs représentations selon la Recommandation Z.100, l'ordre est significatif:

- le mot clé **any** est un élément syntaxique dérivé qui permet d'indiquer la sorte prédéfinie *Any_type* (voir l'Annexe A);
- une sélection <selection> est un élément syntaxique dérivé qui permet d'indiquer la sorte du champ désigné par le nom <name>. Ce champ doit toujours être associé à la sorte;
- une sorte existante <existingsort> représente un identificateur de sorte <sort identifier> ou de syntype <syntype identifier>. Si un <module> est présent, il représente un nom de progiciel <package name> (comme expliqué à l'article 3) qui constitue donc la partie gauche du qualificateur de l'identificateur;
- une sorte qui n'est ni de type <existingsort> ni de type <subrange> représente un identificateur de sorte <sort identifier> d'une sorte implicitement définie. Celle-ci possède un nom anonyme (alias) unique et ses propriétés étendues <extended properties> sont l'expression de sorte <sort expression>. Cette sorte sera définie dans la plus proche unité de portée englobante;
- une sorte qui est un sous-intervalle <subrange> représente un identificateur de syntype <syntype identifier> d'un syntype implicitement défini, contenant ce sous-intervalle <subrange>. Ce syntype possède un nom anonyme (alias) unique et est défini dans la plus proche unité de portée englobant le module où la sorte apparaît. Un modèle spécial s'applique au cas d'un sous-intervalle <subrange> qui est un identificateur de sorte parente <parent sort identifier> (voir 4.2.7);
- une définition de type partielle <partial type definition> contenant des propriétés étendues <extended properties> désignant un constructeur de sorte <sort constructor> est représentée comme spécifié dans les paragraphes ci-après.

4.2.1 Champs et proposition d'options

Afin de simplifier les notations de type et de valeur dans le *Modèle*, on définira comme suit la notion de champ *(field*).

Zéro, un ou plusieurs noms de champ est (sont) associé(s) à une sorte. Les noms de champ d'une sorte sont les noms qui, une fois concaténés avec le nom d'opérateur Extract!, forment des noms d'opérateur pour les opérateurs définis localement pour la sorte et ne possédant qu'un seul argument, qui est la sorte. Chaque champ possède également une sorte de champ (*field sort*) qui est la sorte du résultat (result sort) de l'opérateur.

Un champ est facultatif (*optional*) s'il existe également un opérateur local dont le nom est le nom du champ concaténé avec le nom qualificateur Present, dont le (seul) argument est la sorte à laquelle ce champ est associé et dont la sorte de résultat est de la sorte prédéfinie Boolean.

NOTE 1 – Dans le modèle de données sous-jacent (ACT ONE), les propriétés d'une sorte ne sont caractérisées que par ses littéraux, ses opérateurs et ses axiomes. Les structures de données composites telles que des définitions SDL de structure <structure definition> et des séquences ASN.1 <sequence> sont donc des notations dérivées pour définir des littéraux, des opérateurs et des axiomes. C'est pour cette raison que les notions de champs et de champs facultatifs sont construites d'après l'existence d'opérateurs spécifiques au lieu d'en faire des propriétés de sortes définies par une construction comme <sequence>.

NOTE 2 – Du *Modèle* de <sequence> (voir 4.2.2) il découle qu'un champ qui possède une valeur par défaut est traité comme un champ facultatif qui contient toujours une valeur bien définie et dont l'opérateur Present a toujours la valeur True.

NOTE 3 – Il s'ensuit que les sortes définies par les notations abrégées <sequence> et <choice> ont des champs associés. L'utilisation de ces notations abrégées n'est pas la seule façon d'associer des champs à des sortes.

NOTE 4 – Il s'ensuit que les champs spécifiés avec le mot clé **default** à l'intérieur d'une séquence <sequence> (voir 4.2.2 ci-dessous) sont également des champs facultatifs.

Exemple:

Comme expliqué au 4.2.2, la séquence:

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

équivaut à:

```
newtype S
  operators
  Make!      : Integer        -> S;
  AExtract!  : S              -> Integer;
  BExtract!  : S              -> Real;
  CExtract!  : S              -> IA5String;
  AModify!   : S, Integer     -> S;
  BModify!   : S, Real        -> S;
  CModify!   : S, IA5String   -> S;
  BPresent   : S              -> Boolean;
  CPresent   : S              -> Boolean;
axioms
/* Les propriétés des opérateurs ci-dessus définis, voir 4.2.2 */
endnewtype S;
```

NOTE 5 – De cette définition de type nouveau (newtype), il découle (comme prévu) que la sorte S possède trois champs: A, B et C, où B et C sont des champs facultatifs. Il s'ensuit également que la sorte du champ A est Integer, que la sorte du champ B est Real et que la sorte du champ C est IA5String.

4.2.2 Sequence

<sequence> ::= { **sequence** | **set** } **{** [<elementsort> { **,** <elementsort>}*] **}**
<elementsort> ::= <namedsort> [**optional** | **default** <ground expression>] |
components of <sort>
<namedsort> ::= [<name>] <sort>

Dans la structure de sorte d'éléments nommés <namedsort>, le nom <name> ne peut être omis que si la sorte est une sélection <selection>. Dans ce cas, le nom est dérivé comme étant équivalent au nom <name> contenu dans la sélection <selection> (cela s'applique à tous les usages de la production <namedsort> dans la présente Recommandation).

La spécification du mot clé **sequence** ou du mot clé **set** n'est pas déterminante.

Une expression close <ground expression> contenue dans une sorte d'élément <elementsort> doit toujours être de la sorte indiquée par la production <sort> contenue dans la sorte d'éléments nommés <namedsort> de la sorte d'élément <elementsort>.

Le nom <name> contenu dans les sortes d'éléments nommés <namedsort> de la séquence <sequence> doit toujours être de type distinctif.

Modèle

Bien que cela ne soit pas indiqué dans les axiomes dérivés dans ce *Modèle*, tous les identificateurs d'opérateur contenus dans les axiomes sont entièrement qualifiés de façon à éviter toute confusion avec des opérateurs d'autres sortes.

Une sorte d'élément <elementsort> qui fait partie des composants d'une sorte (**components of** <sort>) est un élément syntaxique dérivé qui correspond à une liste ordonnée de sortes d'élément <elementsort>, chacune étant affectée à un des champs associés à la sorte principale <sort> (voir 4.2.1). Ces sortes d'élément <elementsort>, qui sont des champs facultatifs dans la sorte <sort>, sont spécifiées avec le mot clé **optional**. Les noms des champs sont mis en relation d'ordre au moyen des mêmes règles que pour les noms de littéraux de type classe de nom (**nameclass**). Voir 5.3.1.14/Z.100.

Une séquence <sequence> est un élément syntaxique dérivé qui correspond à une expression de propriétés <properties expression> contenant les opérateurs suivants:

- Des opérateurs de type "Extract!", qui associent les noms de champ <name> à la séquence <sequence> (voir 4.2.1).
- Des opérateurs de type "Present" qui transforment en champs facultatifs les noms <name> qui sont contenus dans des sortes d'éléments nommés <namedsort> et qui sont suivis du mot clé **optional** ou **default** (voir 4.2.1).
- Un opérateur de type "Make!", qui crée des valeurs de séquence ou le littéral Empty si tous les champs sont facultatifs ou si aucune sorte d'élément <elementsort> n'est spécifiée. Dans le cas du littéral Empty, toutes les occurrences de l'opérateur Make! dans les axiomes du *Modèle* sont considérées comme étant remplacées par le littéral Empty.
- Des opérateurs de type "Modify!", qui modifient des champs de valeur d'une séquence. Le nom de l'opérateur implicite pour la modification d'un champ est le nom <name> de ce champ concaténé avec l'opérateur "Modify!".

La liste d'arguments <argument list> qui correspond à l'opérateur Make! est la liste des sortes <sort> contenues dans la sorte d'élément <elementsort>, dans l'ordre des noms de champ. Ces derniers sont ordonnés selon les mêmes règles que pour les littéraux de type classe de nom (**nameclass**). Voir 5.3.1.14/Z.100.

Les sortes <sort> de champ qui sont suivies du mot clé **optional** ou **default** sont exclues de la liste de sortes. La sorte de résultat <result sort> qui correspond à l'opérateur Make! est la sorte contextuelle englobante. C'est également la sorte de la séquence.

La liste d'arguments <argument list> qui correspond à chaque opérateur de modification de champ est la sorte <sort> de la séquence, suivie de la sorte de ce champ. La sorte de résultat <result sort> qui correspond à un opérateur de modification de champ est la sorte de la séquence.

La liste d'arguments <argument list> qui correspond à chaque opérateur d'extraction de champ est la sorte de la séquence. La sorte de résultat <result sort> qui correspond à un opérateur d'extraction de champ est la sorte de ce champ.

La liste d'arguments <argument list> qui correspond à chaque opérateur de présence d'un champ est la sorte de la séquence.

La sorte de résultat <result sort> qui correspond à chaque opérateur de présence d'un champ est la sorte prédéfinie Boolean.

Pour chaque champ, il existe un axiome qui décrit la signification de l'extraction de ce champ:

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

où F est le nom du champ et où E est l'argument de l'opérateur Make! dont la position correspond au champ F si celui-ci n'est pas un champ facultatif. Si F est un champ facultatif (spécifié avec le mot clé **optional**), alors E a la valeur **error!**; et si le champ F est spécifié avec le mot clé **default**, alors E est l'expression close associée (<ground expression>). Si le champ F est facultatif et dépend d'une construction de type **components of**, alors l'argument E est l'opérateur FExtract!(<<**type** S1>> Make!(a1,b1,...,n1)), où S1 est la sorte de composant mentionnée dans la construction **components of**.

A chaque champ facultatif correspondent des axiomes qui décrivent les conséquences d'un test de présence de ce champ:

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

où F et G sont les noms de champs différents et où F est un champ facultatif. L'argument B est vrai (valeur True) si le champ F est spécifié avec le mot clé **default**. Si le champ F est dérivé de la construction **components of**, alors B a la valeur FPresent(<<**type** S1>> Make!(a1,b1,...,n1)), où S1 est la sorte mentionnée dans la construction **components of**. Dans les autres cas, l'argument B a la valeur False.

A chaque champ correspondent des axiomes qui indiquent que sa valeur ne sera pas influencée par la modification d'autres champs:

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

où F et G sont les noms de champs différents.

Il existe également un axiome qui exprime l'égalité de valeurs de séquence:

pour tous les s1, s2 en S

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

où S est la sorte de séquence, A1,...,An sont les champs non facultatifs et où B1,...,Bn sont les champs facultatifs.

Exemple:

La sorte de séquence suivante:

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

équivaut à:

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

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

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

NOTES

1 Les champs spécifiés avec le mot clé **default** ont un opérateur "Present" et sont donc traités comme un champ facultatif qui renvoie une valeur par défaut au lieu d'une erreur si on le consulte avant de lui affecter une autre valeur.

2 Les champs dérivés de la construction **components of** doivent être ordonnés d'une façon telle que l'ordre des sortes dans la liste d'arguments <argument list> de l'opérateur Make! soit déterministe. Dans la présente Recommandation, on utilise l'ordre alphabétique.

3 Aucune distinction n'est opérée entre l'usage du mot clé **sequence** et celui du mot clé **set**. Cette règle allège celle de la Recommandation X.680.

4 Dans la présente Recommandation, des étiquettes ne sont pas nécessaires pour établir une distinction entre composants du même type car on part du principe que l'étiquetage est automatiquement assuré par la notation ASN.1.

4.2.3 Sequenceof

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

Modèle

Le fait de spécifier une séquence d'éléments <sequenceof> revient à spécifier une transformation par générateur <generator transformation> dont le premier générateur réel <generator actual> est une sorte <sort> et dont le second générateur réel <generator actual> est le nom Emptystring. L'identificateur de générateur dans la structure de transformation par générateur <generator transformation> est le générateur prédéfini String (chaîne) si le mot clé **sequence** est spécifié; sinon, l'identificateur de générateur est le générateur prédéfini Bag (sac).

Si une contrainte de taille <sizeconstraint> est spécifiée, la séquence <sequenceof> est un syntype dont la condition d'intervalle <range condition> est cette contrainte de taille <sizeconstraint> (voir 4.2.7).

Si la condition <asn1 range condition> est spécifiée, la séquence <sequenceof> est un syntype dont la condition d'intervalle <range condition> est spécifiée dans la condition d'intervalle ASN.1 <asn1 range condition> (voir 4.2.7).

La sorte parente du syntype (c'est-à-dire la séquence de composants <sequenceof> sans la contrainte de taille <sizeconstraint> ou la condition d'intervalle ASN.1 <asn1 range condition>) possède un nom implicite et unique; d'autre part, elle est définie dans la plus proche unité de portée englobant le module où la structure <sequenceof> apparaît.

NOTE – Le générateur de type String est défini dans la Recommandation Z.100. Le générateur de type Bag est défini dans l'Annexe A. En principe, la relation d'ordre des éléments contenus dans un générateur Bag n'est pas significative, contrairement aux éléments d'une chaîne String. Même si les éléments sont en fait rangés dans un certain ordre à l'intérieur du générateur prédéfini de type Bag (ce qui permet d'appliquer des conditions d'intervalle à chaque élément), il n'est pas recommandé d'utiliser ces informations.

Exemple:

La définition:

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

est la même que les trois définitions suivantes, en langage SDL:

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

On notera que la construction **size** est une extension de la grammaire concrète du langage SDL (voir 4.2.7).

4.2.4 Choice

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

Modèle

Les noms <name> contenus dans les sortes d'éléments nommés <namedsort> du choix <choice> doivent être distincts. Une structure de choix <choice> est un élément syntaxique dérivé qui correspond à une expression de propriétés <properties expression> contenant les opérateurs suivants:

- des opérateurs de type "Make!", qui créent des valeurs choix;
- des opérateurs de type "Extract!", qui associent les noms <name> au choix <choice> (voir 4.2.1);
- des opérateurs de type "Present" qui transforment en champs facultatifs les noms <name> qui sont contenus dans des sortes d'éléments nommés <namedsort> (voir 4.2.1);
- des opérateurs de type "Modify!", qui modifient des composants de valeur d'un choix. Le nom de l'opérateur implicite pour la modification d'un composant est le nom <name> de ce champ concaténé avec l'opérateur "Modify!".

Il existe un seul opérateur de type "Make!" pour chaque sorte d'éléments nommés <namedsort>. Chaque opérateur "Make!" possède le nom "FMake!", où F est le champ <name> apparaissant dans la sorte d'éléments nommés <namedsort>. La liste d'arguments <argument list> de l'opérateur "FMake!" est le champ <sort> apparaissant dans la sorte d'éléments nommés <namedsort>. La sorte de résultat <result sort> pour les opérateurs de type "FMake!" est la sorte qui englobe l'expression des propriétés <properties expression>.

Les noms et les signatures des opérateurs "Extract!", "Modify!" et "Present" agissant sur les champs sont les mêmes que pour les séquences <sequence> (voir 4.2.2).

La structure de choix <choice> définit implicitement, par ailleurs, une sorte d'éléments énumérés dont les littéraux ont des noms identiques aux noms de champ contenus dans les opérateurs du choix <choice>. L'ordre de ces littéraux est celui dans lequel les champs sont spécifiés. Pour décrire le *Modèle*, les littéraux sont numérotés 1, 2, 3, ... n, où n est le nombre de champs/littéraux sauf le champ d'élément dérivé (comme défini ci-dessous). Cette sorte d'énumérés implicites contient un champ d'élément dérivé implicite qui indique quel champ explicite est présent à un moment quelconque. Le nom de ce champ est "Present".

Pour chaque champ F, il existe un ensemble d'équations décrivant la signification de l'extraction de ce champ:

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

où E a la valeur **error!** sauf pour l'équation mentionnant les opérateurs "FExtract!" et "FMake!" pour le même champ. Dans ce cas, l'argument E est égal à l'argument A.

Pour chaque champ F, il existe un ensemble d'équations décrivant la signification du test de présence:

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

où B a la valeur False sauf pour l'équation mentionnant "FPresent" et "FMake!" pour le même champ. Dans ce cas, l'argument B a la valeur True.

Pour chaque champ F, il existe un ensemble d'équations décrivant la signification d'une modification de valeur d'un choix:

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

Pour chaque champ F, il existe un ensemble d'équations décrivant la valeur du champ implicite "Present":

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

Il existe une seule équation décrivant l'égalité de valeurs d'un choix:

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

NOTES

1 Le champ d'élément dérivé, indiquant le type d'énuméré dérivé, est un champ qui donne des renseignements sur le champ actif du choix. L'opérateur "PresentExtract" peut donc être utilisé pour le multibranchement.

2 Dans la présente Recommandation, des étiquettes ne sont pas nécessaires pour établir une distinction entre composants du même type car on part du principe que l'étiquetage est automatiquement assuré par la notation ASN.1.

Exemple:

Le type CHOICE suivant:

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

équivaut à:

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

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

4.2.5 Enumerated

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

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

Les expressions simples <simple expression> contenues dans les valeurs d'élément nommé <named value> doivent être de la sorte Integer et ces valeurs doivent être disjointes.

Modèle

Un numéro d'élément nommé <named number> est un nom <name>: c'est donc un élément syntaxique dérivé qui correspond à une valeur d'élément nommé <named value> contenant le nom de cet élément ainsi qu'une expression simple <simple expression> indiquant la plus petite valeur non négative d'entier possible, n'apparaissant dans aucune autre valeur d'élément nommé <named value> de la sorte d'éléments énumérés <enumerated>. Le remplacement des noms <name> par des valeurs d'élément nommé <named value> s'effectue de gauche à droite, nom après nom. Après ce remplacement, les littéraux sont remis dans l'ordre de leurs simples expressions <simple expression>.

Une sorte d'éléments énumérés <enumerated> est représentée par une spécialisation de la sorte prédéfinie Enumeration, comme défini dans l'Annexe A. Les noms <name> contenus dans les valeurs d'élément nommé <named value> sont de nouveaux littéraux qui sont ajoutés dans l'ordre ascendant de leurs expressions simples <simple expression> associées. La sorte ainsi spécialisée possède des axiomes qui définissent les structures suivantes: First, pour indiquer la valeur du littéral ayant la plus petite expression simple <simple expression>; Last, pour indiquer la valeur du littéral ayant la plus grande expression simple <simple expression>; Succ, pour indiquer la valeur du deuxième plus grand littéral (s'il existe, sinon la valeur **error!**); Pred, pour indiquer la valeur du deuxième plus petit littéral (s'il existe, sinon la valeur **error!**); et Num pour indiquer la valeur d'entier de l'expression simple <simple expression>.

Par exemple, la définition:

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

équivaut au module:

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

NOTE – Une sorte d'énumérés possède également des opérateurs relationnels, mais comme la sorte prédéfinie Enumerated est définie avec le mot clé **ordering**, les propriétés de ces opérateurs relationnels ne sont pas indiquées dans le sous-type.

4.2.6 Integer Naming

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

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

L'identificateur <identifier> doit toujours indiquer la sorte prédéfinie Integer ou la sorte prédéfinie Bit_String; d'autre part, les expressions simples <simple expression> contenues dans les valeurs d'éléments nommés <named value> doivent toujours être de la sorte Integer.

Modèle

Le fait de spécifier une dénomination par entier <integernaming> revient à spécifier une sorte existante <existingsort> contenant l'identificateur <identifier> d'entier (Integer) et à spécifier un item de définition de synonyme <synonym definition item> pour chaque valeur d'élément nommé <named value> désignée par la plus proche unité de portée englobante.

Exemple:

La définition de nouveau type **newtype** suivante:

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

équivaut à:

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

NOTES

1 Une dénomination par entier <integernaming> permet de regrouper des définitions de synonyme ayant une certaine relation les unes avec les autres. Un tel groupage peut être utile afin de faciliter la lecture mais, du point de vue de la présente Recommandation, il n'est pas déterminant.

2 Il n'existe pas de corrélations entre les valeurs de synonyme définies par les entiers des valeurs de type <named value> et la sorte désignée au moyen d'un entier ou d'une chaîne binaire par la dénomination <integernaming>.

4.2.7 Subrange

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

Un sous-intervalle <subrange> définit l'intervalle d'un syntype. Lorsqu'un opérateur de type <sequenceof> ou <selection> est suivi de la construction **(**<range condition>**)**, le sous-intervalle <subrange> s'applique à la sorte <sort> ainsi contenue plutôt qu'à l'ensemble de la construction <sequenceof> ou <selection>.

Modèle

Le fait de spécifier un sous-intervalle <subrange> sous la forme de l'identificateur <parent sort identifier> d'un <syntype> revient à spécifier la sorte contenue et à ajouter la condition d'intervalle <range condition> après le mot clé **constants** dans le <syntype> (s'il est spécifié; sinon, il faut introduire cette construction).

Exemple:

La définition de syntype suivante:

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

équivaut à:

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

La façon dont cette construction correspond à un syntype tel que décrit dans la Recommandation Z.100 est indiquée ci-après.

4.3 Condition d'intervalle

La grammaire abstraite pour la condition d'intervalle est modifiée comme suit:

Range-condition :: Operator-identifier

Une condition d'intervalle définit un ensemble de valeurs. Elle sert à définir un syntype et à effectuer un branchement lors d'une action de décision. Elle possède une sorte parente associée qui est, pour un syntype, la sorte spécifiée dans la définition de ce syntype et qui est, pour une action de décision, l'expression de la question. Une valeur s'inscrit dans l'ensemble de valeurs si l'opérateur indiqué par l'identificateur d'opérateur renvoie la valeur True lorsqu'il est appliqué à cette valeur.

L'identificateur d'opérateur pour une condition d'intervalle donnée sera donc défini comme suit:

```
newtype A
    operators o: S -> Boolean;
    axioms
    /* Les axiomes sont dérivés de la syntaxe concrète comme expliqué ci-dessous */
endnewtype A;
```

où o est l'opérateur implicite et anonyme qui apparaît dans la condition d'intervalle et où A est une sorte implicite et anonyme, S étant la sorte parente.

La syntaxe concrète pour la condition d'intervalle est modifiée comme suit:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Que le mot clé **from** ou **with component** soit utilisé dans le champ <innercomponent> n'est pas déterminant.

Les parenthèses autour de la liste d'intervalles <range list> ne sont pas déterminantes.

L'utilisation de deux points verticaux (**:**) ou horizontaux (**..**) dans un intervalle fermé <closed range> n'est pas déterminante.

Il y a ambiguïté syntaxique entre un intervalle fermé <closed range> spécifié avec deux points verticaux (**:**) et un intervalle <range> commençant par une expression close <ground expression> qui est un primaire de type choice <choice primary> (voir 4.4.1). La solution de ce problème dépendra du contexte (voir 2.2.2/Z.100). Une expression close <ground expression> qui est une primaire de type choice doit être mise entre parenthèses lorsqu'elle apparaît sous la forme d'une valeur inférieure d'intervalle <lowerendvalue> complète (par exemple A((b:c):d)) ou <open range> (par exemple A((b:c))).

L'usage d'une virgule (**,**) ou d'une barre verticale (|) dans une condition d'intervalle <range condition> n'est pas déterminant.

Les noms <name> contenus dans les contraintes sur éléments nommés <named constraint> d'un champ de composants internes <innercomponents> doivent correspondre à des noms de champs facultatifs (voir 4.2.1) de la sorte parente.

Les noms de champ doivent être distincts.

Chaque intervalle <range> de la condition d'intervalle <range condition> contribue aux propriétés de l'opérateur qui définit l'ensemble de valeurs:

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

Si un syntype est spécifié sans condition d'intervalle <range condition>, le résultat de l'opérateur est True.

Dans l'explication suivante de la manière dont chaque intervalle <range> contribue au résultat d'opérateur, la lettre V désigne la valeur de l'argument. Chaque contribution doit être bien formée, c'est-à-dire que les opérateurs utilisés doivent exister avec une signature appropriée au contexte.

- Si aucun des mots clés **min** et **max** n'est spécifié dans un intervalle fermé <closed range>, celui-ci apporte la contribution suivante:

  E1 rel1 V et V rel2 E2

  où E1 est l'expression close <ground expression> de la valeur inférieure de l'intervalle <lowerendvalue> et où E2 est l'expression close <ground expression> de la valeur supérieure de l'intervalle <upperendvalue> de l'intervalle.

  Si le symbole "<" est spécifié pour la structure <lowerendvalue>, l'argument rel1 est l'opérateur "<"; sinon, c'est l'opérateur "<=".

  Si le symbole "<" est spécifié pour la structure <upperendvalue>, l'argument rel2 est l'opérateur "<"; sinon, c'est l'opérateur "<=".

  Si le mot clé **min** est spécifié et que le mot clé **max** ne le soit pas, la condition d'intervalle <range condition> contribue au résultat de:

  V rel2 E2

  Si le mot clé **max** est spécifié et que le mot clé **min** ne le soit pas, la condition d'intervalle <range condition> contribue au résultat de:

  E1 rel1 V

  Si les deux mots clés **min** et **max** sont spécifiés, l'opérateur renvoie toujours la valeur True.

- Un intervalle ouvert <open range> apporte la contribution suivante au résultat:

  V rel E

  où E est l'expression close <ground expression> de l'intervalle ouvert <open range> et où rel est l'opérateur infixe de l'intervalle ouvert. Si aucun opérateur infixe n'est spécifié dans l'intervalle ouvert, rel est l'opérateur =.

- Un sous-intervalle contenu <contained subrange> apporte la contribution suivante:

  o1(V)

  où o1 est l'opérateur implicite qui définit l'ensemble de valeurs pour la sorte mentionnée dans le sous-intervalle contenu.

- Une contrainte de taille <sizeconstraint> apporte la contribution suivante:

  o1(Length(V))

  où o1 est l'opérateur implicite qui définit l'ensemble de valeurs pour la condition d'intervalle mentionnée <range condition> dans la contrainte de taille <size constraint>.

- Un composant interne <innercomponent> apporte soit la contribution:

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

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

  selon ce qui convient pour la sorte de V. La lettre o désigne l'opérateur implicite auquel le composant interne <innercomponent> contribue et o1 désigne l'opérateur implicite pour la condition d'intervalle spécifiée dans <innercomponent>.

  Les composants internes <innercomponents> apportent une contribution pour chaque contrainte sur élément nommé <named constraint> contenue, qui spécifie des contraintes sur le champ (voir 4.2.1) désigné par le nom <name> de la sorte parente.

  Si le symbole **...** est omis, le mot clé **present** est ajouté aux contraintes sur éléments nommés <named constraints> qui ne possèdent pas de mot clé final (comme **present**, **absent** ou **optional**) et on ajoute des contraintes sur élément nommé <named constraint> de la forme:

  <name> **absent**

  pour tous les champs (c'est-à-dire les noms) non mentionnés explicitement dans une contrainte sur élément nommé <named constraint>. Les contraintes de type <named constraint> sont ajoutées à la structure <innercomponents> avant la détermination de la contribution apportée par chaque contrainte de type <named constraint>.

Si une condition d'intervalle est spécifiée pour une contrainte de type <named constraint>, la contribution apportée sera la suivante:

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

où E est la contrainte présente pour le champ, F (venant du nom de l'opérateur FPresent) est le nom du champ facultatif et où o1 est l'opérateur implicite pour la condition d'intervalle <range condition>. Si celle-ci est omise, la contribution n'est que la contrainte E présente pour le champ.

La contrainte présente pour un champ F est:

FPresent(V)

si la contrainte <named constraint> contient pour ce champ le mot clé **present**; ou

not FPresent(V)

si la contrainte <named constraint> contient pour ce champ le mot clé **absent**. Dans tous les autres cas, la contrainte présente a la valeur True.

4.4 Expressions des valeurs

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

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

NOTES

1 Dans la production <extended primary>, les variantes <choice primary> et <composite primary> s'ajoutent à celles de la Recommandation Z.100.

Dans la production <extended identifier>, la variante <string primary> s'ajoute à celles de la Recommandation Z.100.

2 La production <extended primary> fait partie d'expressions, tandis que la production <extended literal identifier> fait partie de termes d'équations.

4.4.1 Choice primary

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

Modèle

Une production de type <choice primary> est un élément syntaxique dérivé qui correspond à une application d'opérateur <application operator> dont l'argument est un primaire <primary>. L'identificateur d'opérateur <operator identifier> contenu dans l'application d'opérateur <operator application> contient le qualificateur <qualifier> de l'identificateur <identifier> et un nom d'opérateur qui est le nom <name> contenu dans l'identificateur, suivi de "Make!".

Exemple:

Le primaire de choix <choice primary>:

Myvalue : Mychoice

est une syntaxe dérivée qui correspond à

MyvalueMake!(Mychoice)

Si un primaire de choix <choice primary> particulier peut désigner une application d'opérateur parmi plusieurs (c'est-à-dire un champ contenant plus d'une seule sorte de choix), un qualificateur est utilisé comme suit:

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

dont la syntaxe dérivée devient alors:

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

NOTE – Les noms d'opérateur composés d'un nom de champ suivi de "Make!" peuvent être utilisés dans chaque sorte construite au moyen d'un choix <choice>. La notation <choice primary> sera donc utilisée surtout pour de telles sortes. Cette notation convient cependant pour des valeurs de type **any**; par exemple, on peut définir un opérateur comme RealMake! au moyen de la signature suivante:

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

ce qui implique que la notation

Real : <expression>

peut être utilisée pour «convertir» en valeur de réel une expression de type **any**.

4.4.2 Composite Primary

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

Un primaire composite <composite primary> est une valeur construite au moyen de ses éléments constituants. Il y a ambiguïté syntaxique entre les diverses variantes contenues dans la production <composite primary>. C'est alors le contexte qui permettra de déterminer ce qui est visé par le primaire composite (voir 2.2.2/Z.100).

NOTE – Cela implique probablement que les outils logiciels actuels ne permettent de distinguer, en syntaxe concrète, que les valeurs de séquence <sequencevalue> et les séquences de valeurs <sequenceofvalue>. Dans le cadre de l'analyse statique, on pourra déterminer si une séquence de valeurs <sequenceofvalue> pointe au contraire sur une des variantes restantes.

4.4.2.1 Sequence value

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

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

La sorte de la valeur de séquence <sequencevalue> doit toujours avoir, comme champs associés (voir 4.2.1) les noms <name> apparaissant dans les valeurs d'élément nommé <namedvalue>. L'expression <expression> contenue dans chaque valeur d'élément nommé <namedvalue> doit toujours être de la même sorte que le champ indiqué par le nom. Chaque champ non facultatif associé à la sorte doit être mentionné exactement une seule fois et chaque champ facultatif associé à la sorte doit être mentionné au plus une seule fois.

Modèle

Une valeur de séquence <sequencevalue> est un élément syntaxique dérivé qui correspond à ce qui suit:

- une application d'opérateur dont l'opérateur a le nom "Make!". Ses arguments seront les expressions des champs non facultatifs. Les expressions contenues dans l'application d'opérateur dérivée apparaissent en ordre alphabétique, triées par nom de champ. Si tous les champs associés à la sorte sont facultatifs, la construction syntaxique est un identificateur de littéral dont le nom est "Empty";
- si la sorte possède des champs facultatifs, l'application d'opérateur sert de premier argument d'une application d'opérateur dénommé "FModify!", où F est, dans l'ordre alphabétique, le premier nom des champs facultatifs mentionnés dans la valeur de séquence <sequencevalue>. Le deuxième argument de l'opérateur "FModify!" est l'expression associée au champ. L'application d'opérateur résultante devient à son tour le premier argument de l'opérateur "FModify" du champ facultatif suivant et ainsi de suite jusqu'à ce que tous les champs facultatifs aient été pris en considération.

Exemple:

Si la sorte S comporte quatre champs A, B, C, D associés et que les champs C et D soient facultatifs, le primaire composite <composite primary> suivant:

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

sera un élément syntaxique dérivé correspondant à:

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

L'ordre des valeurs d'élément nommé <namedvalue> dans la valeur de séquence <sequencevalue> n'est pas déterminant; par exemple:

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

correspond à la même expression que ci-dessus. D'autre part, le qualificateur <qualifier> peut être omis:

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

auquel cas il doit toujours être possible de déterminer, d'après le contexte (voir 2.2.2/Z.100), de quelle sorte il s'agit.

NOTE – Les valeurs de séquence <sequencevalue> sont le plus souvent utilisées pour construire des valeurs de sortes définies avec la construction <sequence> car de telles sortes ont un "Make!" opérateur dont les arguments sont des valeurs de champ triées dans l'ordre alphabétique des noms de champ. Pour une sorte définie au moyen de la production <structure definition>, il est souvent préférable d'utiliser le primaire de structure <structure primary> car l'ordre d'apparition est important pour l'opérateur "Make!" de telles sortes.

4.4.2.2 Sequenceof value

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

Modèle

Une valeur d'élément de séquence <sequenceofvalue> est un élément syntaxique dérivé qui correspond à l'expression:

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

où E1, E2, ..., En sont les expressions de la production <sequenceofvalue>, triées dans l'ordre de leur apparition. Si aucune <expression> n'est spécifiée, la production <sequenceofvalue> devient l'élément syntaxique dérivé pour le nom Emptystring.

NOTE – Une valeur <sequenceofvalue> peut indiquer une valeur de la sorte prédéfinie Octet_string, bien que la Recommandation X.680 (contrairement à la présente Recommandation) prescrive que chaque <expression> soit, dans ce cas, un identificateur <identifier>.

4.4.2.3 Object identifier value

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

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

Modèle

Si le composant identificateur d'objet <objidcomponent> situé le plus à gauche est un identificateur <identifier> de synonyme ou de variable de la sorte prédéfinie Object_identifier, au moins un composant identificateur d'objet <objidcomponent> doit suivre. La valeur de l'identificateur d'objet <objectidentifiervalue> équivaudra alors à l'expression suivante:

Id // {Olist}

où Id est le premier composant de type <objidcomponent> et où Olist représente les autres composants de type <objidcomponent>.

Si un composant identificateur d'objet <objidcomponent> est spécifié avec une expression close <ground expression>, c'est alors un élément syntaxique dérivé qui définit l'identificateur <identifier> comme étant un synonyme (**synonym**) pour l'expression close <ground expression> située dans la plus proche unité de portée englobante, compte non tenu de l'expression close <ground expression> située dans le composant identificateur d'objet <objidcomponent>.

Après la transformation ci-dessus, tous les composants identificateurs d'objet <objidcomponent> sont des identificateurs <identifier> qui doivent renvoyer à une variable ou à un synonyme de la sorte prédéfinie Object_element (voir l'Annexe A). La construction est alors un élément syntaxique dérivé qui équivaut à l'expression suivante:

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

où Id1, Id2, ..., Idn sont les identificateurs de la valeur d'identificateur d'objet <objectidentifiervalue>, indiqués dans le même ordre que dans cette valeur.

Exemple:

La valeur d'identificateur d'objet <objectidentifiervalue> suivante:

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

équivaut à:

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

où le terme root est une variable ou un synonyme de la sorte prédéfinie Object_Identifier et une définition de synonyme de niveau 4 est construite dans la plus proche unité de portée englobante:

synonym level4 Object_element = 4;

4.4.2.4 Real value

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

<mantissa> ::= <expression>

<base> ::= <simple expression>

<exponent> ::= <expression>

Ces trois expressions constituantes doivent être de la sorte Integer. La <base> doit toujours avoir la valeur 2 ou la valeur 10.

Modèle

Une valeur de réel <real value> est représentée par l'<expression> suivante: (<mantissa>)*Power(<base>,<exponent>) où le terme Power est un opérateur additionnel qui est défini pour la sorte Real.

L'opérateur possède la signature suivante:

Power : Integer, Integer -> Real;

et les équations:

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

Exemple:

L'expression:

{3,2,-1}

équivaut à:

3*Power(2,-1)

dont le résultat est égal à 1,5

4.4.3 String Primary

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

Les termes <bitstring> et <hexstring> sont les littéraux des sortes prédéfinies Bit_string et Octet_string (voir l'Annexe A).

C'est le contexte ou l'utilisation d'un qualificateur <qualifier> qui permettra de déterminer si une chaîne binaire <bitstring> ou hexadécimale <hexstring> particulière est un littéral de la sorte Bit_string ou Octet_string.

Modèle

Un primaire de type chaîne <string primary> contenant une chaîne citée <quoted string> représente un identificateur de littéral d'une chaîne de caractères <character string literal identifier> composé du qualificateur <qualifier> et d'un littéral de chaîne de caractères <character string literal> ayant le même texte <text> que la chaîne citée <quoted string>.

Un primaire de type chaîne <string primary> contenant une chaîne binaire <bitstring> ou une chaîne hexadécimale <hexstring> représente un identificateur composé du qualificateur <qualifier> et d'un nom <name> que l'on forme en enlevant les apostrophes de citation de la chaîne binaire <bitstring> ou hexadécimale <hexstring>.

NOTES

1 Une chaîne citée <quoted string> est une chaîne de caractères où des apostrophes doubles (") ont été utilisées au lieu d'apostrophes simples (').

2 A titre indicatif, il y a lieu que les notations de valeurs pour chaînes binaires ou hexadécimales utilisent des apostrophes simples (par exemple: '01'B au lieu de: 01B).

Remplacée par une version plus récente

Annexe A

Langage SDL en combinaison avec des données ASN.1 prédéfinies

(Cette annexe fait partie intégrante de la présente Recommandation)

La présente annexe définit le progiciel de prédéfinition Predefined, qui contient les sortes de données prédéfinies et les générateurs de données prédéfinies.

```
package Predefined

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

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

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

/* String0 generator */

/* Definition            */

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

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

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

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

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

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

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

/*   Usage */

/*

Un générateur de chaîne vide (string0) peut être utilisé pour définir des chaînes appartenant à n'importe quelle sorte d'item. La différence par rapport à un générateur de chaîne (String) (défini dans la Recommandation Z.100) est la position '0' du premier élément. Conformément à la Recommandation X.680, les chaînes binaires en notation ASN.1 commencent à partir de la position 0.

Les opérateurs Extract! et Modify! seront normalement utilisés en notation abrégée conformément aux 5.3.3.4/Z.100 et 5.4.3.1/Z.100 pour accéder aux valeurs des chaînes et pour modifier ces valeurs.
*/

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

/*   Definition of ASN.1 character strings          */

syntype
    IA5String  =  Charstring
endsyntype;

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

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

endsyntype;

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

endsyntype;

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

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

/*      Usage      */

/*

Ces sortes définissent les chaînes de caractères en notation ASN.1. On notera que seuls les caractères de l'Alphabet international n° 5 peuvent être utilisés car le jeu de caractères du langage SDL est limité.
*/

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

/*      Usage                    */

/*
```

Les définitions ASN.1 de type INTEGER peuvent déclarer des constantes additionnelles:
INTEGER { name-1(number-1), ... ,name-n(number-n) }
Such constants are defined using synonyms:

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

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

/*   Enumeration sort */

/*   Definition    */

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

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

*/

synonym  PLUS_INFINITY        Real  =  external,
         MINUS_INFINITY       Real  =  external;

/*      Usage      */

/*

La sorte real est utilisée pour représenter des nombres réels accessibles par langage SDL comme par notation ASN.1. Les notations de valeurs ASN.1 { mantissa, base, exponent } sont ainsi traduites:
                    mantissa * Power (base, exponent)
*/

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

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

/* Bag generator  */

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

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

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

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

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

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

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

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

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

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

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

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

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

/*  Usage       */

/*
```

Les sacs servent à représenter des ensembles multiples. Par exemple, la notation:

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

peut être utilisée pour une variable qui peut être vide ou contenir les valeurs suivantes: (True), (False), (True,False) (True,True), (False,False),... .
Les sacs sont utilisés pour représenter la construction SET OF de la notation ASN.1.
*/

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

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

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

/* ASN.1 Bit sorts */

/* Definition */

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

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

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

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

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

/*  Usage */

/* Les chaînes binaires ASN.1 sont représentées par le nouveau type Bit_String.

Les bits nommés peuvent être fournis par des synonymes:
BIT STRING { name-1(number-1), ... ,name-n(number-n) }

Les bits nommés peuvent être représentés comme suit:

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

/* ASN.1 Octet sorts  */

/* Definition     */

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

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

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

/* Usage */

/* Les chaînes d'octets ASN.1 peuvent être représentées par le nouveau type Octet_String */

/* ASN.1 Null sort */

/* Definition */

newtype Null
    literals Null;
endnewtype Null;

/* Usage */

/* Ce type peut être utilisé pour signaler une information de présence. La déclaration d'une variable de la sorte Null n'est pas utile */

/* ASN.1 Object Identifier sort */

/* Definition */

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

newtype Object_Identifier String (object_element, EmptyString )
endnewtype Object_Identifier;

/* Usage */

/*

Il y a lieu que les valeurs d'identificateur d'objet soient supportées conformément aux règles ASN.1. L'utilisation de formes nommées NamedForm est assurée par les synonymes. Cette définition ne fait aucune référence à la structure mondiale des informations, c'est-à-dire que ce sémantème supplémentaire ne fait pas partie du domaine d'application de la combinaison SDL/ASN.1.

On notera que l'expression synonym ccitt object_element = 0;
implique que la valeur (. ccitt, ccitt, ccitt .) est valide en SDL.
*/

/* ASN.1 ANY sort */

/* Definition */

newtype Any_type
endnewtype Any_type;

/* Usage */

/*

Le type ANY de l'ASN.1 est appliqué sur la sorte Any_type. Noter que les valeurs de cette sorte sont incompatibles avec d'autres valeurs, c'est-à-dire qu'on ne peut affecter et appliquer que les opérateurs '=', '/='. Par exemple, il est possible de prendre en compte des signaux à contenu inconnu:
```

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

*/

/*   ASN.1 Useful types */

GeneralizedTime      ::=   VisibleString;

UTCTime              ::=   VisibleString;

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

ObjectDescriptor::=  GraphicString;

endpackage Predefined;
```

Appendice I

Restrictions relatives à la notation ASN.1 et au langage SDL

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

I.1 Résumé du sous-ensemble supporté de la notation ASN.1

Ce paragraphe contient un résumé du sous-ensemble ASN.1 qui est supporté dans la présente Recommandation.

En principe, la version de la notation ASN.1 définie dans la Recommandation X.680 est supportée. Les caractéristiques des Recommandations X.681, X.682 et X.683 ne sont pas supportées, c'est-à-dire que la spécification des objets informationnels, le paramétrage des spécifications ASN.1 et la spécification des contraintes ne sont pas pris en charge. De plus, le type ANY, qui est défini dans la Recommandation X.208, est partiellement supporté.

Les principales restrictions imposées à la notation ASN.1 sont les suivantes:

- Les macros ne sont pas supportées. C'est-à-dire que la macro d'opération n'est pas non plus supportée. En langage SDL, la macro d'opération n'est pas nécessaire parce qu'il existe des procédés plus appropriés à la définition des opérations, à savoir l'échange de signaux ou les procédures distantes. L'Appendice II donne un exemple de remplacement de la macro d'opération par des constructions syntaxiques du langage SDL.
- Il est permis d'utiliser des étiquettes de balisage mais celles-ci n'ont aucune signification et sont ignorées. L'explication est que ces étiquettes ne sont utilisées que pour le codage et que celui-ci est hors du domaine d'application du langage SDL.
- Les règles de codage sont hors du champ d'application de la présente Recommandation. Il est possible que des vendeurs d'outils logiciels supportent les règles de codage, mais cela dépendra de la mise en œuvre choisie.
- Aucune notation de valeur n'est supportée pour le type ANY (ou ANY DEFINED BY). L'introduction d'opérateurs spéciaux permet dans certains cas d'imiter une forme limitée de notation de valeur pour ANY, comme montré au 4.4.1. Dans certains cas, on peut remplacer le type ANY par un type CHOICE offrant un choix parmi les types pour lesquels une notation de valeur est requise. L'Appendice II en donne un exemple.

- Le tiret n'est pas admis dans les noms en notation ASN.1 situés à l'intérieur de descriptions en langage SDL. Il est permis d'utiliser des tirets dans des noms situés à l'intérieur de modules ASN.1 qui sont importés dans le langage SDL. Mais, dans ce cas, il y aura lieu de transformer ces tirets en soulignements. L'explication de cette restriction est qu'en langage SDL, le tiret est considéré comme un opérateur de soustraction.

- La sensibilité à l'inversion haut/bas de casse n'est pas supportée. L'explication de cette restriction est que le langage SDL ne fait pas cette distinction.

  Cette restriction implique que l'introduction de deux types ayant le même nom (en dehors de la sensibilité à l'inversion haut/bas de casse) sera considérée comme une erreur. Il est toutefois permis d'avoir un même nom pour des classes d'entités différentes. Les entités de ces classes seront par exemple des noms de type, des noms de valeur et des identificateurs, c'est-à-dire que la notation suivante:

  ```
  SameName ::= INTEGER {sameName (0)}
  sameName SameName := sameName
  ```

  est autorisée.

- Les définitions doivent toujours se terminer par un point-virgule. La raison en est que les définitions en langage SDL se terminent ainsi. Si le point-virgule n'était pas obligatoire, la grammaire serait ambiguë car le langage SDL ne fait pas la distinction entre haut et bas de casse.

- Le type ASN.1 REAL n'est pas représenté sous forme d'une séquence de trois entiers, comme c'est le cas dans la Recommandation X.680. La notation de valeur de la Recommandation X.208 doit être utilisée dans ce cas, c'est-à-dire qu'il y a lieu d'écrire { 314, 10, –2 } au lieu de { mantisse 314, base 10, exposant – 2 }. En variante, on peut utiliser la syntaxe SDL pour écrire des valeurs de réel, c'est-à-dire que la forme 3.14 est également permise. En conséquence, aucun sous-typage de mantisse, de base ou d'exposant n'est permis et aucun opérateur n'est pris en charge pour accéder à la mantisse, à la base ou à l'exposant d'une valeur de réel ou pour les modifier.

- Il faut éviter d'utiliser, dans la même unité de visibilité, le même identificateur pour des numéros ou bits de différents types d'éléments nommés. La raison en est que ces numéros ou bits seront appliqués sur des synonymes d'entiers en langage SDL. Le fait d'utiliser deux fois le même identificateur provoquerait un résultat d'erreur en SDL (redéfinition du même synonyme).

  Autrement dit, il n'est pas permis d'utiliser deux fois l'élément 'notAllowed' dans les définitions de type ci-dessous:

  ```
  Int1 ::= INTEGER { notAllowed(0) }
  Int2 ::= INTEGER { notAllowed(0) }
  ```

  ni dans les définitions ci-dessous:

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

  Il est permis d'utiliser deux fois le même identificateur dans des types différents d'éléments énumérés ou dans un type d'énuméré et dans un entier ou bit d'élément nommé, car les identificateurs contenus dans des types d'énumérés ne sont pas appliqués sur des entiers synonymes. C'est-à-dire que les formes suivantes sont autorisées:

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

- Dans les références externes de types et de valeurs, des espaces doivent encadrer le point ("."); on écrira par exemple: Modulereference . Typereference plutôt que Modulereference.Typereference. La non-insertion des caractères d'espace provoquerait des problèmes syntaxiques en langage SDL, parce que dans ce langage, il peut y avoir des points dans des identificateurs.

- Les valeurs de composant attribuées par l'UIT-T, par l'ISO ou par les deux à un identificateur d'objet (OBJECT IDENTIFIER) ne sont pas définies dans le progiciel Predefined parce qu'elles ne pourront pas être tenues à jour dans la présente Recommandation. Par exemple, pour utiliser l'arc:

  ```
  { ccitt recommendation z }
  ```

  il faudra définir les valeurs des composants ccitt, recommendation et z dans un progiciel d'utilisateur.

I.2 Résumé du sous-ensemble supporté du langage SDL

Ce paragraphe contient un résumé du sous-ensemble du langage SDL qui est supporté dans la présente Recommandation.

L'usage complet du langage SDL est autorisé, avec les exceptions suivantes:

- l'utilisation des caractères {, }, [,] et | (barre verticale) dans des noms n'est pas autorisée parce qu'ils ont une signification particulière en notation ASN.1;
- la présente Recommandation introduit plusieurs mots clés nouveaux qui ne peuvent pas être utilisés comme noms. Ces mots clés sont énumérés à l'article 3;
- un point situé dans un nom ne doit pas être suivi d'un soulignement ou d'un autre point. Un caractère de soulignement dans un nom ne doit pas être suivi d'un point, c'est-à-dire que les noms suivants ne sont pas autorisés:

  invalid..name, invalid._name, invalid_.name
- l'utilisation de la double négation (par exemple 1--2) n'est pas supportée parce que le double tiret (--) est utilisé pour les commentaires ASN.1;
- les séparateurs et les commentaires ne sont pas autorisés à l'intérieur de la définition d'un opérateur cité, parce que cela provoquerait une ambiguïté syntaxique. Par exemple, la notation suivante n'est pas autorisée:

  operators

  "+/* infix operator */": MyType, MyType -> MyType
- un soulignement dans un nom faisant partie d'un primaire composite doit toujours être spécifié explicitement. Par exemple, pour la définition de type suivante:

  ```
  T ::= SEQUENCE
        {a_b INTEGER}
  ```

  la notation de valeur suivante n'est pas autorisée:

  tT ::= {a b 5}

  et il faut écrire au contraire:

  tT ::= {a_b 5}

Appendice II

Exemples

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

Le présent appendice contient des exemples d'utilisation de la notation ASN.1 en combinaison avec le langage SDL. Dans cet appendice, on part du principe que le lecteur est familier de la notation ASN.1 et du langage SDL. Seule la représentation graphique du SDL sera utilisée.

L'objet du présent appendice est d'illustrer les principes d'utilisation du langage SDL en combinaison avec la notation ASN.1. Les principaux points traités sont les suivants:

- définition des types de données ASN.1 dans des diagrammes SDL;
- définition de valeurs ASN.1 dans des diagrammes SDL;
- utilisation des opérateurs définis dans la présente Recommandation pour agir sur des types de données ASN.1;
- neutralisation de certaines restrictions imposées par la présente Recommandation à la notation ASN.1;
- élaboration d'un «guide de style» pour utiliser la notation ASN.1 en combinaison avec le langage SDL.

L'intention n'est pas de donner une vue d'ensemble complète de tous les opérateurs agissant sur tous les types de données ASN.1. On trouvera ces descriptions dans d'autres parties de la présente Recommandation.

II.1 Définitions en langage SDL des types de données ASN.1

Il existe trois façons de définir en langage SDL des types de données ASN.1:

1) en important une définition de type d'un module ASN.1;
2) en définissant directement le type en langage SDL, au moyen de la syntaxe ASN.1;
3) en définissant directement le type en langage SDL, à l'intérieur d'une clause de nouveau type (newtype) SDL.

II.1.1 Importation d'une définition de type à partir d'un module ASN.1

Un type ASN.1 qui est défini dans un module ASN.1 peut être importé en langage SDL au moyen de la construction IMPORTS, qui peut être écrite en symboles de texte SDL.

Les types Address, DisconReason et DisconIndParameters sont définis dans le module ASN.1 ProtocolData qui est reproduit ci-dessous:

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

Address ::= GraphicString;

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

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

END
```

Ces types peuvent être importés en langage SDL comme indiqué sur la Figure II.1.1.1.

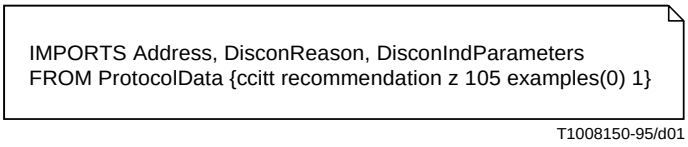

FIGURE II.1.1.1/Z.105

Importation de types ASN.1 en langage SDL

II.1.2 Définition directe en SDL de types ASN.1

Un type ASN.1 peut être défini directement en langage SDL (c'est-à-dire sans être importé d'un module ASN.1), au moyen de la syntaxe ASN.1 normale. La seule différence est qu'il faut mettre un point-virgule à la fin d'une définition de type. Les définitions de nouveaux types de données sont écrites en mode texte, comme illustré dans la Figure II.1.2.1 ci-dessous.

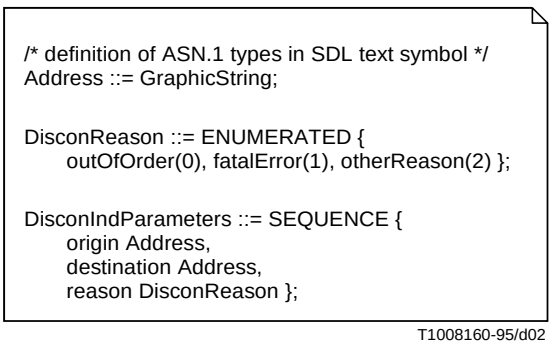

FIGURE II.1.2.1/Z.105

Définition de types ASN.1 directement en SDL

De même, on peut déclarer des variables comme étant de type ASN.1, comme illustré dans la Figure II.1.2.2 ci-dessous.

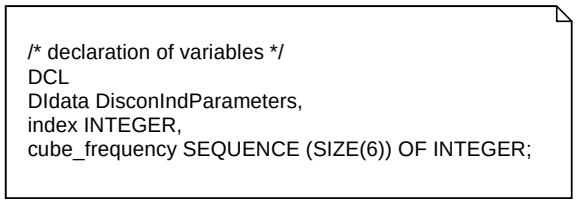

FIGURE II.1.2.2/Z.105

Déclaration de variables de types ASN.1

II.1.3 Opérateurs définis par l'utilisateur pour agir sur des types ASN.1

Il est possible de définir en SDL des types ASN.1 et d'ajouter à ces types des opérateurs définis par l'utilisateur. A cette fin, on peut définir directement le type ASN.1 en SDL, à l'intérieur d'une clause de *newtype*.

Par exemple, on prend un type Rectangle qui contient des informations sur la longueur et sur la largeur d'un rectangle. On peut définir ce type ASN.1 comme suit:

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

Il est possible de définir en SDL un opérateur de type 'surface' qui renvoie la surface (longueur multipliée par largeur) du rectangle. On peut y arriver de deux manières:

1) *Introduction d'un nouveau type qui hérite du type Rectangle*

   Si le type Rectangle est déjà défini quelque part (par exemple dans un module ASN.1), un nouveau type peut être introduit, qui hérite du type Rectangle et qui ajoute un nouvel opérateur. C'est ce qu'illustre la Figure II.1.3.1, où le type Rectangle2 hérite de Rectangle mais possède aussi un opérateur. Celui-ci ne peut être appliqué qu'à des valeurs de type Rectangle2 et non sur des valeurs de type Rectangle.

   On notera que, dans l'axiome, on utilise l'opérateur Make!(len, wid) et non pas la notation de valeur ASN.1 {length len, width wid}.

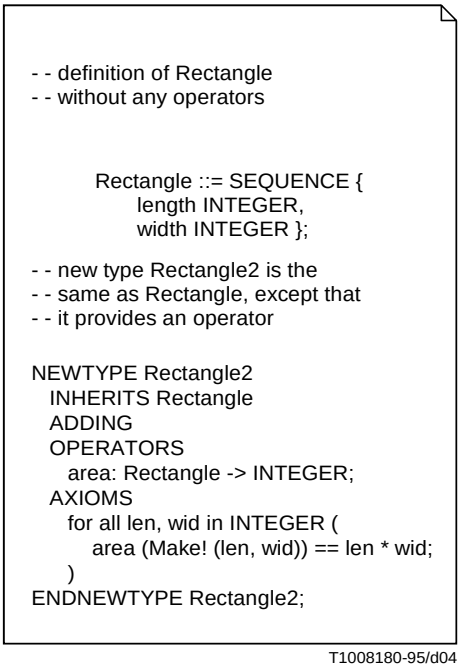

FIGURE II.1.3.1/Z.105

Héritage d'un type existant et adjonction d'un opérateur

2) *Définition directe du type Rectangle et de son opérateur*

Si le type Rectangle n'est pas encore défini, il est possible de donner une seule définition de type pour ce type et son opérateur. A cette fin, on définit le nouveau type Rectangle dans le cadre de la construction newtype du SDL et l'opérateur est ajouté après la définition des champs, comme indiqué sur la Figure II.1.3.2 ci-dessous.

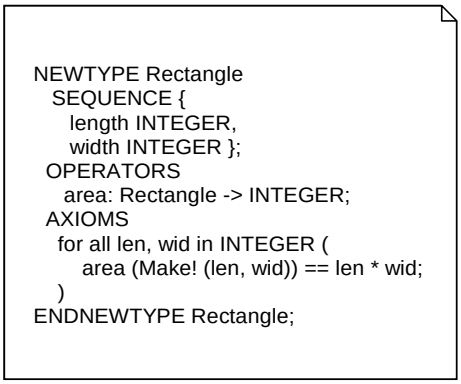

FIGURE II.1.3.2/Z.105

Définition d'opérateurs agissant sur des types ASN.1

II.2 Utilisation en langage SDL de valeurs en notation ASN.1

On peut utiliser des valeurs ASN.1 dans des expressions en langage SDL. Les expressions sont utilisées à un certain nombre d'endroits du texte SDL, par exemple:

- dans une affectation à une variable;
- dans des symboles décisionnels;
- dans les paramètres de sortie d'un signal, d'un appel de procédure ou d'une création de processus;
- dans la définition de constantes.

Le moyen le plus simple pour utiliser des valeurs ASN.1 consiste à utiliser la notation normale de valeurs ASN.1. La Figure II.2.1 ci-dessous montre l'utilisation d'une valeur ASN.1 dans une sortie de signal SDL de type DisconInd (indication de déconnexion). On suppose que ce signal contient, en tant que paramètre, l'élément DisconIndParameters qui est défini au II.1.1.

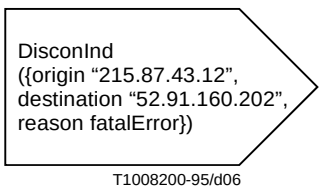

FIGURE II.2.1/Z.105

Valeur ASN.1 dans une sortie de signal SDI

La notation de valeur ASN.1 peut également être utilisée pour définir des constantes en langage SDL, exactement comme pour définir des valeurs en notation ASN.1, ce qui est illustré dans la Figure II.2.2 ci-dessous.

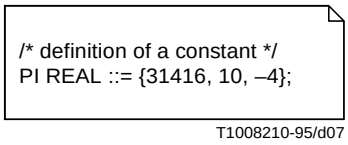

FIGURE II.2.2/Z.105

Définition d'une constante au moyen de la notation de valeur ASN.1

II.3 Illustration de l'utilisation en langage SDL de certains types ASN.1

La notation ASN.1 n'a pas d'opérateurs. En langage SDL, on a besoin d'opérateurs agissant sur des données ASN.1 afin de comparer et de manipuler de telles données. Dans ce paragraphe, on décrira l'utilisation de certains opérateurs sur des types ASN.1. Comme le corps de la présente Recommandation, l'Annexe A définit l'ensemble complet des opérateurs prédéfinis et leur signification.

II.3.1 Opérateurs agissant sur des types ASN.1 simples

Comparaison de valeurs de données

Pour chaque type de données ASN.1, au moins deux opérateurs sont disponibles: = (égal) et /= (non égal). Tous les types ordonnés ont également des opérateurs: < (inférieur à), > (supérieur à), <= (inférieur ou égal à) et >= (supérieur ou égal à).

Les opérateurs de comparaison sont fréquemment combinés avec des opérateurs booléens (et, ou, non, etc.), par exemple dans des symboles de décision. La Figure II.3.1.1 montre trois exemples d'opérateurs de comparaison.

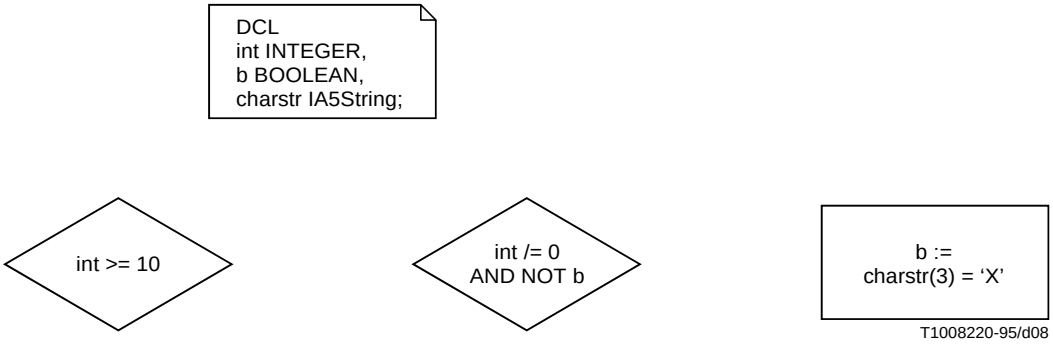

FIGURE II.3.1.1/Z.105

Opérateurs de comparasion et opérateurs booléens

Manipulation de valeurs

Il existe un certain nombre d'opérateurs définis pour agir sur des types ASN.1 simples, comme +, –, * et / qui sont applicables à des entiers et à des réels. La Figure II.3.1.2 en montre quelques-uns:

- + pour additionner deux entiers (par exemple index + 1 dans une liste d'indices) ou deux réels;

- // pour concaténer des chaînes de caractères (préfixe // "X" // suffixe) ou des chaînes d'éléments binaires ('03FC'H // Mkstring(1)). Cet opérateur est applicable à tous les types de chaîne (c'est-à-dire aux chaînes de caractères et aux types BIT STRING, OCTET STRING, SEQUENCE OF);
- Mkstring pour construire une chaîne composée d'un seul élément (Mkstring(1)). Cet opérateur est applicable à tous les types de chaîne.

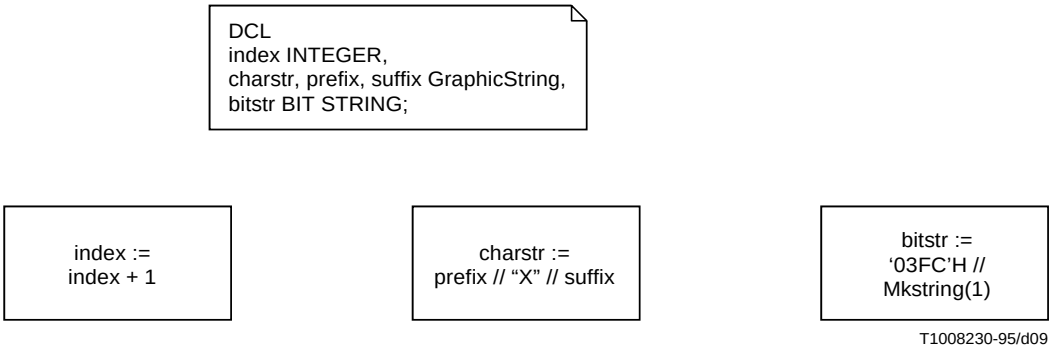

FIGURE II.3.1.2/Z.105

Exemples d’opérateurs sur types simples

L'Annexe A définit tous les opérateurs sur types simples.

II.3.2 SEQUENCE

Pour les types définis par SEQUENCE, un opérateur permet d'accéder à la valeur de chaque composant. Prenons par exemple le type ConReqData (données de demande de connexion), qui est défini dans la spécification ASN.1 partielle ci-dessous. Le composant "Receiver" de la variable v du type ConReqData peut être affecté d'une valeur et être appelé par l'opérateur v!Receiver comme illustré dans la Figure II.3.2.1.

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

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

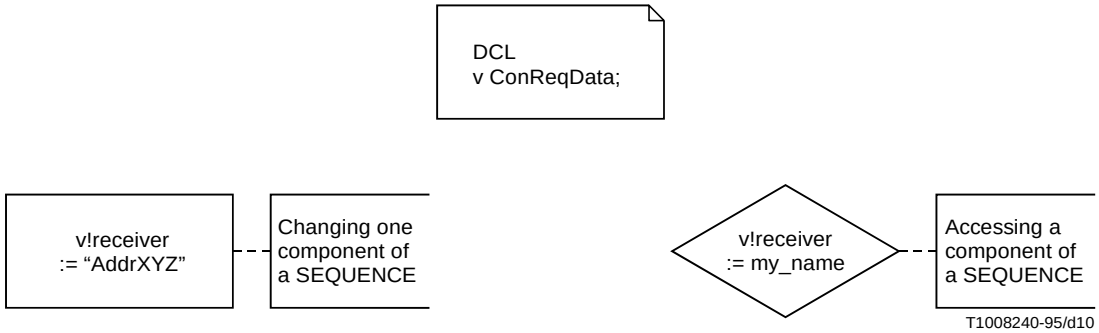

FIGURE II.3.2.1/Z.105

Accès à un composant de SEQUENCE

La lecture d'un composant par défaut est toujours possible, même si sa valeur est absente: dans ce cas, c'est la valeur par défaut qui est renvoyée.

La lecture d'un composant facultatif exige des précautions particulières: une erreur dynamique se produit si le composant est absent. Avant d'accéder à un composant facultatif, il convient toujours de rechercher si ce composant est présent ou non. C'est à cette fin que l'on peut utiliser l'opérateur '<identifier>Present', comme illustré dans la Figure II.3.2.2.

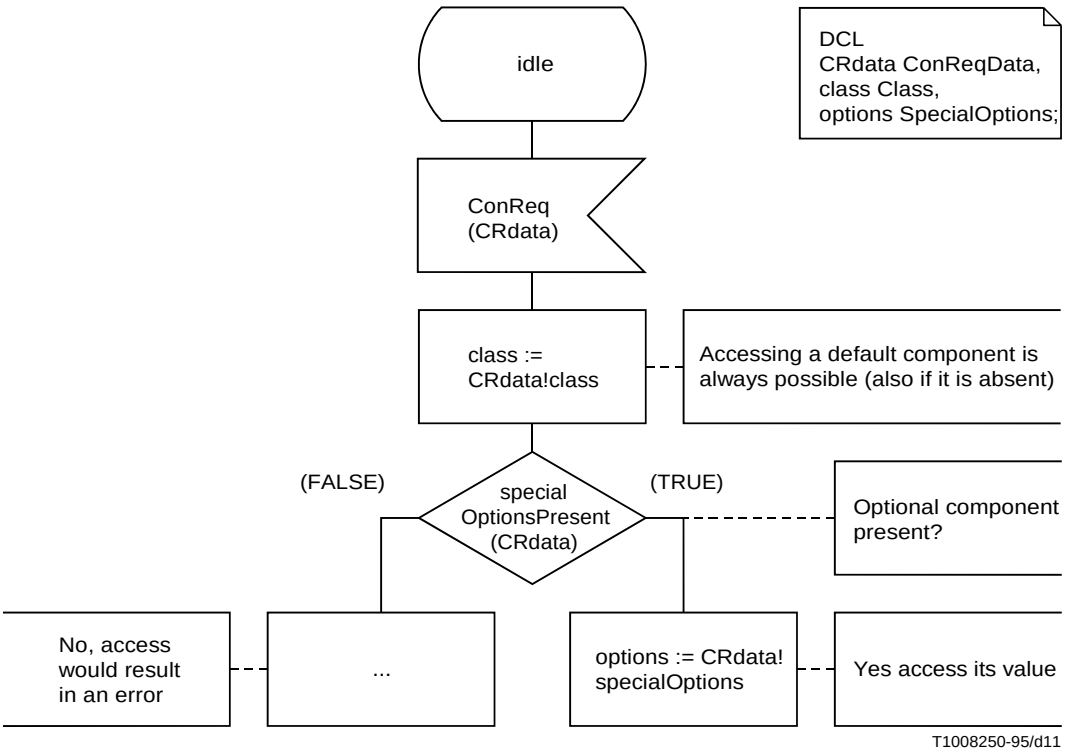

FIGURE II.3.2.2/Z.105

Accès à des composants par défaut et facultatifs

II.3.3 SEQUENCE OF

La Figure II.3.3 montre un exemple d'opérateurs sur des types SEQUENCE OF. La variable 'list' contient une séquence ordonnée d'entiers. Dès réception d'un signal 'nr', un nouvel entier est inséré dans la liste.

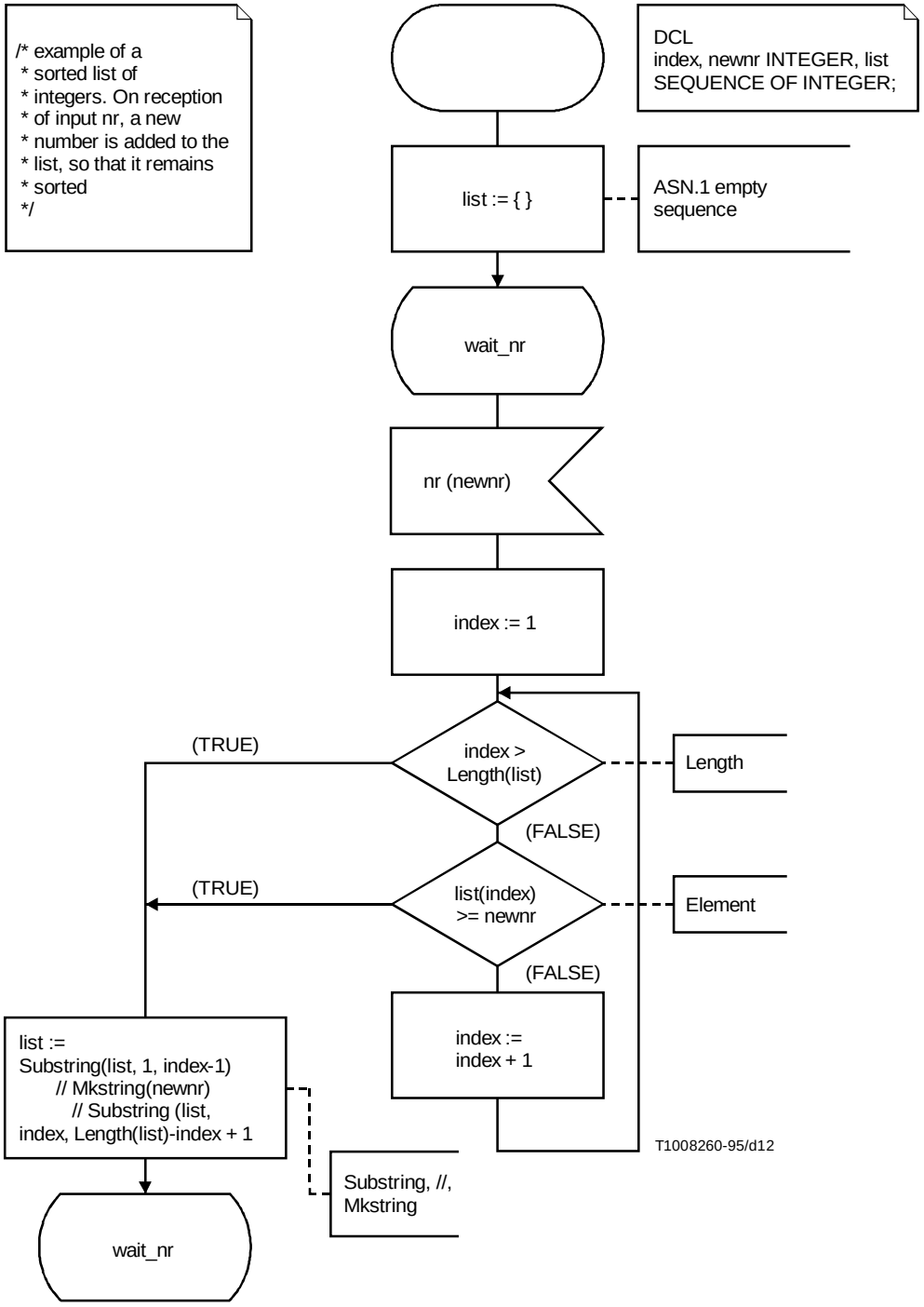

FIGURE II.3.3/Z.105

Utilisation des opérateurs prédéfinis pour SEQUENCE OF

Dans cet exemple, on utilise plusieurs opérateurs pour le type SEQUENCE OF:

- l'opérateur list(index) indexe un élément d'une séquence SEQUENCE OF;
- l'opérateur Length(list) renvoie la longueur d'une chaîne;
- l'opérateur Mkstring(newnr) construit une chaîne contenant un seul item;
- l'opérateur Substring(list, 1, index-1) renvoie une sous-chaîne de liste à partir de la position 1 (première position) et avec la longueur index-1;
- l'opérateur ... // ... renvoie deux chaînes concaténées.

II.3.4 SET OF

Le type ASN.1 SET OF peut servir à représenter des ensembles. Cet exemple montre l'emploi du type SET OF pour modéliser le traitement d'un appel téléphonique. On suppose qu'un même abonné peut activer plusieurs services dits complémentaires.

Pour ces services complémentaires, les types de données sont définis avec les types ASN.1 qui sont énumérés ci-dessous. Le type SupplementaryService est un énuméré dont les valeurs sont tous les services offerts. Le type ServiceSet est celui dont les valeurs sont des ensembles de services complémentaires.

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

ServiceSet ::= SET OF SupplementaryService;
```

La Figure II.3.4 montre un fragment de processus SDL qui assure le traitement d'un appel téléphonique pour un seul abonné. La variable ActiveServ du type ServiceSet est utilisée pour garder trace des services qui sont activés pour un abonné déterminé. L'envoi du signal Activate permet d'activer un service et l'envoi du signal Deactivate permet de désactiver un service.

Si l'abonné décroche, le processus vérifie si un service d'appel direct sur ligne réservée est actif. Si tel est le cas, un numéro prédéfini est immédiatement appelé, sans attendre que l'abonné compose un numéro. Si le service d'appel direct sur ligne réservée n'est pas actif, une tonalité d'invitation à numéroter est envoyée à l'abonné, qui peut alors composer un numéro.

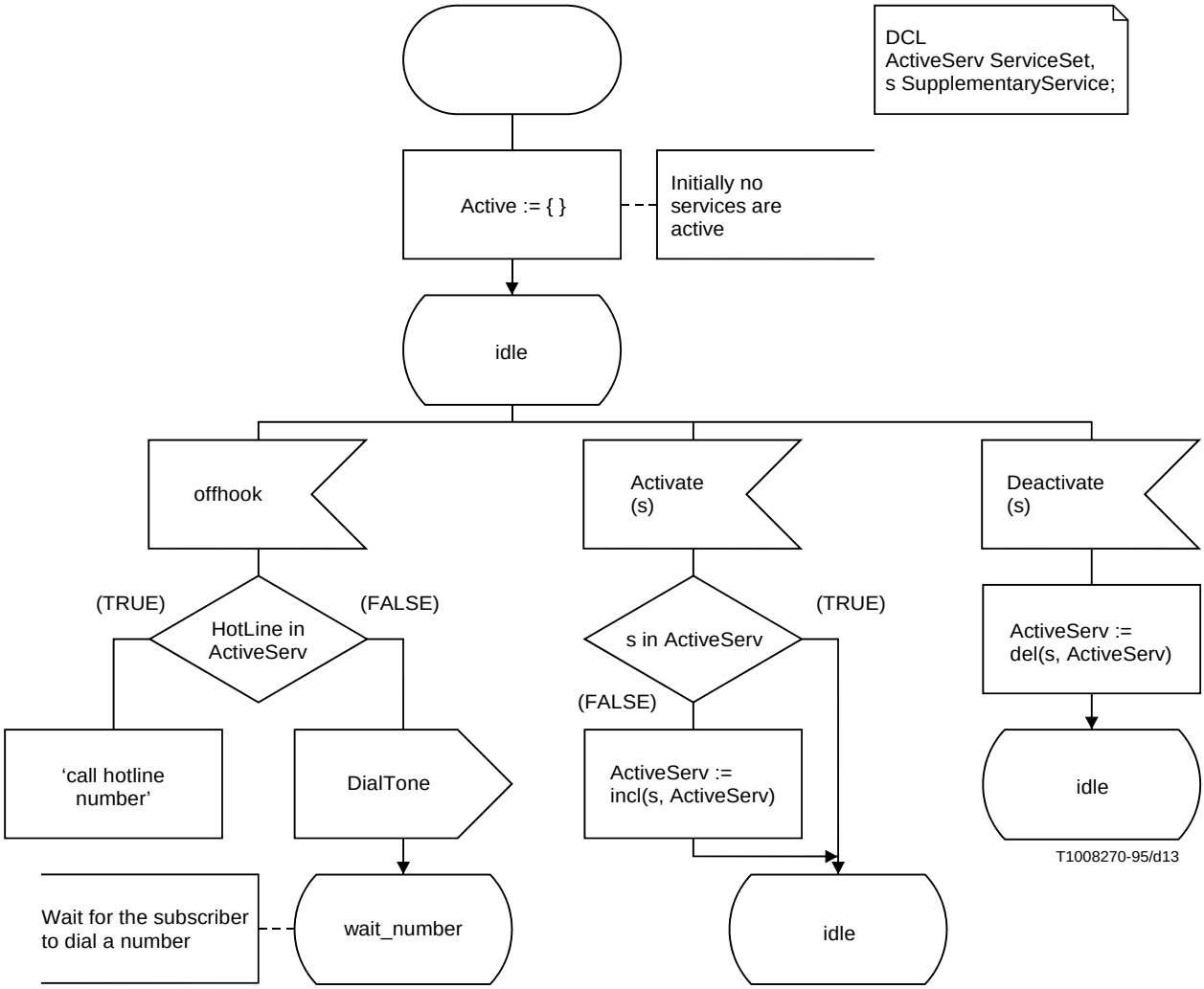

FIGURE II.3.4/Z.105

Utilisation d'opérateurs prédéfinis pour le type SET OF

Dans cet exemple, plusieurs opérateurs agissant sur SET OF sont mis à contribution:

- l'opérateur s dans la variable ActiveServ: renvoie la valeur TRUE si le service s fait partie de l'ensemble ActiveServ;
- l'opérateur incl (s, ActiveServ): indique l'ensemble ActiveServ + l'élément s. On notera qu'en notation ASN.1, le nombre d'apparitions d'un élément dans un type SET OF est significatif: { HotLine, HotLine } n'équivaut pas à { HotLine }. Dans l'exemple, on ne voit pourquoi un service serait activé plus d'une fois pour le même abonné. Le processus vérifie donc si le service s fait déjà partie de la variable Active avant que ce service soit ajouté réellement à l'ensemble;
- l'opérateur del (s, ActiveServ): indique l'ensemble ActiveServ duquel l'élément de service s est retiré.

II.3.5 CHOICE

On peut affecter une valeur à une variable d'un type CHOICE au moyen de la notation de valeur ASN.1 pour CHOICE. On notera que l'identificateur doit être fourni et que deux points verticaux doivent être présents entre cet identificateur et la valeur. On peut également affecter une valeur en utilisant un opérateur SDL (<variable>!<identifier>). Ces deux méthodes sont illustrées dans la Figure II.3.5.

L'accès à un composant non sélectionné d'un type CHOICE provoque une erreur dynamique. On peut éviter de telles erreurs en utilisant l'opérateur !present afin de déterminer quelle variante de composant a été sélectionnée avant d'accéder à ce composant. C'est ce que montre la Figure II.3.5.

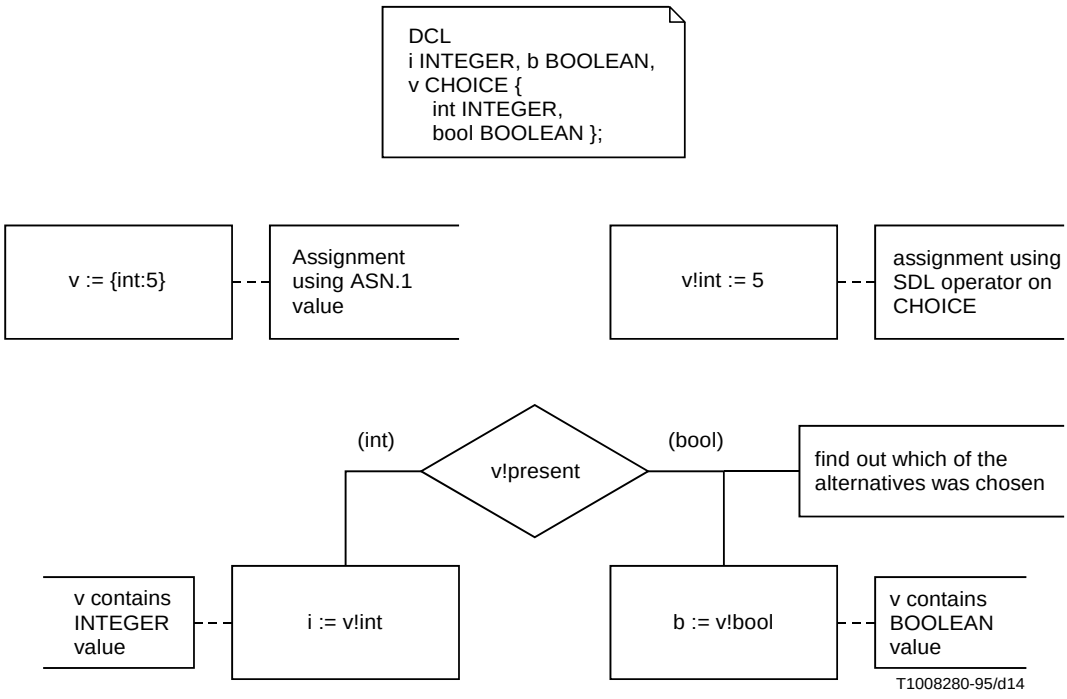

FIGURE II.3.5/Z.105

Utilisation du type CHOICE en combinaison avec le SDL

II.4 Directives en vue d'éviter les restrictions imposées à l'ASN.1

Ce paragraphe donne quelques directives sur la façon de traiter certaines constructions ASN.1 dont la combinaison avec le langage SDL n'est pas supportée. Il n'a pas pour objet de présenter toutes les solutions possibles.

II.4.1 Type ANY DEFINED BY / classe d'objet informationnel TYPE-IDENTIFIER

Le type ANY DEFINED BY (selon la Recommandation X.208, remplacé dans la Recommandation X.681 par la classe d'objet informationnel utile TYPE-IDENTIFIER) est souvent utilisé pour spécifier que le type d'un paramètre dépend de la valeur d'un autre paramètre. Supposons par exemple qu'un processus possède un certain nombre d'attributs, éventuellement de types différents. Un attribut est désigné par son identificateur. Il est possible d'avoir une opération générale pour lire un attribut. Le type d'attribut renvoyé par l'opération de lecture dépendra de l'attribut qui aura été lu.

```
ReadArg ::= OBJECT IDENTIFIER;

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

Dans la présente Recommandation, aucune opération n'est supportée pour le type ANY DEFINED BY, sauf l'égalité = et l'inégalité /=. Cela empêche de formuler en SDL une spécification de l'opération de lecture.

La spécification de l'opération de lecture est possible si les différents types d'attribut sont connus. Dans ce cas, le type ANY DEFINED BY peut être remplacé par un type CHOICE offrant un choix parmi les différents attributs. Si l'on suppose qu'il y a au total deux attributs, l'un du type INTEGER, l'autre du type BOOLEAN, le type ReadResult pourra être réécrit comme suit:

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

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

Au moyen du type ReadResult ainsi redéfini, l'opération de lecture peut être spécifiée en langage SDL comme indiqué dans la Figure II.4.1.

II.4.2 Classe d'objet informationnel OPERATION / macro OPERATION

Ni le mécanisme de macro ASN.1 ni le mécanisme de spécification d'un objet informationnel (voir la Recommandation X.681) ne peut être utilisé en combinaison avec le langage SDL. C'est-à-dire que l'objet informationnel OPERATION, qui est souvent utilisé, ne peut pas l'être en combinaison avec le langage SDL.

Dans ce paragraphe, deux méthodes sont exposées afin de modéliser la réalisation d'une opération en langage SDL:

1) par échange de signaux SDL;
2) par procédure distante SDL.

Soit par exemple une opération de lecture. Sa définition en notation ASN.1 est la suivante:

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

(les variables ReadArg et ReadResult sont définies au II.4.1).

II.4.2.1 Modélisation d'une opération par échange de signaux

L'opération de lecture peut être modélisée par introduction de trois signaux différents:

1) Le signal ReadInvoke, avec un paramètre de type ReadArg afin de modéliser l'invocation de l'opération de lecture.
2) Le signal ReadResult, avec un paramètre de type ReadResult afin de modéliser le renvoi du résultat de l'opération de lecture.
3) Le signal ReadError, avec un paramètre de type «Enuméré», énumérant les différentes erreurs de lecture, afin de modéliser l'échec de l'opération de lecture.

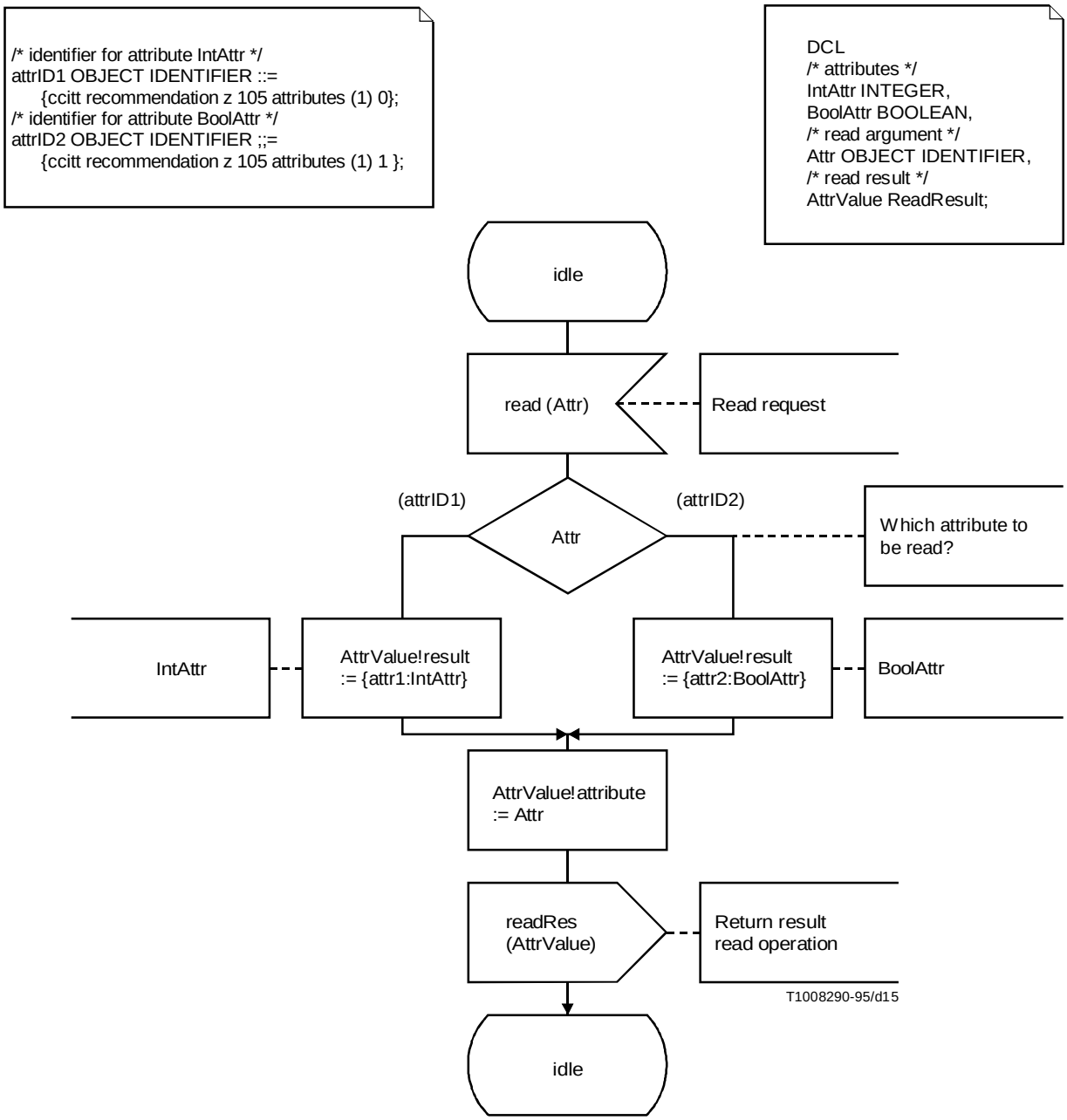

FIGURE II.4.1/Z.105

Spécification de l'opération de lecture

Pour le signal ReadError, un paramètre de type énuméré doit être défini afin d'énumérer les différentes erreurs de lecture:

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

Le code d'opération n'a pas besoin d'être présent dans le modèle SDL: le nom du signal est utilisé à sa place pour identifier l'opération.

L'opération proprement dite peut être modélisée sous la forme d'une transition de processus, comme indiqué sur la Figure II.4.2.1. Ce diagramme contient du texte informel. Le paragraphe II.4.1 montre comment le texte informel peut être remplacé par du langage SDL formel.

L'opération est invoquée par envoi du signal ReadInvoke au processus qui est en mesure d'effectuer l'opération puis le résultat est attendu, comme illustré sur la Figure II.4.2.2.

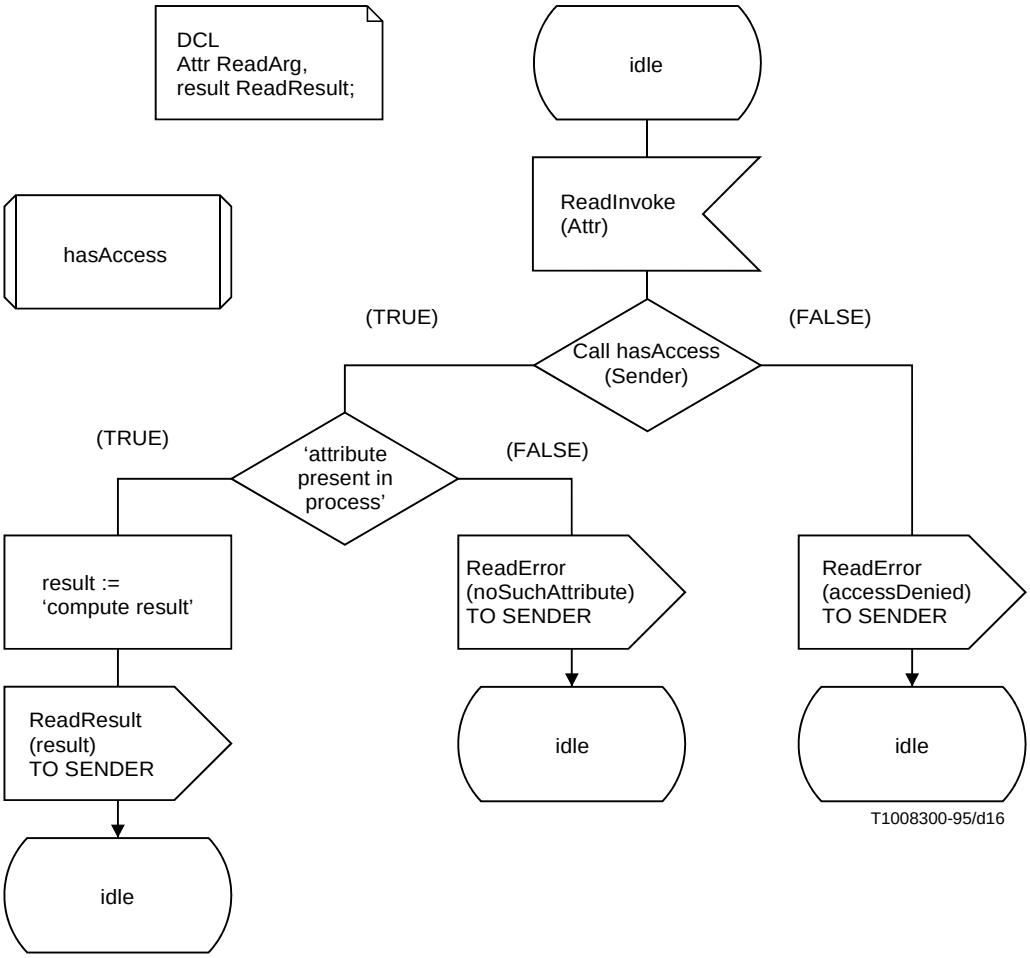

FIGURE II.4.2.1/Z.105

Transition de processus SDL modélisant une opération de lecture

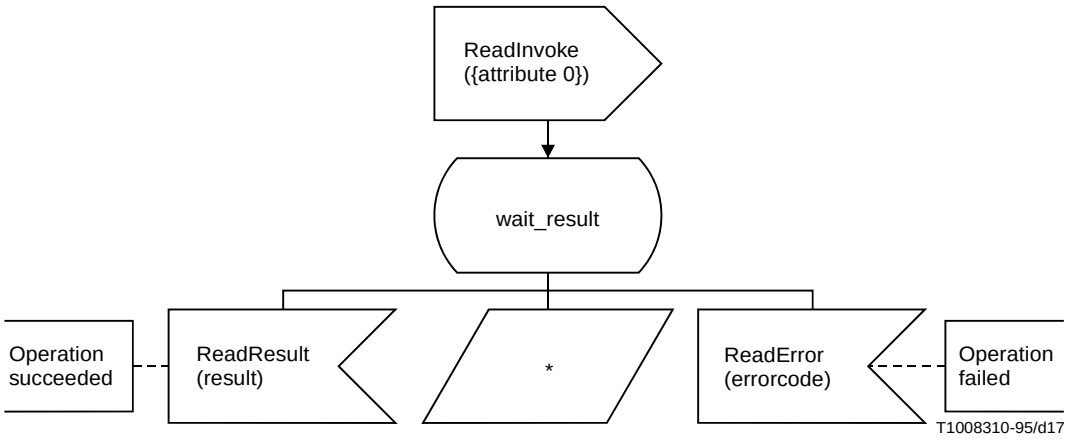

FIGURE II.4.2.2/Z.105

Invocation de l'opération de lecture

II.4.2.2 Modélisation d'une opération par procédure distante

Une autre méthode de modélisation d'une opération de lecture consiste à introduire une procédure distante en langage SDL. Cette procédure possède deux paramètres:

- un paramètre **in** pour l'argument de l'opération de lecture (paramètre de type ReadArg);
- un paramètre **in/out** pour le résultat de l'opération, qui peut être que l'opération a réussi (auquel cas le paramètre est de type ReadResult) ou que l'opération a échoué (auquel cas le paramètre est de type ReadError comme défini au II.4.2.1).

Pour le paramètre **in/out**, un nouveau type est introduit comme suit:

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

La procédure qui modélise l'opération de lecture est représentée sur la Figure II.4.2.3.

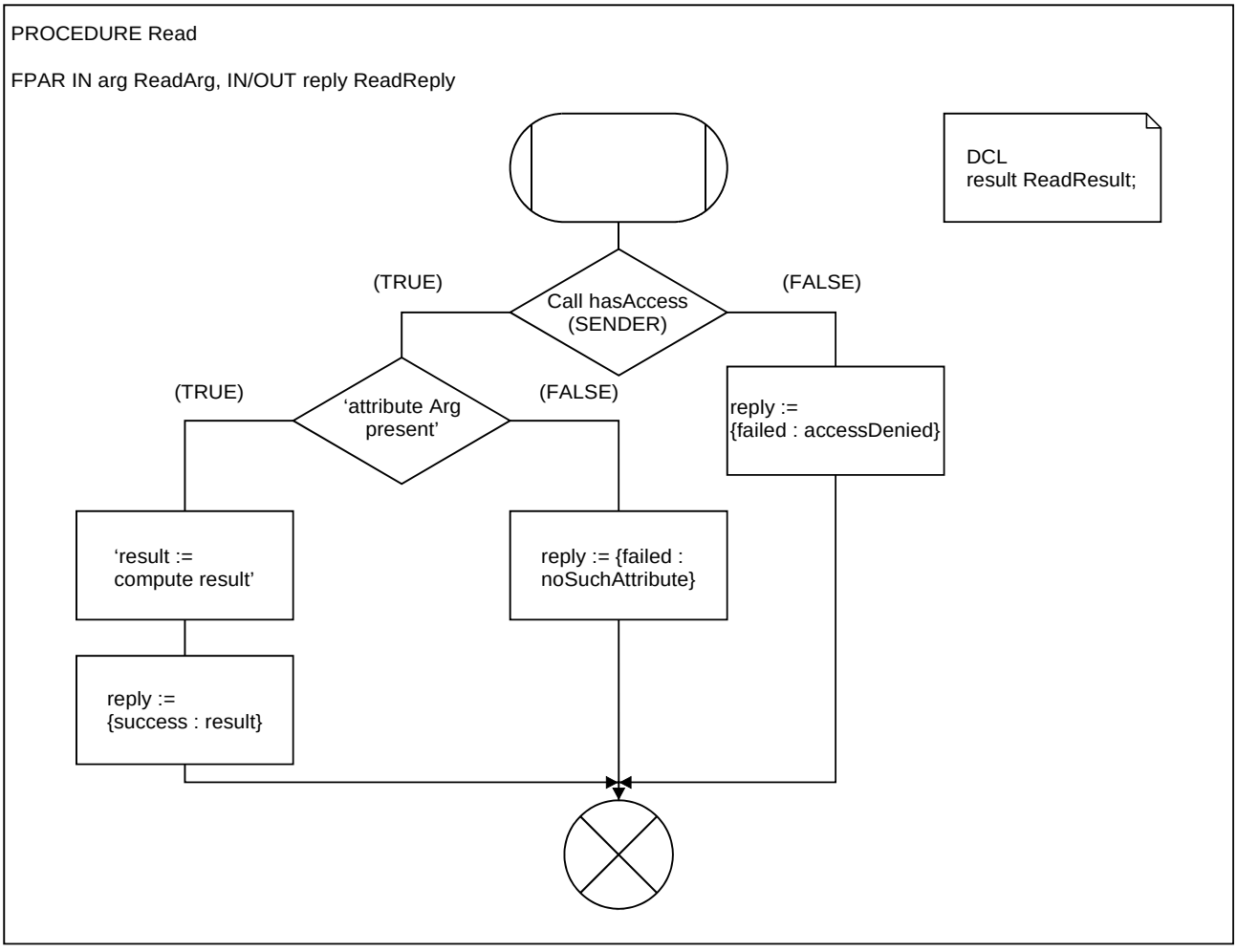

FIGURE II.4.2.3/Z.105

Procédure distante modélisant l'opération de lecture

On invoque l'opération par appel de la procédure distante Read, comme indiqué sur la Figure II.4.2.4. La procédure distante doit être importée par le processus d'appel, ce qui est également représenté sur la figure.

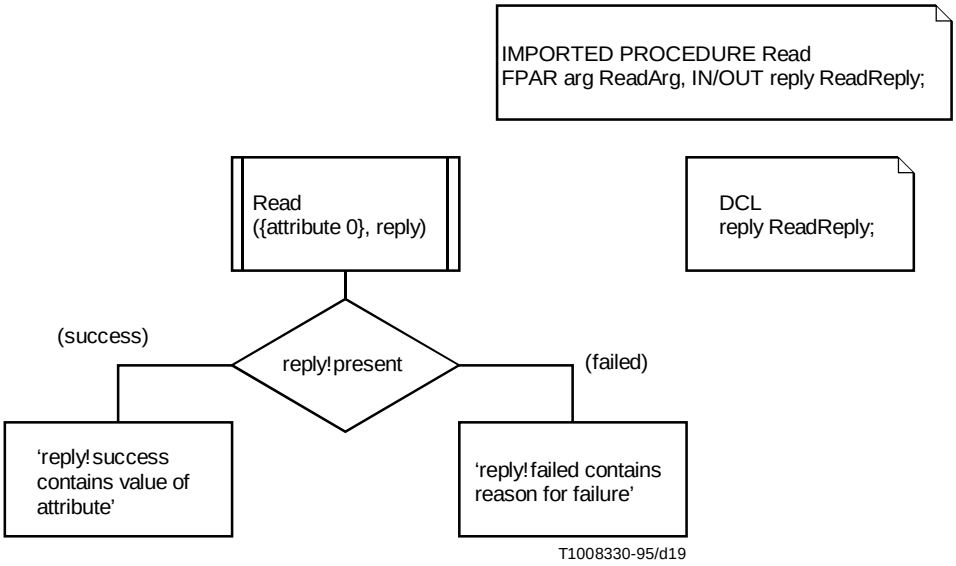

FIGURE II.4.2.4/Z.105

Invocation de l'opération de lecture

II.5 Directives sur l'utilisation combinée du langage SDL et de la notation ASN.1

Le langage qui est défini dans la présente Recommandation est un mélange de langage SDL et de notation ASN.1. Etant donné que les constructions ASN.1 sont appliquées sur les constructions SDL correspondantes, il est en principe possible de mélanger complètement ASN.1 et SDL. On peut par exemple définir des diagrammes de processus dans un module ASN.1 et des données ASN.1 dans un progiciel SDL. Ce procédé n'est toutefois pas considéré comme étant une bonne application de la présente Recommandation.

Ce paragraphe donne quelques directives sur la façon d'utiliser la combinaison du langage SDL et de la notation ASN.1, telle que prévue par la présente Recommandation. Le principe général est le suivant:

Utiliser la syntaxe ASN.1 pour/dans des constructions ASN.1 et la syntaxe SDL pour/dans des constructions SDL

Les directives suivantes suivent ce principe:

- Dans les modules ASN.1, n'utiliser que des notations ASN.1 pour les types et les valeurs. Il y a lieu de définir les constructions SDL sous forme de progiciels. De même, il est préférable que les progiciels SDL ne contiennent pas de définitions de type ASN.1.

  Par exemple, le module ASN.1 WrongUse défini ci-dessous contient des constructions SDL. Une telle formulation, bien que correcte selon la présente Recommandation, est à éviter.

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

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

- Utiliser autant que possible la notation de valeur ASN.1 pour les types définis au moyen de la notation de type ASN.1, et utiliser la notation SDL pour les types définis en notation SDL.

  Exception: pour les opérateurs agissant sur des types ASN.1, il faut utiliser la syntaxe SDL parce qu'il n'y a pas d'opérateurs en notation ASN.1.

  Par exemple:

  – pour S ::= SEQUENCE {a INTEGER}: utiliser { a 5 } comme notation de valeur et non pas (. 5 .)

  – pour **newtype** s **struct** a: integer **endnewtype**: utiliser (. 5 .) comme notation de valeur et non { a 5 }

  – pour une chaîne en mode IA5: utiliser "abc" comme notation de valeur et non 'abc'

  – pour une chaîne de caractères: utiliser 'abc' comme notation de valeur et non "abc"

  – pour une chaîne binaire BIT STRING: il faut écrire '10'B // '0A3C'H parce que le symbole de concaténation // est un opérateur qui n'est pas disponible en notation ASN.1

- Il est recommandé de respecter la sensibilité de la notation ASN.1 à l'inversion de hauteur de casse, même si la présente Recommandation ne supporte pas cette distinction.

  Par exemple:

  – ne pas écrire c ::= Choice { A Integer, B Boolean },

    mais écrire C ::= CHOICE { a INTEGER, b BOOLEAN }

- Eviter autant que possible d'utiliser des types SDL spécifiques à l'intérieur de la définition de types ASN.1 et d'utiliser des types ASN.1 spécifiques à l'intérieur de la définition de types SDL. Essayer d'éviter des types qui mélangent des types SDL spécifiques avec des types ASN.1 spécifiques. Exemples de types SDL spécifiques: PId, Time, Duration, Character. Exemples de types ASN.1 spécifiques: NULL, ANY, BIT STRING.

  Par exemple:

  – ne pas écrire S ::= SEQUENCE { p PId, t Time },

    mais plutôt **newtype** S **struct** p: PId; t: Time **endnewtype**

  – ne pas écrire newtype S struct a: ANY; n: NULL endnewtype,

    mais plutôt S ::= SEQUENCE { a ANY, n NULL }

  – essayer d'éviter la forme S ::= SEQUENCE { n Null, p PId }, bien qu'on ne puisse pas toujours faire autrement.

Appendice III

Opérateurs supportés pour les types ASN.1

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

Le présent appendice donne une vue d'ensemble des types ASN.1 supportés et des opérateurs qui sont disponibles en langage SDL pour ces types.

Au moins deux opérateurs existent pour chaque type: = (égalité) et /= (inégalité).

Any

Opérateurs:

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

Bit string

Pour le type BIT STRING, un nouveau type BIT est introduit, avec les valeurs 0 et 1, ainsi que tous les opérateurs de type BOOLEAN.

Certains opérateurs sont illustrés selon le type suivant:

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

Opérateurs:

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

Length(...),

/* cet opérateur renvoie la longueur, c'est-à-dire Length('100'B) = 3 */

Mkstring(...),

/* conversion du type BIT au type BIT STRING, par exemple Mkstring(1) = '1'B */

...//...,

/* cet opérateur effectue la concaténation de deux types BIT STRING, par exemple '1'B // '00'B = '100'B */

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

/* cet opérateur renvoie une sous-chaîne de type BIT STRING à partir d'une position de départ indiquée et sur une longueur indiquée, par exemple SubString('011'B, 1,2) = '01'B */

..(...),

/* cet opérateur indexe un seul bit à partir de 0, c'est-à-dire que si la variable v = '10'B, on aura v(0) = 1, v(1) = 0. Les bits nommés peuvent également être utilisés pour l'indexation, c'est-à-dire que l'opérateur v(bit0) indexe le bit 0 pour la variable v du type Bitstr défini ci-dessus */

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

/* opérateurs not, and, or, xor, => bit par bit */

Boolean

Opérateurs:

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

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

/* pour des opérations logiques. => est l'opérateur d'implication */

Character string types

Les chaînes de caractères sont des chaînes du type SDL Char.

Opérateurs:

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

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

/* ces opérateurs sont sembables à ceux du type BIT STRING, mais les indices commencent à 1, c'est-à-dire que pour une variable v d'une chaîne de type IA5String de valeur "abc", v(1) = 'a', v(2) = 'b' */

Num (...)

/* cet opérateur renvoie le numéro canonique d'un caractère */

Choice

Les exemples sont fondés sur une expression de type:

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

Opérateurs:

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

...!<identifier>,

/* sélection de champ, par exemple c!field1 = 5, c!field2 produit une erreur dynamique */

<identifier>Present,

/* indique si l'identificateur est présent,
par exemple field1_present (c) = TRUE, field2Present (c) = FALSE */

...!present ,

/* indique quel identificateur a été choisi dans une valeur en donnant
son identificateur, par exemple c!present = field1 */

... < ...

/* cet opérateur indique le type de sélection ASN.1,
par exemple field1 < C indique le type INTEGER */

Enumerated

Les exemples sont de la forme suivante:

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

Opérateurs:

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

/* opérateurs de comparaison (sur la base des numéros indiqués),
par exemple element1 < element2 */

num (...),

/* cet opérateur indique la valeur d'entier associée à une valeur d'énuméré,
num (element1) = 1 */

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

/* ces opérateurs indiquent le prédécesseur et le successeur d'une valeur d'énuméré (pred pour le premier élément et succ pour le dernier élément de résultat erroné),
par exemple succ(element1) = element2, pred(element1) produit une erreur */

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

/* l'opérateur first indique le plus petit élément, c'est-à-dire first (element1) = first (element2) = first (element3) = element1; l'opérateur last indique le plus grand élément, c'est-à-dire last (element1) = last (element2) = last (element3) = element3 */

Integer

Opérateurs:

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

/* opérateurs de comparaison */

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

/* addition, soustraction, soustraction unaire, multiplication, division */

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

/* reste et modulo, par exemple 10 mod 7 = 3, 10 rem 7 = 3, –10 mod 7 = 4, –10 rem 7 = –3 */

Null

Opérateurs:

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

Object identifier

Un type spécial Object_element est introduit, dont les valeurs sont tous les entiers positifs. Le type Object_element ne dispose que des opérateurs = et /=.

Opérateurs agissant sur le type OBJECT IDENTIFIER:

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

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

/* voir sous BIT STRING, mais l'indexation commence par 1 */

Octet string

Un nouveau type OCTET est introduit pour le type OCTET STRING, dont les valeurs sont toutes les chaînes d'octets (OCTET STRING) de taille 1 (octet) et dont les opérateurs sont = et /=.

Opérateurs pour OCTET STRING:

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

Length(...),

/* renvoie le nombre d'octets, c'est-à-dire Length('3FC'H) = 2 */

Mkstring(...),

/* conversion du type OCTET au type OCTET STRING */

... // ...,

/* concaténation, par exemple '1'B // 'A6'H = '10A6'H */

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

/* cet opérateur donne une sous-chaîne d'un type OCTET STRING à partir d'un point de départ donné et sur une longueur donnée,
par exemple SubString('FEDCBA'H, 1, 2) = 'FEDC'H */

...(...),

/* cet opérateur indexe un seul octet, à partir de 1,
c'est-à-dire que si la variable v = '3A10'H, on aura v(2) = '10'H */

BIT STRING (...),

/* conversion du type OCTET STRING au type BIT STRING */

OCTET STRING (...)

/* conversion du type BIT STRING au type OCTET STRING */

Real

Opérateurs:

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

... + ..., ... − ..., − ..., ... * ..., ... / ...,

/* comme pour les opérateurs sur le type INTEGER */

Power(..., ...),

/* opérateur exponentiel. Les arguments sont des entiers
par exemple Power(2, −1) = 0,5 */

float(...),

/* conversion du type INTEGER au type REAL, par exemple float(4) = {4, 10, 0} = 4,0 */

fix(...)

/* conversion du type REAL au type INTEGER, par exemple ({19, 10, −1}) = 1 */

Sequence

Les exemples sont de la forme suivante:

```
S ::= SEQUENCE {
    field1 INTEGER,
    field2 INTEGER DEFAULT 7,
    field3 BOOLEAN OPTIONAL}
s S ::= {field1 105}
```

Opérateurs:

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

...!<identifier>,

/* cet opérateur donne la valeur d'un seul composant, par exemple s!field1 = 105, s!field2 = 7, s!field3 produit une erreur dynamique */

<identifier>Present(...)

/* pour des composants facultatifs, cet opérateur indique si l'identificateur est présent ou absent: field3Present(s) = FALSE. Pour les composants par défaut, l'opérateur Present renvoie toujours la valeur TRUE, c'est-à-dire field2Present(s) = TRUE */

Sequence of

Opérateurs:

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

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

/* opérateurs semblables à ceux du type BIT STRING, mais l'indexation commence par 1 au lieu de 0 */

Set

Voir sous SEQUENCE.

Set of

Les exemples sont de type SET OF INTEGER

Opérateurs:

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

/* opérateurs de comparaison. < est un sous-ensemble de, > l'opérateur de surensemble, par exemple {3, 3} /= {3}, {3, 3} > {3}, {1, 2} = {2, 1} */

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

/* réunion et intersection d'ensembles, par exemple {1} or {0, 1} = {1, 0}, {1, 1} and {4, 1} = {1} */

... in ...,

/* est un élément dans l'ensemble, par exemple 7 in {4, 7} = TRUE, 7 in {4, 8} = FALSE */

Makebag (...),

/* transforme un seul élément en un ensemble, par exemple Makebag (1) = {1} */

Incl (..., ...)

/* ajoute un élément à un ensemble, par exemple Incl (5, {2, 3}) = {5, 2, 3} */

Del (..., ...)

/* retire un élément d'un ensemble, par exemple Del (2, {2, 3, 2}) = {3, 2},
Del (1, {2}) = {2} */

Length (...)

/* nombre d'éléments dans l'ensemble, par exemple Length ({1, 1, 2}) = 3 */

Take (...)

/* indique un seul élément d'un ensemble, par exemple Take ({1, 2, 3}) = 1.
Take ({}) produit une erreur dynamique */

Sous-types

Tous les mécanismes de sous-typage sont admis. Un sous-type possède exactement les mêmes opérateurs que son type d'origine.

Types étiquetés

L'étiquetage de types est autorisé mais les étiquettes sont complètement ignorées. Un type étiqueté a exactement les mêmes opérateurs que son type de base.

Types utiles

Comme pour tous les types, les opérateurs = et /= sont définis pour agir sur les types utiles. Il n'existe pas d'opérateurs spéciaux pour ces types: les types utiles sont définis à l'aide d'autres types ASN.1.

Appendice IV

Résumé de la syntaxe

(Cet appendice ne fait pas partie intégrante de la présente Recommandation)

On trouvera ci-dessous la liste des productions syntaxiques qui, dans la présente Recommandation, sont différentes des productions syntaxiques correspondantes de la Recommandation Z.100. Les productions citées mais non définies sont identiques à celles de la Recommandation Z.100. Toutes les règles syntaxiques énumérées sont issues du changement des règles de production syntaxique suivantes de la Recommandation Z.100:

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

IV.1 <lexical unit>

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

IV.2 <special>

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

IV.3 <national>

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

IV.4 <quoted operator>

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

IV.5 <composite special>

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

IV.6 <package>

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

IV.7 <data definition>

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

IV.8 <sort>

<sort> ::= <sort expression>
<sort expression> ::= {<existingsort> | <subrange> | <sort constructor> |
<inheritance rule> | <generator transformations> |
<structure definition>}
<existingsort> ::= [<package name> **.**] {<sort identifier> | <syntype identifier>} |
any [**defined by** <identifier>] |
<selection>
<selection> ::= <name> **<** <sort>
<sort constructor> ::= <tag> <sort expression> |
<sequence> |
<sequenceof> |
<choice> |
<enumerated>|
<integernaming>
<tag> ::= **[** [**universal** | **application** | **private**]
<simple expression> **]** [**implicit** | **explicit**]
<sequence> ::= { **sequence** | **set** } **{** [<elementsort> { **,** <elementsort>}*] **}**
<elementsort> ::= <namedsort> [**optional** | **default** <ground expression>] |
components of <sort>
<namedsort> ::= [<name>] <sort>
<sequenceof> ::= { **sequence** | **set** } [<sizeconstraint> | <asn1 range condition>]
of <sort>
<choice> ::= **choice {** [<namedsort> { **,** <namedsort>}*] **}**
<enumerated> ::= **enumerated {** <named number> { **,** <named number>}* **}**
<named number> ::= <named value> | <name>
<integernaming> ::= <identifier> **{** <named value> { **,** <named value>}* **}**
<named value> ::= <name> **(** <simple expression> **)**
<subrange> ::= <sort> **(** <range condition> **)**

IV.9 <extended properties>

<extended properties> ::= <sort expression>

IV.10 <range condition>

<range condition> ::= <range> { { **,** | **|** } <range>}*
<range> ::= <closed range> |
<open range> |
<contained subrange> |
<sizeconstraint> |
<innercomponent> |
<innercomponents>
<closed range> ::= <lowerendvalue> { **:** | **..** } <upperendvalue>
<lowerendvalue> ::= {<ground expression> | **min** } [**<**]
<upperendvalue> ::= [**<**] {<ground expression> | **max** }
<open range> ::= [= | /= | < | > | < = | > =] <ground expression>
<contained subrange>::= **includes** <sort>
<sizeconstraint> ::= **size (** <range condition> **)**
<innercomponent> ::= { **from** | **with component** } **(** <range condition> **)**
<innercomponents> ::= **with components**
{ [**... ,**] <named constraint> { **,** <named constraint>}* **}**
<named constraint> ::= <name> [<asn1 range condition>]
[**present** | **absent** | **optional**]
<asn1 range condition> ::= **(** <range condition> **)**

IV.11 <extended primary>

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

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

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

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

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

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

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

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

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

<mantissa> ::= <expression>

<base> ::= <simple expression>

<exponent> ::= <expression>

IV.12 <extended literal identifier>

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

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

INDEX

Note du TSB – La version française ne s'applique pas à l'index de la Recommandation Z.105. Veuillez vous référer à la version anglaise.

_____