Reemplazada por una versión más reciente

UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

UIT-T Z.105

SECTOR DE NORMALIZACIÓN
DE LAS TELECOMUNICACIONES
DE LA UIT

(03/95)

LENGUAJES DE PROGRAMACIÓN

LENGUAJE DE ESPECIFICACIÓN Y DESCRIPCIÓN COMBINADO CON NOTACIÓN DE SINTAXIS ABSTRACTA UNO

Recomendación UIT-T Z.105
Reemplazada por una versión más reciente

(Anteriormente «Recomendación del CCITT»)

PREFACIO

El UIT-T (Sector de Normalización de las Telecomunicaciones) es un órgano permanente de la Unión Internacional de Telecomunicaciones (UIT). Este órgano estudia los aspectos técnicos, de explotación y tarifarios y publica Recomendaciones sobre los mismos, con miras a la normalización de las telecomunicaciones en el plano mundial.

La Conferencia Mundial de Normalización de las Telecomunicaciones (CMNT), que se celebra cada cuatro años, establece los temas que han de estudiar las Comisiones de Estudio del UIT-T, que a su vez producen Recomendaciones sobre dichos temas.

La aprobación de Recomendaciones por los Miembros del UIT-T es el objeto del procedimiento establecido en la Resolución N.° 1 de la CMNT (Helsinki, 1 al 12 de marzo de 1993).

La Recomendación UIT-T Z.105 ha sido preparada por la Comisión de Estudio 10 (1993-1996) del UIT-T, y fue aprobada por el procedimiento de la Resolución N.° 1 de la CMNT el 6 de marzo de 1995.

NOTA

En esta Recomendación, la expresión «Administración» se utiliza para designar, en forma abreviada, tanto una administración de telecomunicaciones como una empresa de explotación reconocida de telecomunicaciones.

Reemplazada por una versión más reciente

ÍNDICE

RESUMEN

Objetivo

La presente Recomendación define cómo se puede utilizar la ASN.1 en combinación con el SDL. El propósito es que la estructura y el comportamiento de los sistemas se describan con el SDL, mientras que los parámetros de los mensajes intercambiados y los datos utilizados internamente se describan con la ASN.1. Esta Recomendación es una ampliación de la Recomendación Z.100. Para los usuarios que están familiarizados con el SDL y la ASN.1 las ampliaciones son directas, en el sentido de que se puede utilizar la ASN.1 en todos los lugares donde se puede utilizar datos en SDL.

Alcance

Esta Recomendación presenta una definición de sintaxis y semántica de la combinación de SDL y ASN.1. En los apéndices se proporcionan varias descripciones generales del lenguaje combinado, así como ejemplos y directrices para la utilización de esta Recomendación.

Aplicación

La esfera fundamental de aplicación de la presente Recomendación es la especificación de sistemas de telecomunicaciones. La utilización combinada de SDL y ASN.1 permite especificar de manera coherente la estructura y el comportamiento de los sistemas de telecomunicaciones, junto con los datos, mensajes y codificación de mensajes que estos sistemas utilizan.

Estado/estabilidad

Esta Recomendación es el manual de referencia completo que describe la combinación de SDL y ASN.1.

Trabajo conexo

Recomendación Z.100: Lenguaje de especificación y descripción.

Recomendación X.680: Notación de sintaxis abstracta uno.

Recomendación Z.105

LENGUAJE DE ESPECIFICACIÓN Y DESCRIPCIÓN COMBINADO CON NOTACIÓN DE SINTAXIS ABSTRACTA UNO

(Ginebra, 1995)

1 Introducción

La presente Recomendación define la combinación de la notación de sintaxis abstracta uno (ASN.1, *Abstracts Syntax Notation One*) y el lenguaje de especificación y descripción (SDL, *specification and description language*). Esta combinación permite utilizar la ASN.1 en diagramas SDL, y la importación de módulos ASN.1 en descripciones SDL.

El SDL es un lenguaje para la especificación y descripción de sistemas de telecomunicaciones, que tiene conceptos para:

- estructurar sistemas;
- definir el comportamiento de sistemas;
- definir los datos utilizados por los sistemas.

La ASN.1 es un lenguaje para la definición de datos. Guardan relación con la ASN.1 las reglas de codificación, que definen cómo se transfieren valores ASN.1 como trenes de bits durante la comunicación.

1.1 Objetivo

La combinación de SDL y ASN.1 permite especificar de manera coherente la estructura y el comportamiento de sistemas de telecomunicación, junto con los datos, mensajes y codificación de mensajes que estos sistemas utilizan, a saber: estructura y comportamiento que utilizan el SDL, datos y mensajes que utilizan la ASN.1, codificación por referencia a las reglas de codificación pertinentes definidas para la ASN.1.

Esta Recomendación sustenta la plena utilización de tipos de datos SDL.

1.2 Diferencias entre los datos SDL y la ASN.1

El SDL y la ASN.1 proporcionan facilidades para la definición de datos, pero se basan en conceptos diferentes.

En el SDL, un tipo de datos está caracterizado por los conjuntos de valores (denominados géneros), y los operadores que se definen para los géneros. Se dispone de facilidades para definir géneros, valores y operadores en valores. El SDL tiene géneros predefinidos y proporciona medios para definir nuevos géneros, valores y operadores, basados en definiciones existentes o en definiciones con propiedades completamente nuevas. Las propiedades de los datos se basan en la *equivalencia algebraica de términos*.

La ASN.1 se basa en *conjuntos*: un tipo de datos ASN.1 define un conjunto de valores. Se proporciona una notación para escribir determinados valores de un tipo de datos. No se proporcionan operadores en tipos de datos (por ejemplo, en las ASN.1 no existe un «+» para sumar dos valores enteros). Se puede definir nuevos tipos de datos, basados en tipos existentes. Mediante la elección de reglas de codificación adecuadas (por ejemplo, las reglas de codificación básicas definidas en la Recomendación X.690), se puede especificar cómo se codifican los valores en trenes de bits para transmitirlos a otro dispositivo.

1.3 Características de la combinación de SDL y ASN.1

La presente Recomendación es una extensión pura de la Recomendación Z.100, salvo para algunas pequeñas restricciones en las reglas léxicas (véase el Apéndice II.2). Básicamente, esta Recomendación define una ampliación de la cláusula 5/Z.100 sobre datos.

Los sistemas descritos en SDL combinado con ASN.1 tienen las siguientes características:

- la estructura y el comportamiento se definen utilizando conceptos SDL;
- los parámetros de las señales se definen mediante tipos ASN.1 o géneros SDL;
- los datos se pueden definir con definiciones de tipo ASN.1 o definiciones de género SDL;
- la codificación de valores de datos ASN.1 se puede definir mediante una referencia a las reglas de codificación pertinentes. La codificación está fuera del ámbito de esta Recomendación.

El Cuadro 1 resume las características de los datos SDL, ASN.1, y su combinación:

CUADRO 1/Z.105

	SDL	ASN.1	SDL/ASN.1
Definición de tipos	X	X	X
Notación para valores	X	X	X
Definición de operadores	X		X
Expresiones	X		X
Codificación de valores		X	X

1.4 Restricciones del SDL y la ASN.1

La ASN.1, según se define en la Recomendación X.680 se utiliza en combinación con el SDL, con unas pocas restricciones. El Apéndice I.1 contiene un resumen del subconjunto de ASN.1 sustentado. No se sustentan las características definidas en las Recomendaciones X.681, X.682 y X.683.

No se sustentan las características de la Recomendación X.208 que han sido sustituidas en la Recomendación X.680, con excepción del tipo ANY.

Se sustenta la utilización de SDL, según se define en la Recomendación Z.100 con algunas restricciones relacionadas con las reglas léxicas.

El Apéndice I.2 resume las restricciones impuestas al SDL cuando se utiliza en combinación con la ASN.1.

1.5 Estructura de esta Recomendación

La presente Recomendación no es independiente: el lenguaje definido en esta Recomendación se basa en la Recomendación Z.100. Se aplica el lenguaje definido en la Recomendación Z.100, con la excepción de 13 reglas de producciones de sintaxis de la Recomendación Z.100 que se redefinen en la presente Recomendación, que está estructurada de la siguiente manera:

La cláusula 2 define los cambios de las reglas léxicas de la Recomendación Z.100.

La cláusula 3 define los cambios de la Recomendación Z.100 para incorporar módulos ASN.1.

La cláusula 4 define los cambios del capítulo sobre datos de la Recomendación Z.100 para incorporar tipos y valores de datos ASN.1.

El Anexo A define el lote con datos predefinidos.

El Apéndice I resume las restricciones en la utilización del SDL y la ASN.1.

El Apéndice II contiene ejemplos sobre la utilización del SDL en combinación con la ASN.1, junto con algunas directrices.

El Apéndice III contiene una descripción general de los constructivos admitidos de ASN.1, y de los operadores disponibles para dichos constructivos.

El Apéndice IV resume los cambios sintácticos de las reglas de producción de la Recomendación Z.100.

1.6 Convenios utilizados en la presente Recomendación

Se aplican básicamente los convenios de la Recomendación Z.100, es decir, las palabras clave aparecen en negritas, los nombres predefinidos comienzan con mayúscula, etc. Sin embargo, en algunos ejemplos se utilizan convenios de ASN.1 para facilitar la lectura a los usuarios de ASN.1, es decir, las palabras clave aparecen con mayúsculas (no en negritas).

En el índice se enumeran únicamente las ocurrencias definidoras. Los no terminales a los que se hace referencia en el texto, pero que no figuran en el índice, se definen en la Recomendación Z.100.

2 Adiciones a la Recomendación Z.100

2.1 Unidades léxicas (Lexical units)

La producción <lexical unit> se modifica como sigue:

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

NOTA – Hay dos cambios de <lexical unit> como se define en la Recomendación Z.100:

1) <character string> se sustituye por <string> para que las comillas sirvan como delimitadores de cadena.

2) Se añade <single line note> para sustentar la utilización de dos guiones como comienzo de un comentario.

2.2 Palabras clave (Keywords)

Se añaden las siguientes palabras clave como alternativa a la producción <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**			

NOTAS

1 En la presente Recomendación no se hace distinción entre letras mayúsculas y minúsculas en palabras clave y nombres. De conformidad con los convenios de la Recomendación Z.100, las palabras clave se muestran en minúsculas y en negritas.

2 No todas las palabras clave de la Recomendación X.680 son palabras clave en la presente Recomendación. Por ejemplo, BOOLEAN (que en esta Recomendación se escribe Boolean) es una palabra clave en la Recomendación X.680, pero es un nombre predefinido en la presente Recomendación.

2.3 Nombres (Names)

Las producciones <national> y <special> se sustituyen por:

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

En un <name>, un <full stop> no debe ir seguido inmediatamente de un <underline> ni de otro <full stop>.

Un <underline> no debe ir seguido inmediatamente de un <full stop>.

NOTAS

1 Las condiciones anteriores significan que a..b y a . b denotan tres <lexical unit>s y no un solo nombre. En la regla sintáctica <existingsort> se utiliza <full stop> como <lexical unit> (es decir, como alternativa en <special>).

2 Las dos producciones anteriores implican que los caracteres llave izquierda, llave derecha, corchete izquierdo, corchete derecho y barra vertical no pueden ser parte de nombres en la presente Recomendación, contrariamente al caso en la Recomendación Z.100.

2.4 Cadenas (Strings)

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

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

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

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

<character string> se define en la Recomendación Z.100.

Para representar un carácter comilla (') dentro de <quoted string>, se utilizan dos comillas.

<bitstring>s y <hexstring>s son los literales de los géneros predefinidos Bit_string y Octet_string (véanse 4.4.3 y el Anexo A).

2.5 Operadores con comillas (Quoted operators)

<quoted-operator> se cambia de una construcción sintáctica a una unidad léxica:

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

donde <infix operator> forma parte de la unidad léxica, aun cuando <infix operator> en otras producciones aparece como un constructivo sintáctico.

Hay ambigüedad léxica entre <quoted string> y <quoted operator>, porque la secuencia de caracteres que identifica a <quoted operator> es también una secuencia de caracteres válida para identificar <quoted string>. En tales casos, la unidad léxica es <quoted operator>.

NOTAS

1 En caso de conflicto, se debe utilizar una <character string> en vez de una <quoted string>. Por ejemplo, una cadena de caracteres formada por un solo asterisco se tiene que escribir '*', pues la construcción "*" es siempre un <quoted operator>.

2 El cambio de <quoted operator> de construcción sintáctica a unidad léxica supone que ya no se pueden poner separadores ni comentarios dentro de un <quoted operator>.

2.6 Notas de una sola línea (Single line notes)

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

Se aplican las mismas reglas para <single line note> y para <note>, salvo que una <single line note> comienza con -- y no con /*, y que una <single line note> es terminada con un corte de línea o la secuencia -- (lo que aparezca primero) y no con */.

NOTA – Las dos formas:

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

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

son idénticas, mientras que la forma:

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

es algo diferente, pues el constructivo de comentario tiene su propia notación en la sintaxis gráfica (el símbolo de comentario).

2.7 Símbolos especiales

La producción <composite special> se extiende como sigue:

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

NOTA – Las tres últimas alternativas son adiciones.

2.8 Utilización de caracteres de espacio en nombres

En 2.2.2/Z.100 se enumeran cinco excepciones a la regla de que los caracteres de espacio y control pueden sustituir a los caracteres de subrayado en los <name>s. A continuación se indican otras excepciones, que sólo se aplican a los nuevos constructivos definidos en esta Recomendación:

- Los <underline>s en el <name> de un <namedvalue> se deben especificar explícitamente.
- Los <underline>s en los <name>s contenidos en <composite primary> se deben especificar explícitamente.

3 Lote (Package)

La producción <package> se extiende como sigue:

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

donde <module definition> es:

<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>}*

En un <objectidentifiervalue> de un <module> sólo son visibles los literales y operadores de los géneros definidos en el lote predefinido Predefined.

No tiene importancia si se especifica <tagdefault> ni cómo se especifica.

Modelo

Si un <objectidentifiervalue> está presente en un <module>, el <module> se transforma en un nuevo <module>, en el cual se omite el <objectidentifiervalue>, y en el que <package name> contiene información sobre el valor que se especificó en <objectidentifiervalue>. En esta Recomendación no se define cómo se produce realmente esta transformación.

Una <module definition> tiene el mismo significado que una <package definition>, donde:

- <module> (sin ningún <objectidentifiervalue>) corresponde a <package name>;
- <imports> corresponde a <package reference clause>s;
- <exports> corresponde a <interface>;

Ejemplo:

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

es lo mismo que:

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

De manera similar, cuando el lote se utiliza en <imports> de otro lote:

IMPORTS yes FROM myway;

es el mismo que <package reference clause>:

use myway/yes;

NOTAS

1 Una <module definition> no termina con un <end>, contrariamente a una <package definition>.

2 Cuando se utilizan lotes/módulos en la <system definition>, se debe emplear la notación de la Recomendación Z.100 (es decir, <package reference clause> en vez de <imports>).

3 Como una orientación: por razones de compatibilidad con la Recomendación X.680, <module definition>s deben contener, en la medida posible, definiciones ASN.1.

4 Definición y utilización de datos

4.1 Definiciones de variables y de datos

La producción <data definition> se extiende como sigue:

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

Las cuatro alternativas <partial type definition>, <syntype definition>, <generator definition>, y <synomym definition> se definen en la Recomendación Z.100, mientras que <sort assignment> y <value assignment> se definen a continuación.

NOTAS

1 En la presente Recomendación, la producción <data definition> permite utilizar la notación de la Recomendación X.680 para definir géneros (es decir, tipos ASN.1) y sinónimos. Las alternativas <sort assignment> y <value assignment> son nuevas.

2 En esta Recomendación <end> es obligatorio, mientras que en la Recomendación X.680 no hay símbolos separadores entre definiciones.

4.1.1 Asignaciones de género

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

Modelo

<sort assignment> es una forma abreviada de una <partial type definition> o una <syntype definition> que tiene el <sort name> como nombre definido.

Si las <extended properties> son un <existingsort> o una <subrange>, la <sort assignment> es igual que una <syntype definition> que contiene únicamente <extended properties>.

Una <sort assignment> es igual que una <partial type definition>, cuando <properties expression> está vacía, y los <formal context parameters> se omiten.

Ejemplo:

La asignación de género:

Integerlist ::= SEQUENCE OF INTEGER;

es igual que:

newtype Integerlist
sequence of Integer
endnewtype Integerlist;

La asignación de género:

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

es igual que:

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

NOTA – Por lo tanto, si se necesitan literales/operadores adicionales, se debe utilizar una <partial type definition> (es decir, la forma completa).

4.1.2 Asignaciones de valor

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

<ground expression> está definida en la Recomendación Z.100.

<value assignment> introduce un nombre para un valor específico.

Modelo

<value assignment> es igual que <synonym definition item>.

Ejemplo:

La definición:

yes BOOLEAN ::= TRUE;

es igual que:

synonym yes Boolean = True;

4.2 Expresiones de género

Las producciones <sort> y <extended properties> se cambian como sigue:

<sort> ::= <sort expression>

<extended properties> ::= <sort expression>

donde <sort expression> es:

<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 se especifica el <identifier> **defined by**, el <identifier> debe denotar un género o sintipo. Si no, <identifier> no es significativo.

Las alternativas <existingsort> y <subrange> no se deben utilizar como <extended properties> de una <partial type definition> o <syntype>.

Una <sort expression> define las propiedades de un género. No es significativa si <tag>s están presentes. No obstante, si están presentes, la <simple expression> contenida debe ser de género Integer.

Un <sort>, en una <operator definition> referenciada (véase la Recomendación Z.100), que por posición corresponde con un <sort> en una <textual operator reference>, debe ser sintácticamente igual que ese <sort>.

Un <sort>, en una <operator signature> (véase la Recomendación Z.100), que por posición corresponde con un <sort> en una <operator definition>, debe ser sintácticamente igual que en la <operator definition>. No obstante, si el <sort> es un <existingsort>, no tiene por qué ser sintácticamente igual, pero debe denotar el mismo género o sintipo.

Existe una ambigüedad sintáctica entre <generator transformations> y <subrange>. Un <sort> que empieza con un <identifier> seguido de un paréntesis de izquierda se trata como un <sort identifier>, si es posible de acuerdo con las reglas de visibilidad (véase 2.2.2/Z.100) y, en caso contrario, como un <generator identifier>.

Modelo

En la siguiente descripción de cómo los anteriores constructivos se transforman en su representación Z.100, el orden es importante:

- La palabra clave **any** es sintaxis derivada para proporcionar el género predefinido Any_type (véase el Anexo A).
- Una <selection> es sintaxis derivada para proporcionar el género del campo dado por el <name>. Este campo debe estar asociado al <sort>.
- Un <existingsort> representa un <sort identifier> o un <syntype identifier>. Si está presente un <module>, representa un <package name> (según se explica en la cláusula 3) que entonces constituye la parte más a la izquierda del calificador del identificador.
- Un <sort> que no es un <existingsort> ni una <subrange> representa un <sort identifier> de un género definido implícitamente. Este género tiene un nombre anónimo y único y tiene la <sort expression> como <extended properties>. El género se define en la unidad de alcance contenedora más próxima.
- Un <sort> que es una <subrange> representa un <syntype identifier> de un sintipo definido implícitamente que contiene la <subrange>. Este sintipo tiene un nombre anónimo y único y se define en la unidad de alcance más próxima que contiene el lugar donde aparece <sort>. Se aplica un Modelo especial para una <subrange> que sea un <parent sort identifier> (véase 4.2.7).
- Una <partial type definition> con <extended properties> que es una <sort constructor> se representa tal como se indica a continuación.

4.2.1 Campos y posibilidad de opción

Por razones de conveniencia, en el *Modelo* para las notaciones de tipo y valor, se define la noción de campos:

Para un género, se asocia ningún nombre o varios nombres de campo. Los nombres de campo de un género son aquellos nombres que, cuando están concatenados con el nombre Extract!, forman nombres de operadores que están definidos localmente para el género y que tienen un argumento que es el género. Cada campo posee asimismo un género de campo que es el género resultado del operador.

Un campo es facultativo si existe además un operador local cuyo nombre es el nombre de campo concatenado con el nombre Present, cuyo (único) argumento es el género que tiene el campo asociado y cuyo género de resultado es el género predefinido Boolean.

NOTA 1 – En el modelo de datos subyacente (ACT ONE), las propiedades de un género sólo se caracterizan por sus literales, operadores y axiomas. Las estructuras de datos compuestas, como <structure definition> de SDL y <sequence>s de ASN.1 son, notaciones derivadas para definir literales, operadores y axiomas. Por esta razón, la noción de campos y campos facultativos se sintetiza a partir de la existencia de operadores específicos en vez de ser propiedades de géneros definidos con un constructivo como <sequence>.

NOTA 2 – Del *Modelo* de <sequence> (véase 4.2.2) se desprende que un campo que tiene un valor por defecto se trata como un campo facultativo que siempre contiene un valor bien definido y cuyo operador Present siempre da True.

NOTA 3 – De ahí que los géneros definidos con las notaciones abreviadas <sequence> y <choice> tienen campos asociados. No obstante, la utilización de estas notaciones abreviadas no es la única forma de asociar campos a géneros.

NOTA 4 – Por ende, los campos especificados con la palabra clave **default** dentro de una <sequence> (véase 4.2.2) también son campos facultativos.

Ejemplo:

Como se indica en 4.2.2, la <secuencia>:

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

es igual que:

```
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
/* Propiedades de los anteriores operadores aquí definidos, véase 4.2.2*/
endnewtype S;
```

NOTA 5– De esta definición **newtype**, se desprende (como se ha previsto) que el género S tiene tres campos, A, B y C, de los cuales B y C son campos facultativos. Se desprende asimismo que el género de campo de A es Integer, el género de campo de B es Real y el género de campo de C es IA5String.

4.2.2 Secuencia

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

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

components of <sort>

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

El <name> en <namedsort> sólo se puede omitir si el <sort> es una <selection>. En este caso, se considera que el <name> es el mismo <name> contenido en la <selection> (esto se aplica a cualquier tipo de utilización de <namedsort> en la presente Recomendación).

No es importante si se especifica la palabra clave **sequence** o la palabra clave **set**.

Una <ground expression> en un <elementsort> debe ser del género denotado por el <sort> contenido en el <namedsort> del <elementsort>.

Los <name>s en los <namedsort>s de la <secuencia> deben ser distintos.

Modelo

Aunque no se muestra en los axiomas derivados en el *Modelo*, todos los identificadores de operador contenidos en los axiomas están totalmente calificados para evitar ambigüedades con los operadores de otros géneros.

Un <elementsort> que es **components of** <sort> es sintaxis derivada para una lista de <elementsort>s ordenados, uno para cada campo asociado al <sort> (véase 4.2.1). Los <elementsort>s que en <sort> son campos facultativos tienen la palabra clave **optional** especificada. Los nombres de campo se ordenan utilizando las mismas reglas que para los nombres literales **nameclass** (véase 5.3.1.14/Z.100).

Una <sequence> es sintaxis derivada para una <properties expression> que contiene:

- Operadores "Extract!" que asocian los <name>s de campo con la <sequence> (véase 4.2.1).
- Los operadores "Present" que hacen campos facultativos de los <name>s en los <namedsort>s que están seguidos de la palabra clave **optional** o **default** (véase 4.2.1).
- Un operador "Make!", para crear valores de secuencia, o el literal Empty si todos los campos son facultativos o no se especifican <elementsort>s. En el caso del literal Empty, cada vez que aparece el operador "Make!" en los axiomas del *Modelo*, se considera en cambio como ese literal.
- Los operadores "Modify!" para modificar campos de valores de secuencia. El nombre del operador utilizado para modificar un campo es el <name> de campo concatenado con "Modify!".

La <argument list> para el operador Make! es la lista de <sort>s que aparecen en el <elementsort>, ordenados de conformidad con los nombres de campo. Los nombres de campo se ordenan utilizando las mismas reglas que para **nameclass** (véase 5.3.1.14/Z.100).

Los <sort>s de campo que van seguidos de la palabra clave **optional** o **default** se excluyen de la lista. El <result sort> para el operador Make! es el género del contexto contenedor. Este es también el <sort> de la secuencia.

La <argument list> de cada operador de modificación de campo es el <sort> de la secuencia, seguido por el <sort> de campo de dicho campo. El <result sort> de un operador de modificación de campo es el <sort> de la secuencia.

La <argument list> de cada operador de extracción de campo es el <sort> de la secuencia. El <result sort> de un operador de extracción de campo es el <sort> de campo de dicho campo.

La <argument list> de cada operador de presentación de campo es el <sort> de la secuencia.

El <result sort> de cada operador de presentación de campo es el género predefinido Boolean.

Para cada campo hay un axioma que describe el significado de la extracción de un campo:

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

donde F es el nombre del campo, y E es el argumento del operador Make! que, por posición, corresponde al campo F, en caso de que F no sea un campo facultativo. Si F es un campo facultativo especificado con **optional**, entonces E es **error!**, y si el campo se especifica con **default**, E es la <ground expression> asociada. Si F es un campo facultativo derivado del constructivo **components of**, entonces E es FExtract!(<<**type** S1>> Make!(a1,b1,...,n1)), siendo S1 <sort> mencionado en el constructivo **components of**.

Para cada campo facultativo hay axiomas que describen el significado de probar la presencia de un campo:

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

donde F y G son nombres de diferentes campos y F es un campo facultativo. B es True si F se especifica con **default**. Si F se deriva del constructivo **components of**, entonces B es FPresent(<<**type** S1>>Make!(a1,b1,...,n1)), donde S1 es el <sort> mencionado en el constructivo **components of**. En otros casos, B es False.

Para cada campo hay axiomas que indican que su valor no está influido por la modificación de otros campos:

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

donde F y G son nombres de campos diferentes.

Hay también un axioma que expresa la igualdad de valores de secuencia:

para todos 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;
```

donde S es el género de secuencia, A1,...,An son los campos no facultativos y B1,...,Bn son los campos facultativos.

Ejemplo:

El género de secuencia:

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

es igual que:

```
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;
```

NOTAS

1 Los campos especificados con **default** tienen un operador "Present", y por lo tanto se tratan como un campo facultativo que produce un valor por defecto en vez de un error si se accede a él antes de haberle asignado un valor.

2 Los campos derivados de **components of** se deben ordenar de forma tal que el orden de los géneros en la <argument list> del operador Make! sea determinístico. En esta Recomendación se utiliza el orden alfabético.

3 No se hace distinción entre la utilización de la palabra clave **sequence** y la palabra clave **set**. Hay cierta flexibilidad en comparación con la Recomendación X.680.

4 En esta Recomendación, los rótulos no tienen que distinguir necesariamente entre componentes del mismo tipo: se supone rotulación automática ASN.1.

4.2.3 Sequenceof

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

Modelo

Especificar una <sequence of> es igual que especificar una <generator transformation> que tiene un <sort> como primer <generator actual> y el nombre Emptystring como segundo <generator actual>. El identificador de generador en la <generator transformation> es el generador predefinido String en caso de que se especifique la palabra clave **sequence**, y si no, el identificador del generador es el generador predefinido Bag.

Si se especifica <sizeconstraint>, la <sequenceof> es un sintipo que tiene la <sizeconstraint> como <range condition> (véase 4.2.7).

Si se especifica <asn1 range condition>, la <sequenceof> es un sintipo que tiene la <range condition> tal como se especifica en la <asn1 range condition> (véase 4.2.7).

El género progenitor del sintipo (es decir, la <sequenceof> sin la <sizeconstraint> ni la <asn1 range condition>) tiene un nombre implícito y único y se define en la unidad de alcance más próxima que contiene el lugar donde aparece la <sequenceof>.

NOTA – El generador String se define en la Recomendación Z.100. El generador Bag se define en el Anexo A. En principio, la ordenación de los elementos en Bag no es importante, en contraposición a los elementos en String. Aunque los elementos están de hecho ordenados de alguna forma en el generador Bag predefinido (para aplicar condiciones de intervalo en cada elemento), no es una práctica adecuada utilizar esta información.

Ejemplo:

La definición:

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

es la misma que las tres definiciones del SDL:

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

Obsérvese que el constructivo **size** es una extensión de la gramática abstracta del SDL (véase 4.2.7).

4.2.4 Elección (Choice)

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

Modelo

Los <name>s en <namedsort>s de la <choice> deben ser distintos. Una <choice> es una sintaxis derivada para una <properties expression> que contiene:

- operadores "Make!", para crear valores de elección;
- operadores "Extract!", para asociar los <name>s en los <namedsort>s con la <choice> (véase 4.2.1);
- operadores "Present", para hacer que los <name>s en los <namedsort>s sean campos facultativos (véase 4.2.1);
- operadores "Modify!", para modificar los valores de elección. El nombre del operador utilizado para modificar un componente es el componente de campo en el <name> de campo concatenado con "Modify!".

Hay un operador "Make!" para cada <namedsort>. Cada operador "Make!" tiene el nombre "FMake!", donde F es el <name> de campo que aparece en el <namedsort>. La <argument list> del operador "FMake!" es el <sort> de campo que aparece en <namedsort>. El <result sort> para los operadores "FMake!" es el género que contiene la <properties expression>.

Los nombres y signaturas de los operadores "Extract!, "Modify!" y "Present" para los campos son iguales que para <secuence>s (véase 4.2.2).

<choice> también define implícitamente un género enumerado cuyos nombres de literales son idénticos a los nombres de campo de <choice>. El orden de los literales es el mismo orden en el cual se especifican los campos. Para describir el *Modelo*, los literales se numeran 1, 2, 3, ... n, siendo n el número de campos/literales, excluido el campo derivado (que se define más adelante). Hay un campo derivado implícito de este género enumerado implícito. Este campo indica el campo explícito que está presente en cualquier momento. El nombre del campo es "Present".

Para cada campo F, hay un conjunto de ecuaciones que describen el significado de la extracción del campo:

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

donde E es **error!**, salvo en la ecuación que menciona "FExtract!" y "FMake!" para el mismo campo. En este caso E es igual a A.

Para cada campo F, hay un conjunto de ecuaciones que describen el significado de probar la presencia:

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

donde B es False, salvo en la ecuación que menciona "FPresent" y "FMake!" para el mismo campo. En este caso B es igual a True.

Para cada campo F, hay un conjunto de ecuaciones que describen el significado de la modificación de un valor de elección:

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

Para cada campo F, hay un conjunto de ecuaciones que describen el valor del campo "Present" implícito:

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

Hay una ecuación que describe la igualdad de valores de elección:

```
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
```

NOTAS

1 El campo derivado y el tipo enumerado derivado es un campo que mantiene información sobre qué campo de elección está activo. De ese modo, el operador "PresentExtract" se puede utilizar para la multiderivación.

2 En esta Recomendación, los rótulos no tienen que distinguir necesariamente entre componentes del mismo tipo: se supone rotulación automática ASN.1.

Ejemplo:

El tipo elección:

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

es igual que:

```
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 Enumerado (Enumerated)

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

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

Las <simple expression>s en los <name value>s deben ser del género Integer y los valores deben ser disyuntos.

Modelo

Un <named number> que es un <name> es sintaxis derivada para un <named value> que contiene el <name> y una <simple expression> que denota el valor entero no negativo más bajo posible que no aparece en ningún otro <named value>s del género <enumerated>. La sustitución de <name>s por <names value>s se efectúa uno por uno de izquierda a derecha. Tras esta sustitución, los literales se reordenan en base a sus <simple expression>s.

Un género <enumerated> se representa como una especialización del género predefinido Enumeration, definido en el Anexo A. Los <name>s en los <named value>s son nuevos literales añadidos en orden ascendente sobre la base de su <simple expression> asociada. El género especializado tiene axiomas que definen First para denotar el valor del literal con la <simple expression> más baja, Last para denotar el valor del literal con la <simple expression> más alta, Succ para denotar el valor del literal más alto siguiente (si existe, y si no, **error!**), Pred para denotar el valor del literal más bajo siguiente (si existe, si no, **error!**) y Num para denotar el valor Integer de la <simple expression>.

La definición:

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

es igual que:

```
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;
```

NOTA – Un género enumerado también tiene operadores relacionales, pero como el género predefinido Enumerated se define con la palabra clave **ordering,** las propiedades de los operadores relacionales no se proporcionan en el subtipo.

4.2.6 Denominación de entero (Integer Naming)

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

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

El <identifier> debe denotar el género predefinido Integer del género predefinido Bit_String, y <simple expression>s en los <named value>s deben ser del género Integer.

Modelo

Especificar un <integernaming> es lo mismo que especificar un <existingsort> que contiene el <identifier> de Integer y especificar un <synonym definition item> para cada <named value> en la unidad de alcance contenedora más próxima.

Ejemplo:

La definición **newtype**:

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

es igual que:

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

NOTAS

1 Una <integernaming> proporciona un medio para agrupar definiciones sinónimas que están relacionadas de alguna manera. La agrupación puede ser conveniente para la legibilidad, pero desde el punto de vista de la presente Recomendación dicha agrupación no es importante.

2 No hay vinculaciones entre los valores sinónimos de Integer definidos por los <named values>s y el género Integer o Bit_string denotado por <integernaming>.

4.2.7 Subgama (Subrange)

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

Una <subrange> define la gama de un sintipo. Cuando una <sequenceof> o <selection> va seguida de (<range condition>), la <subrange> se aplica al <sort> contenido en vez de al constructivo <sequenceof> o <selection> entero.

Modelo

Especificar <subrange> como el <parent sort identifier> de un <syntype> es lo mismo que especificar el <sort> contenido y añadir la <range condition> después de la palabra clave **constants** en el <syntype> (si se especifica, y si no, se introduce el constructivo).

Ejemplo:

La definición de sintipo:

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

es equivalente a:

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

A continuación se describe cómo este constructivo corresponde con un sintipo en la Recomendación Z.100.

4.3 Condición de intervalo (Range condition)

La gramática abstracta para la condición de intervalo se cambia como sigue:

Range-condition :: Operator-identifier

Una condición de intervalo define un conjunto de valores. Se utiliza para definir un sintipo y para la derivación en una acción de decisión. Tiene un género progenitor asociado, que para un sintipo es el género especificado en la definición de sintipo, y para las acciones de decisión es la expresión de pregunta. Un valor está dentro del conjunto de valores si el operador denotado por el identificador de operador da True cuando se aplica al valor.

El identificador de operador para una condición de intervalo dada se define como:

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

donde o es el operador implícito y anónimo que aparece en la condición de intervalo, A es un género implícito y anónimo y S es el género progenitor.

La sintaxis concreta para la condición de intervalo se cambia como sigue:

<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> **)**

No es importante si en <innercomponent> se utiliza o no la palabra clave **from** o la palabra clave **with component**.

No es importante si <range list> figura o no entre paréntesis.

No es importante si el símbolo de dos puntos (:) o el símbolo de dos puntos suspensivos (..) se utiliza o no en <closed range>.

Hay ambigüedad sintáctica entre una <closed range> especificada con: y una <range> que comienza con <ground expression> que es <choice primary> (véase 4.4.1). En este caso, el contexto determina de cuál se trata (véase 2.2.2/Z.100). Una <ground expression> que es una <choice primary> debe encerrarse entre paréntesis cuando aparece como un <lowerendvalue> (por ejemplo, A(b:c):d)) o un <open range> (por ejemplo, A(b:c)) entero.

No es importante si en <range condition> se utiliza el símbolo coma (,) o el símbolo barra vertical (|).

Los <name>s de las <named constraint>s de <innercomponents> deben denotar nombres para campos facultativos (véase 4.2.1) del género progenitor.

Los nombres de campo deben ser distintos.

Cada <range> en <range condition> contribuye a las propiedades del operador que define el conjunto de valores:

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

Si un sintipo se especifica sin <range condition>, el resultado de operador es True.

En la siguiente explicación de cómo cada <range> contribuye al resultado de operador, V denota el valor de argumento. Cada contribución debe estar bien formada, lo que significa que los operadores utilizados deben existir con una signatura adecuada para el contexto.

- Si en una <closed range> no se especifican las palabras clave **min** y **max**, entonces <closed range> contribuye con:

  E1 rel1 V and V rel2 E2

  siendo E1 la <ground expression> de <lowerendvalue> y E2 la <ground expression> de <upperendvalue>.

  Si se especifica "<" para <lowerendvalue>, entonces rel1 es el operador "<", si no, es el operador "<=".

  Si se especifica "<" para <upperendvalue>, entonces rel2 es el operador "<", si no, es el operador "<=".

  Si se especifica la palabra clave **min** y no se especifica la palabra clave **max**, <range condition> contribuye con:

  V rel2 E2

  Si se especifica la palabra clave **max** y no se especifica la palabra clave **min**, <range condition> contribuye con:

  E1 rel1 V

  Si se especifican las palabras clave **min** y **max**, el operador siempre da verdadero.

- Una <open range> contribuye con:

  V rel E

  donde E es la <ground expression> de <open range> y rel es el operador infijo de <open range>. Si no se especifica ningún operador infijo en la <open range>, rel es el operador =.

- Una <contained subrange> contribuye con:

  o1(V)

  donde o1 es el operador implícito que define el conjunto de valores para el <sort> mencionado en <size constraint>.

- Una <sizeconstraint> contribuye con:

  o1(Length(V))

  donde o1 es el operador implícito que define el conjunto de valores de <range condition> mencionada en la <contained subrange>.

- <innercomponent> contribuye con:

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

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

  la que sea apropiada para el género de V. o es el operador implícito <innercomponent> que contribuye a y o1 es el operador implícito para la <range condition> especificada en <innercomponent>.

  <innercomponent> tiene una contribución para cada <named constraint> contenida que especifica constricciones del campo (véase 4.2.1) denotado por el <name> del género progenitor.

  Si se omite el símbolo **...**, se añade la palabra clave **present** a las <named constraint>s que no tienen palabra clave final (**present**, **absent** u **optional**), y <named constraint>s de la forma:

  <name> **absent**

  se añaden para todos los campos (es decir, <name>s) no mencionados explícitamente en una <named constraint>. Las <named constraint>s se añaden a los <innercomponents> antes de derivar las contribuciones de cada <named constraint>.

Si se especifica una <range condition> para una <named constraint>, la contribución es:

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

donde E es la constricción presente para el campo, F (del nombre de operador FPresent) es el nombre del campo facultativo y o1 es el operador implícito para <range condition>. Si se omite <range condition>, la contribución es únicamente la constricción presente E.

La constricción presente para un campo F es:

FPresent(V)

cuando <named constraint> para el campo contiene la palabra clave **present**; y

not FPresent(V)

cuando la <named constraint> para el campo contiene la palabra clave **absent**. En todos los otros casos, la constricción presente es True.

4.4 Expresiones de valor

<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>

NOTAS

1 En <extended primary>, <choice primary> y <composite primary> son adicionales a la Recomendación Z.100.

En <extended identifier>, <string primary> es adicional a la Recomendación Z.100.

2 <extended primary> es parte de expresiones, mientras que <extended literal identifier> es parte de términos en ecuaciones.

4.4.1 Primario de elección (Choice primary)

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

Modelo

<choice primary> es sintaxis derivada para una <operator application> que tiene <primary> como argumento. El <operator identifier> en <operator application> contiene el <qualifier> de <identifier>, y un nombre de operador que es el <name> en <identifier> seguido por "Make!".

Ejemplo:

<choice primary>:

Myvalue : Mychoice

es sintaxis derivada para:

MyvalueMake!(Mychoice)

Si un <choice primary> particular puede denotar una de varias aplicaciones de operador (es decir, un campo de más de un género elección), se utiliza un calificador:

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

que es sintaxis derivada para:

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

NOTA – Los nombres de operador que consisten en un nombre de campo seguido por "Make!" están disponibles en cualquier género construido utilizando <choice>. Por lo tanto, la notación <choice primary> se utiliza principalmente en esos géneros. Sin embargo, la notación es adecuada también para valores **any**, por ejemplo, un operador RealMake! se puede definir con la signatura:

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

lo que implica que la notación:

Real : <expression>

se pueda utilizar para "convertir" una expresión **any** en un valor real.

4.4.2 Primario compuesto (Composite primary)

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

<Composite primary> es un valor construido utilizando sus componentes constituyentes. Hay ambigüedad sintáctica entre las diversas alternativas en <composite primary>. En esos casos, debe ser determinado por el contexto (véase 2.2.2/Z.100) lo que denota <composite primary>.

NOTA – Esto supone probablemente que los instrumentos sólo distinguen <sequencevalue> y <sequenceofvalue> en la sintaxis concreta. Como parte del análisis estático, se puede determinar si una <sequenceofvalue> denota en cambio una de las alternativas restantes.

4.4.2.1 Valor de secuencia (Sequence value)

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

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

El género de <sequencevalue> debe tener los <name>s que aparecen en los <namedvalue>s como campos asociados (véase 4.2.1). La <expression> en cada <namedvalue> debe ser del género del campo proporcionado por el <name>. Cada campo no facultativo asociado al género se debe mencionar exactamente una vez, y cada campo facultativo asociado al género se debe mencionar como máximo una vez.

Modelo

<sequencevalue> es sintaxis derivada para:

- Una aplicación de operador de un operador con el nombre "Make!". Toma como argumentos las <expression>s de los campos no facultativos. Las <expression>s en la aplicación de operador derivada aparecen ordenadas alfabéticamente según los <name>s del campo. Si todos los campos asociados al género son facultativos, el constructivo es un identificador de literal con el nombre "Empty".
- Si el género tiene campos facultativos, la aplicación de operador se da como primer argumento para una aplicación de operador de un operador con el nombre "FModify!", donde F es alfabéticamente el primer <name> de los campos facultativos mencionados en <sequencevalue>. El segundo argumento del operador "FModify!" es la <expression> asociada al campo. La aplicación de operador resultante viene dada a su vez como primer argumento para el operador "FModify" del siguiente campo facultativo, y así sucesivamente, hasta que se han considerado todos los campos facultativos.

Ejemplo:

Si el género S tiene cuatro campos A, B, C y D asociados, y C y D son facultativos, entonces <composite primary>:

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

es sintaxis derivada para:

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

El orden de los <namedvalue>s en la <sequencevalue> no es importante, por ejemplo:

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

denota la misma expresión. El <qualifier> se puede omitir:

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

en cuyo caso debe ser posible determinar por el contexto (véase 2.2.2/Z.100) de qué género se trata.

NOTA – Las <sequencevalue>s se utilizan principalmente para construir valores de géneros definidos con el constructivo <sequence>, dado que dichos géneros tienen un operador "Make!" cuyos argumentos son valores de campo ordenados alfabéticamente de acuerdo con los nombres de campo. Para géneros definidos con el constructivo <structure definition>, a menudo es mejor utilizar <structure primary> pues el orden de aparición es importante para los operadores "Make!" de dichos géneros.

4.4.2.2 Valor secuencia de (Sequenceof value)

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

Modelo

<sequenceofvalue> es sintaxis derivada para:

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

donde E1, E2, ..., En son las <expression>s de <sequenceofvalue> en el orden de aparición. Si no se especifica ninguna <expression>s, <sequenceofvalue> es sintaxis derivada para el nombre Emptystring.

NOTA – Una <sequenceofvalue> puede denotar un valor del género predefinido Octet_string, aunque en la Recomendación X.680 (en contraposición a esta Recomendación) se exige que en este caso las <expression>s sean <identifier>s.

4.4.2.3 Valor de identificador de objeto (Object Identifier value)

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

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

Modelo

Si el <objidcomponent> más a la izquierda es un <identifier> sinónimo o variable del género predefinido Object_identifier, entonces debe seguir al menos un <objidcomponent>. El <objectidentifiervalue> es igual que la expresión:

Id // {Olist}

donde Id es el primer <objidcomponent> y Olist es el <objidcomponent>s restante.

Si un <objidcomponent> se especifica con <ground expression>, es sintaxis derivada para definir al <identifier> como un **synonym** para <ground expression> en la unidad de alcance contenedora más próxima, y omitir <ground expression> en el <objidcomponent>.

Después de las anteriores transformaciones, todos los <objidcomponent>s son <identifier>s que deben denotar una variable o un sinónimo del género predefinido Object_element (véase el Anexo A). El constructivo es sintaxis derivada para:

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

donde Id1, Id2, ..., Idn son los <identifier>s del <objectidentifiervalue> dado en el mismo orden que en el <objectidentifiervalue>.

Ejemplo:

<objectidentifiervalue>:

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

es igual que:

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

donde root (raíz) es una variable o un sinónimo del género predefinido Object_Identifier, y una definición sinónima de level4 se deriva en la unidad de alcance contenedora más próxima:

synonym level4 Object_element = 4;

4.4.2.4 Valor real (Real value)

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

<mantissa> ::= <expression>

<base> ::= <simple expression>

<exponent> ::= <expression>

Las tres expresiones constituyentes deben ser del género Integer. La <base> debe ser idéntica al valor 2 ó 10.

Modelo

Un <realvalue> se representa como la <expression>: (<mantissa>)*Power(<base>,<exponent>) donde Power es un operador adicional definido para el género Real.

El operador tiene la signatura:

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

y las ecuaciones:

```
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);
                       )
```

Ejemplo:

La expresión:

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

es igual que:

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

que es igual a 1.5.

4.4.3 Primario de cadena (String primary)

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

<bitstring>s y <hexstring>s son los literales de los géneros predefinidos Bit-string y Octet_string (véase el Anexo A).

El contexto o la utilización de un <qualifier> determina si cualquier <bitstring> o <hexstring> específico es un literal de Bit_string o de Octet_string.

Modelo

Un <string primary> que contiene <quoted string> representa un <character string literal identifier> que consiste en el <qualifier> y un <character string literal> con el mismo <text> que <quoted string>.

Un <string primary> que contiene una <bitstring> o <hexstring> representa un identificador que consiste en <qualifier> y un <name> formado por la supresión de los apóstrofos de <bitstring> o <hexstring>.

NOTAS

1 Una <quoted string> es una cadena de caracteres en la cual se han utilizado signos de comillas (") en lugar de apóstrofos.

2 Como norma, las notaciones de valores de cadenas binarias y de cadenas hexadecimales deben utilizar apóstrofos, por ejemplo, debe utilizarse '01'B en vez de 01B.

Reemplazada por una versión más reciente

Anexo A

SDL en combinación con datos predefinidos de ASN.1

(Este anexo es parte integrante de esta Recomendación)

En este anexo se define el lote Predefined, que contiene los géneros de datos predefinidos y los generadores de datos.

```
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 */

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

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

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

/*   Definition of ASN.1 character strings          */

syntype
    IA5String  =  Charstring
endsyntype;

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

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

endsyntype;

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

endsyntype;

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

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

/*      Usage      */

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

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

/*      Usage                    */

/*
```

ASN.1 INTEGER definitions can declare additional constants:
INTEGER { name-1(number-1), ... ,name-n(number-n) }
Such constants are defined using synonyms:

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

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

/*   Enumeration sort */

/*   Definition    */

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

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

*/

synonym  PLUS_INFINITY        Real  =  external,
           MINUS_INFINITY     Real  =  external;

/*      Usage      */

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

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

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

/* Bag generator  */

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

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

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

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

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

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

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

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

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

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

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

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

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

/*  Usage       */

/*
```

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

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

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

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

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

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

/* ASN.1 Bit sorts */

/* Definition */

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

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

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

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

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

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

The named bits can be represented by:

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

/* ASN.1 Octet sorts  */

/* Definition     */

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

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

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

endnewtype Octet_String;

/* Usage */

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

/* ASN.1 Null sort */

/* Definition */

newtype Null
    literals Null;
endnewtype Null;

/* Usage */

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

/* ASN.1 Object Identifier sort */

/* Definition */

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

newtype Object_Identifier String (object_element, EmptyString )
endnewtype Object_Identifier;

/* Usage */

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

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

/* ASN.1 ANY sort */

/* Definition */

newtype Any_type
endnewtype Any_type;

/* Usage */

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

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

*/

/*   ASN.1 Useful types */

GeneralizedTime      ::=   VisibleString;

UTCTime              ::=   VisibleString;

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

ObjectDescriptor::=  GraphicString;

endpackage Predefined;
```

Apéndice I

Restricciones de la ASN.1 y del SDL

(Este apéndice no es parte integrante de esta Recomendación)

I.1 Resumen del subconjunto de ASN.1 admitido

En esta subcláusula figura un resumen del subconjunto de ASN.1 admitido en la presente Recomendación.

En principio se admite la versión de ASN.1 definida en la Recomendación X.680. No se admiten las características de las Recomendaciones X.681, X.682 y X.683 es decir, la especificación de objetos de información, la parametrización de especificaciones ASN.1 y especificaciones de constricciones. Además, el tipo ANY que se define en la Recomendación X.208 se admite parcialmente.

Las principales restricciones de la ASN.1 son:

- No se admiten macros. Esto significa que tampoco se admite la macro operación. En el SDL no se necesita la macro operación, porque existen mecanismos más adecuados para la definición de operaciones, a saber, intercambio de señales o procedimientos a distancia. El Apéndice II contiene un ejemplo de la sustitución de la operación macro por constructivos del SDL.
- Se autoriza la utilización de rótulos, pero no tienen significado: se pasan por alto. El motivo es que los rótulos se utilizan únicamente para codificación, y la codificación está fuera del ámbito del SDL.
- Las reglas de codificación están fuera del ámbito de la presente Recomendación. Es posible que los fabricantes sustenten las reglas de codificación, pero esto depende de la realización.
- No se admite ninguna notación de valor para ANY (o ANY DEFINED BY). Como se indica en 4.4.1, al introducir operadores especiales se puede imitar una forma limitada de notación de valor para ANY. En algunos casos, ANY se puede sustituir por CHOICE en los tipos para los cuales se necesitan notaciones de valor. El Apéndice II contiene un ejemplo de esto.

- No se admite el guión en los nombres ASN.1 dentro de descripciones SDL. Se autoriza la utilización de guiones en los nombres en módulos ASN.1 que son importados en el SDL, pero cuando se importan en el SDL, los guiones se deben transformar en subrayado. El motivo de esta restricción es que los guiones se consideran como el operador «menos» en el SDL.

- La sensibilidad a los casos no se sustenta. El motivo de esta restricción es que el SDL es independiente de los casos.

  La restricción entraña que la introducción de dos tipos con el mismo nombre (aparte de la sensibilidad a los casos) es un error. Sin embargo, se puede tener el mismo nombre si son de diferentes clases de entidad. Las clases de entidad son, por ejemplo, nombres de tipo, nombres de valor e identificadores, es decir, se autoriza:

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

- Las definiciones deben terminar con un punto y coma. La razón es que las definiciones SDL terminan con un punto y coma. Si el punto y coma no fuera obligatorio, la gramática sería ambigua, porque el SDL es independiente de los casos.

- El tipo REAL de ASN.1 no se representa como una secuencia de tres enteros, como en la Recomendación X.680. Se utilizará en su lugar la notación de valor de la Recomendación X.208, es decir { 314, 10, –2 } en vez de { mantisa 314, base 10, exponente –2 }. Alternativamente, se puede utilizar la sintaxis SDL para denotar valores REAL, es decir, también se autoriza 3.14. En consecuencia, no se sustenta ninguna subtipificación de mantisa, base o exponente, ni operadores para acceder a la mantisa, la base o el exponente de un valor REAL, ni cambiarlos.

- Se debe evitar la utilización del mismo identificador para números o bits denominados de diferentes tipos en el mismo ámbito. El motivo es que los números y bits denominados corresponden con sinónimos de enteros del SDL. La utilización del mismo identificador dos veces daría como resultado un SDL ilegal (redefinición del mismo sinónimo).

  Es decir, que no se autoriza la doble utilización de 'notAllowed' en las siguientes definiciones de tipo:

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

  ni en la siguiente:

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

  Se autoriza la doble utilización del mismo identificador en diferentes tipos enumerados, o en un tipo enumerado y en un entero o un bit denominados, porque los identificadores en tipos enumerados no corresponden con sinónimos de enteros. Es decir, que se puede utilizar:

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

- En la referencia de tipo y valor externos se deben dejar espacios alrededor de ".", por ejemplo, Modulereference . Typereference en vez de Modulereference.Typereference. La omisión de los espacios crearía problemas sintácticos en el SDL, porque los identificadores en el SDL pueden contener puntos.

- Los valores de componente OBJECT IDENTIFIER asignados por el UIT-T, la ISO, o ambos, no están definidos en el lote Predefined, porque no se pueden mantener en esta Recomendación. Por ejemplo, para utilizar:

  ```
  {  ccitt recommendation z }
  ```

  los valores de componente ccitt, recommendation y z tienen que estar definidos en un lote definido para el usuario.

I.2 Resumen del subconjunto de SDL admitido

Esta subcláusula contiene un resumen del subconjunto de SDL admitido en esta Recomendación.

Se autoriza la plena utilización del SDL, con las siguientes excepciones:

- No se autoriza la utilización de los caracteres {, }, [,], y | (barra vertical) en nombres, porque tienen un significado especial en la ASN.1.
- Esta Recomendación introduce varias nuevas palabras claves que no pueden ser utilizadas como nombres. Estas palabras claves se enumeran en la cláusula 3.
- Un punto en un nombre no debe estar seguido por un subrayado ni por otro punto. Un subrayado en un nombre no debe estar seguido por un punto. Es decir, no se autorizan los siguientes nombres:

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

- No se admite la utilización de la doble negación (por ejemplo, 1 -- 2). La razón es que -- se utiliza para los comentarios ASN.1.
- No se autorizan los separadores y comentarios dentro de la definición de operadores entre corchetes, porque crearía ambigüedad sintáctica. Por ejemplo, no se autoriza lo siguiente:

  operators

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

- Un subrayado en un nombre contenido en un primario compuesto se debe especificar explícitamente. Por ejemplo, para la siguiente definición de tipo:

  T ::= SEQUENCE
  {a_b INTEGER},

  no se autoriza la siguiente notación de valor:

  tT ::= {a b 5}

  Se debe utilizar en su lugar,

  tT ::= {a_b 5}

Apéndice II

Ejemplos

(Este apéndice no es parte integrante de esta Recomendación)

Este apéndice contiene ejemplos de la utilización de la ASN.1 en combinación con el SDL. Se supone que el lector está familiarizado con la ASN.1 y con el SDL. Sólo se utiliza la representación gráfica del SDL.

La finalidad de este apéndice es mostrar los principios para utilizar el SDL en combinación con la ASN.1. Se abordan los siguientes temas generales:

- Definición de tipos de datos ASN.1 en diagramas SDL;
- Definición de valores ASN.1 en diagramas SDL;
- Utilización de operadores en tipos de datos ASN.1 que se definen en la presente Recomendación;
- Cómo soslayar algunas restricciones impuestas a la ASN.1 por la presente Recomendación;
- Una «guía de estilo» para utilizar la ASN.1 en combinación con el SDL.

La finalidad no es dar una versión completa de todos los operadores en todos los tipos de datos ASN.1, ya que éstos figuran en otras partes de la presente Recomendación.

II.1 Definición de tipos de datos ASN.1 en SDL

Hay tres formas de definir tipos de datos ASN.1 en SDL:

1) importando una definición de tipo de un módulo ASN.1;
2) definiendo el tipo directamente en SDL, utilizando la sintaxis ASN.1;
3) definiendo el tipo directamente en SDL, dentro de una cláusula newtype SDL.

II.1.1 Importación de una definición de tipo de un módulo ASN.1

Un tipo ASN.1 definido en un módulo ASN.1 puede ser importado en el SDL utilizando el constructivo IMPORTS, que se puede poner en un símbolo de texto SDL.

Los tipos Address, DisconReason, y DisconIndParameters se definen en el módulo ASN.1 ProtocolData que se muestra a continuación:

```
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
```

Estos tipos pueden ser importados en el SDL como se muestra en la Figura II.1.1.1.

FIGURA II.1.1.1/Z.105

Importación de tipos ASM.1 en SDL

II.1.2 Definición de tipos ASN.1 directamente en SDL

Un tipo ASN.1 puede ser definido directamente en SDL (es decir, sin ser importado de un módulo ASN.1), utilizando sintaxis ASN.1 normal. La única diferencia es que hay que poner un punto y coma al final de una definición de tipo. Las definiciones de nuevos tipos de datos se ponen en símbolos de texto. Esto se ilustra en la siguiente Figura II.1.2.1:

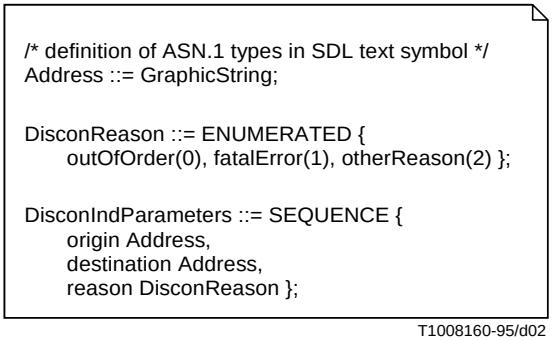

FIGURA II.1.2.1/Z.105

Definición de tipos ASN.1 directamente en SDL

También se puede declarar que las variables son de un tipo ASN.1, según se ilustra en la Figura II.1.2.2.

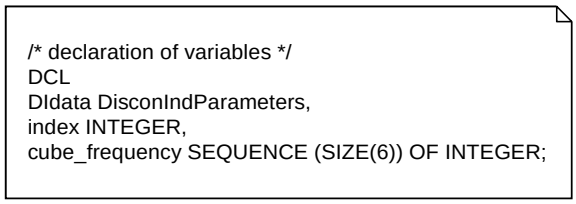

FIGURA II.1.2.2/Z.105

Declaración de variables de tipos ASN.1

II.1.3 Operadores definidos por el usuario en tipos ASN.1

Es posible definir tipos ASN.1 en SDL y añadir a ese tipo operadores definidos por el usuario. Con esa finalidad, el tipo ASN.1 se puede definir directamente en SDL dentro de una cláusula *newtype.*

Como ejemplo se toma un tipo Rectangle que contiene información sobre la longitud y la anchura de un rectángulo. Este tipo ASN.1 se puede definir como:

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

Es posible definir en SDL una «área» de operador que devuelve el área (longitud × anchura) del rectángulo. Hay dos formas de hacer esto:

1) *Introducción de un nuevo tipo que hereda del Rectangle*

   Si el tipo Rectangle ya está definido en algún sitio (por ejemplo en un módulo ASN.1), se puede introducir un nuevo tipo que hereda de Rectangle, y añade un nuevo operador. Esto se muestra en la Figura II.1.3.1, en la cual el tipo Rectangle2 hereda de Rectangle, pero tiene además un operador. El operador sólo se puede utilizar en valores de tipo Rectangle2, pero no se puede utilizar en valores de tipo Rectangle.

   Obsérvese que en el axioma se utiliza Make! (len, wid), en vez de la notación de valor ASN.1 {longitud len, anchura wid}.

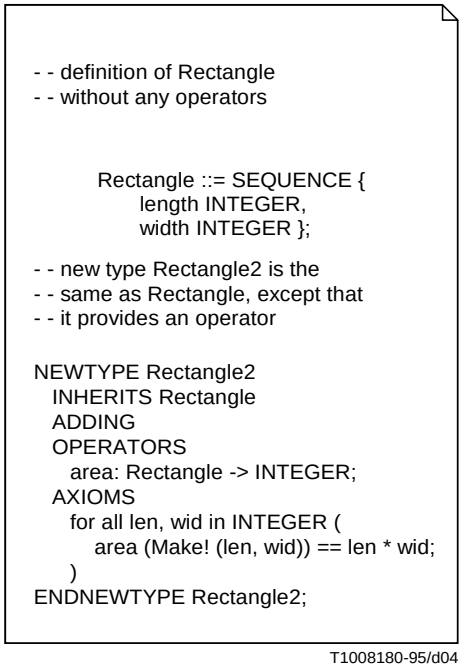

FIGURA II.1.3.1/Z.105

Herencia de un tipo existente y adición de un operador

2) *Definición del rectángulo y el operador desde el principio*

Si aún no se ha definido el rectángulo, es posible definir el tipo y el operador en una definición de tipo. Con esta finalidad, Rectangle se define dentro del constructivo newtype SDL, y el operador se añade después de la definición de los campos, como se muestra en la Figura II.1.3.2.

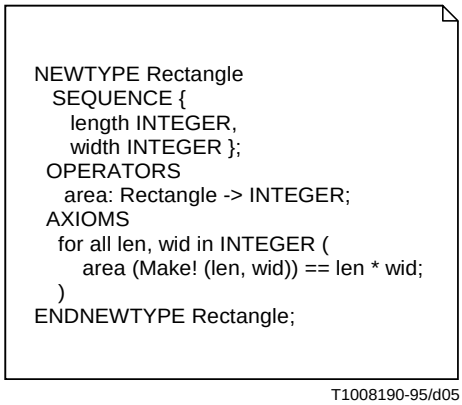

FIGURA II.1.3.2/Z.105

Definición de operadores en tipos ASN.1

II.2 Utilización de valores ASN.1 en SDL

Los valores ASN.1 se pueden utilizar en expresiones del SDL. Las expresiones se utilizan en varios lugares en el SDL, por ejemplo:

- en una asignación a una variable;
- en símbolos de decisión;
- en los parámetros de un resultado de creación de una señal, procedimiento de llamada o proceso;
- en la definición de constantes.

La manera más sencilla de utilizar valores ASN.1 es utilizando la notación de valor ASN.1 normal. La Figura II.2.1 muestra la utilización de un valor ASN.1 en un resultado de señal SDL DisconInd. Se supone que esta señal transporta como parámetro el tipo ASN.1 DisconIndParameters definido en II.1.1.

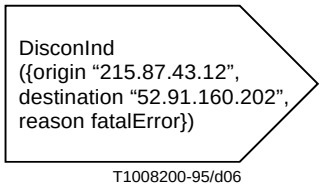

FIGURA II.2.1/Z.105

Valor ASN.1 en un resultado SDL

La notación de valor ASN.1 también se puede utilizar para definir constantes en el SDL, igual que para definir valores en el ASN.1, tal como se muestra en la Figura II.2.2.

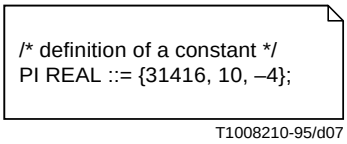

FIGURA II.2.2/Z.105

Definición de una constante mediante la notación de valor ASN.1

II.3 Ilustración de la utilización de algunos tipos ASN.1 en SDL

La ASN.1 no tiene operadores. En el SDL, se necesitan operadores en los datos ASN.1 para comparar y manipular los datos. En esta subcláusula se ilustra la utilización de algunos operadores en tipos ASN.1. El conjunto completo de operadores predefinidos y su significado se definen en el anexo A y en el cuerpo de esta Recomendación.

II.3.1 Operadores en tipos ASN.1 simples

Comparación de valores de datos

Para cada tipo de dato ASN.1 se dispone por lo menos de dos operadores: = (igual) y /= (no igual). Todos los tipos ordenados también tienen operadores < (menor que), > (mayor que), <= (menor o igual que) y >= (mayor o igual que).

Con frecuencia se combinan los operadores de comparación con operadores booleanos (and, or, not, etc.), por ejemplo, en los símbolos de decisión. En la Figura II.3.1.1 se muestran tres ejemplos de operadores de comparación.

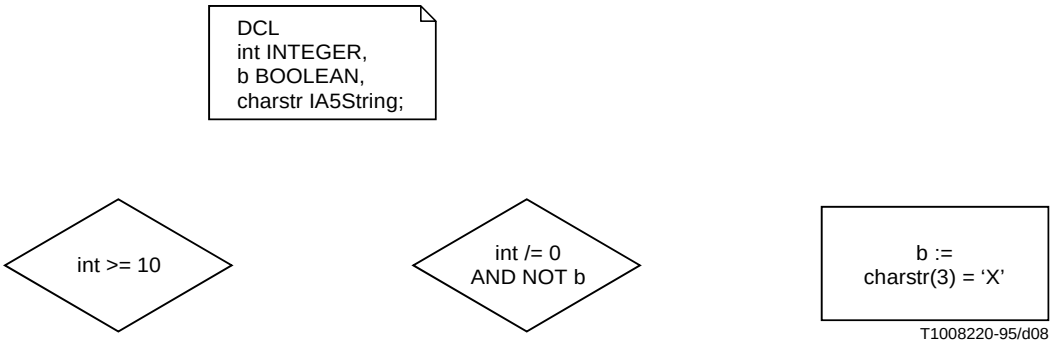

FIGURA II.3.1.1/Z.105

Operadores de comparación y operadores booleanos

Valores de manipulación

Hay varios operadores definidos en tipos simples ASN.1, como +, –, *, / en enteros y reales. La Figura II.3.1.2 muestra algunos:

- + para añadir dos enteros (índice + 1) o reales;

- // para concatenar cadenas de caracteres (prefijo // "X" // sufijo) o cadenas binarias ('03FC'H // Mkstring(1)). Este operador está disponible para todos los tipos de cadena (es decir, cadenas de caracteres, tipos BIT STRING, OCTET STRING, SEQUENCE OF).
- Mkstring para hacer que una cadena consista en un elemento (Mkstring(1)). Este operador está disponible para todos los tipos de cadena.

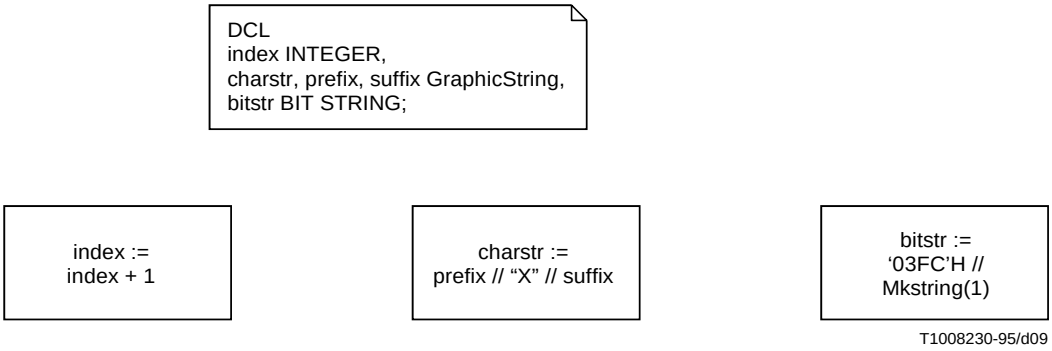

FIGURA II.3.1.2/Z.105

Ejemplos de operadores en tipos simples

En el Anexo A se definen todos los operadores en tipos simples.

II.3.2 SECUENCIA (SEQUENCE)

Para los tipos definidos con SEQUENCE, se dispone de un operador para acceder al valor de un componente. Como ejemplo se toma el tipo ConReqData, que está definido en la especificación ASN.1 parcial que figura a continuación. Al componente «Receiver» de variable v de ConReqData se le puede asignar valor, y acceder al mismo utilizando v!Receiver, según se ilustra en la Figura 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 };
```

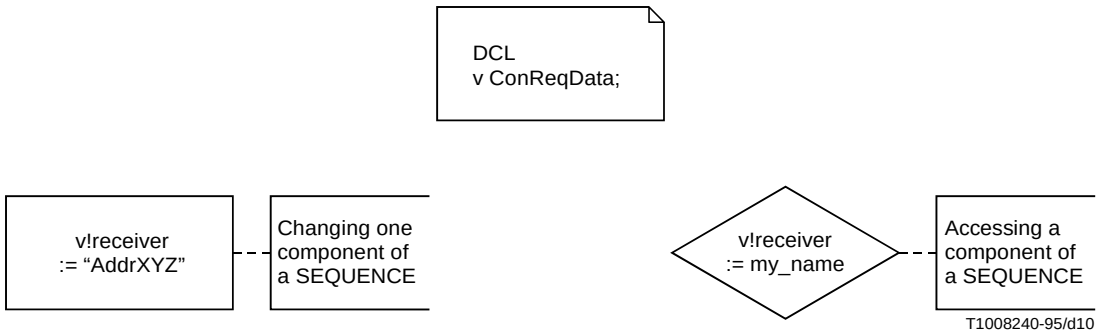

FIGURA II.3.2.1/Z.105

Acceso a un componente de una SEQUENCE

Siempre es posible leer un componente por defecto, aun cuando su valor esté ausente: en tal caso, se devuelve el valor por defecto.

La lectura de un componente facultativo exige especial atención: si el componente está ausente resulta un error dinámico. Antes de acceder a un componente facultativo, siempre hay que investigar si el componente está presente o no. Para esto se puede utilizar el operador '<identifier>Present', que se muestra en la figura II.3.2.2.

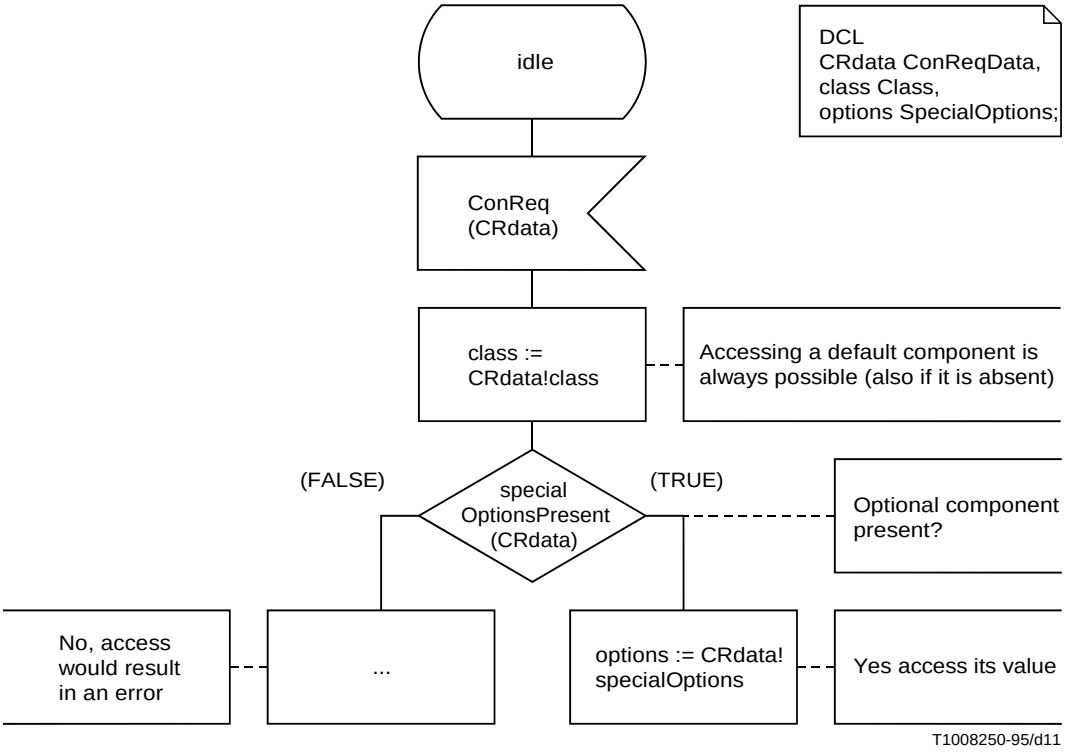

FIGURA II.3.2.2/Z.105

Acceso a componentes por defecto y facultativos

II.3.3 SECUENCIA DE (SEQUENCE OF)

La Figura II.3.3 contiene un ejemplo de operadores en tipos SEQUENCE OF. La «lista» de variables contiene una secuencia ordenada de enteros. Al recibir la señal 'nr', se inserta en la lista un nuevo entero.

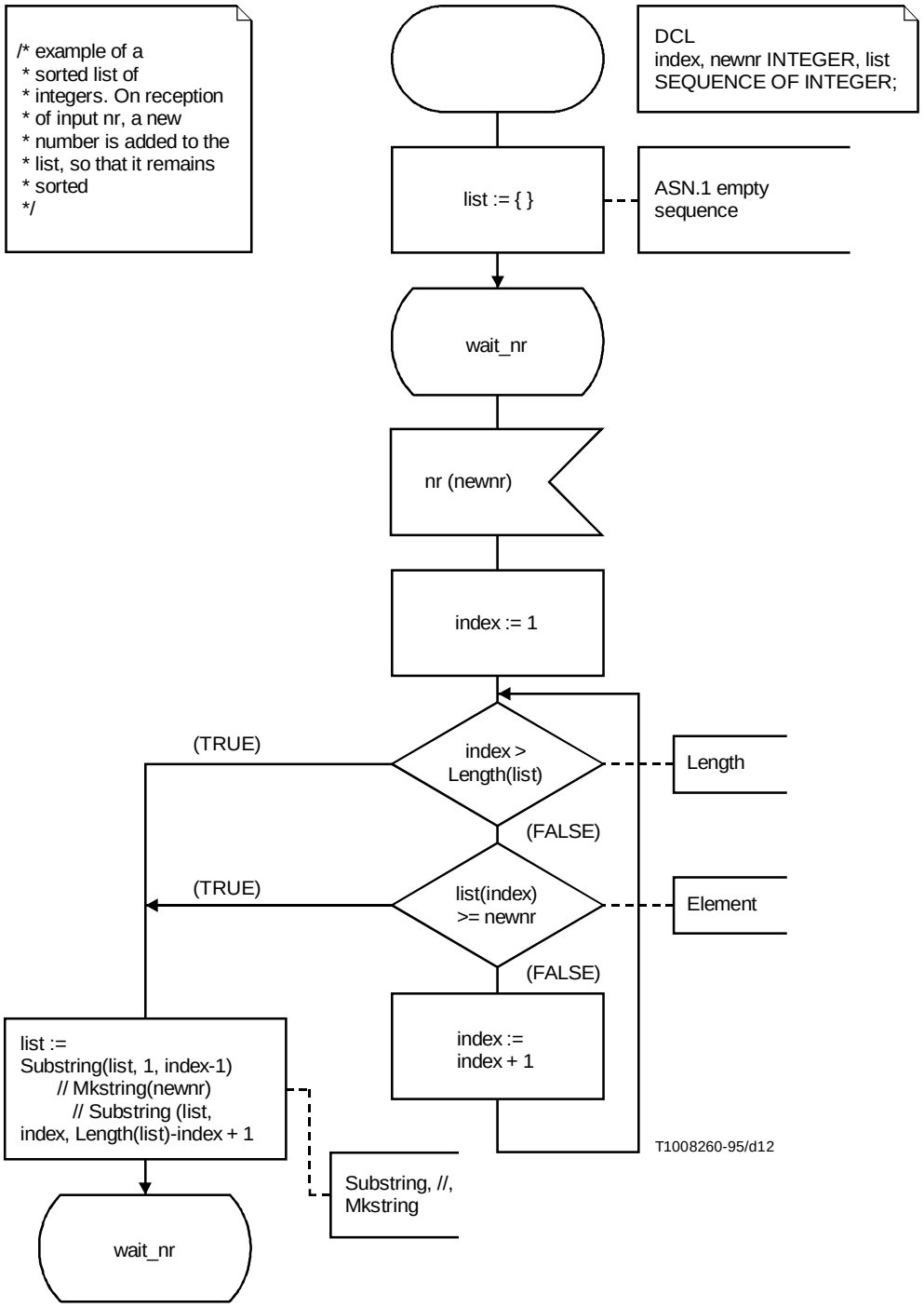

FIGURA II.3.3/Z.105

Utilización de los operadores predefinidos para SEQUENCE OF

En este ejemplo se utilizan varios operadores en SEQUENCE OF:

- list(index) indiza un elemento de una SEQUENCE OF;
- Length(list) devuelve la longitud de una cadena;
- Mkstring(newnr) hace que una cadena contenga un ítem;
- Substring(list, 1, index-1) devuelve una subcadena de lista, comenzando en la posición 1 (la primera posición) y con índice de longitud -1;
- ... // ... devuelve la concatenación de dos cadenas.

II.3.4 CONJUNTO DE (SET OF)

El tipo SET OF de ASN.1 se puede utilizar para representar conjuntos. En este ejemplo se ilustra la utilización de SET OF para establecer modelos de procesamiento de llamadas telefónicas. Se supone que un abonado puede activar varios de los servicios llamados suplementarios.

Los tipos de datos para estos servicios suplementarios se definen con los tipos ASN.1 que se indican a continuación. SupplementaryService es un tipo enumerado que tiene todos los servicios como valores. ServiceSet es el tipo que tiene conjuntos de servicios suplementarios como valores.

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

ServiceSet ::= SET OF SupplementaryService;
```

La Figura II.3.4 muestra un fragmento de un proceso SDL que efectúa el procesamiento de llamada telefónica para un abonado. Se utiliza la variable ActiveServ del tipo ServiceSet para seguir la pista de los servicios que están activos para un determinado abonado. Un servicio se puede activar enviando la señal Activate, y se puede desactivar enviando la señal Deactivate.

Si el abonado descuelga, se verifica si la línea directa está activa. Si éste es el caso, se llama inmediatamente a un número predefinido, sin esperar que el abonado marque un número. Si la línea directa no está activa, se envía al abonado un tono de marcación, después de lo cual el abonado puede marcar un número.

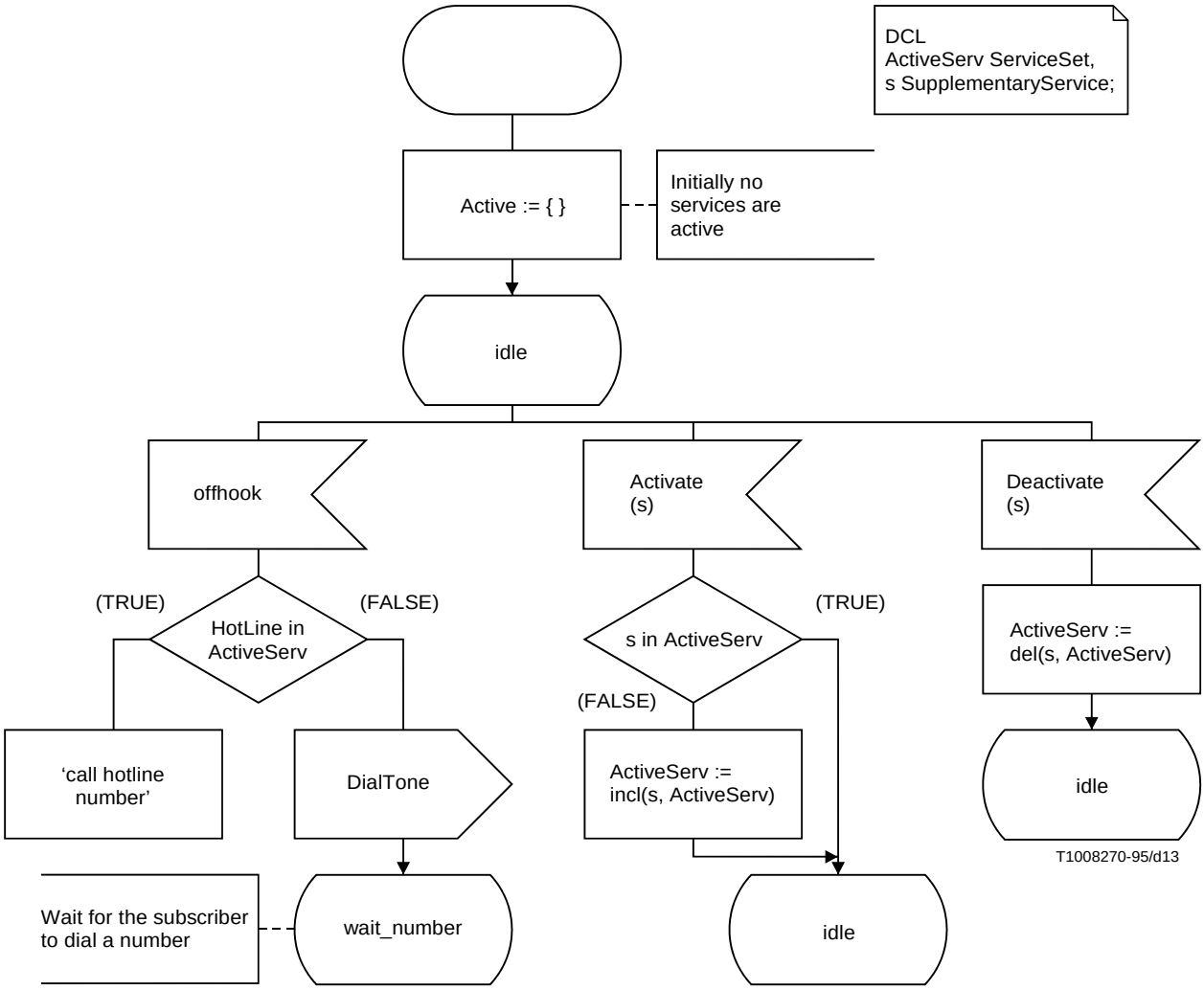

FIGURA II.3.4/Z.105

Utilización de operadores predefinidos para SET OF

En el ejemplo se ilustran diversos operadores en SET OF:

- s en ActiveServ: da TRUE si el servicio s es miembro de un conjunto ActiveServ.

- incl (s, ActiveServ): proporciona el conjunto ActiveServ + element s. Obsérvese que en la ASN.1, es importante el número de veces que aparece un elemento en un SET OF: {HotLine, HotLine} no es igual que {HotLine}. En el ejemplo, no tiene sentido activar un servicio más de una vez. Por lo tanto, se verifica si s ya es miembro de Active antes de añadir realmente s al conjunto.

- del (s, ActiveServ): da el conjunto ActiveServ del cual se suprime el elemento s.

II.3.5 ELECCIÓN (CHOICE)

Se puede asignar un valor a una variable de un tipo CHOICE utilizando la notación de valor ASN.1 para CHOICE. Obsérvese que hay que proporcionar el identificador, y que debe haber una coma entre el identificador y el valor. Se puede asignar también un valor a una variable utilizando un operador SDL (<variable>!<identifier>). Estas dos formas se ilustran en la Figura II.3.5.

El acceso a un componente no seleccionado de un valor CHOICE produce un error dinámico. Estos errores se pueden evitar utilizando el !present-operator para determinar la alternativa que se seleccionó antes de acceder a un componente. Esto se ilustra en la Figura II.3.5.

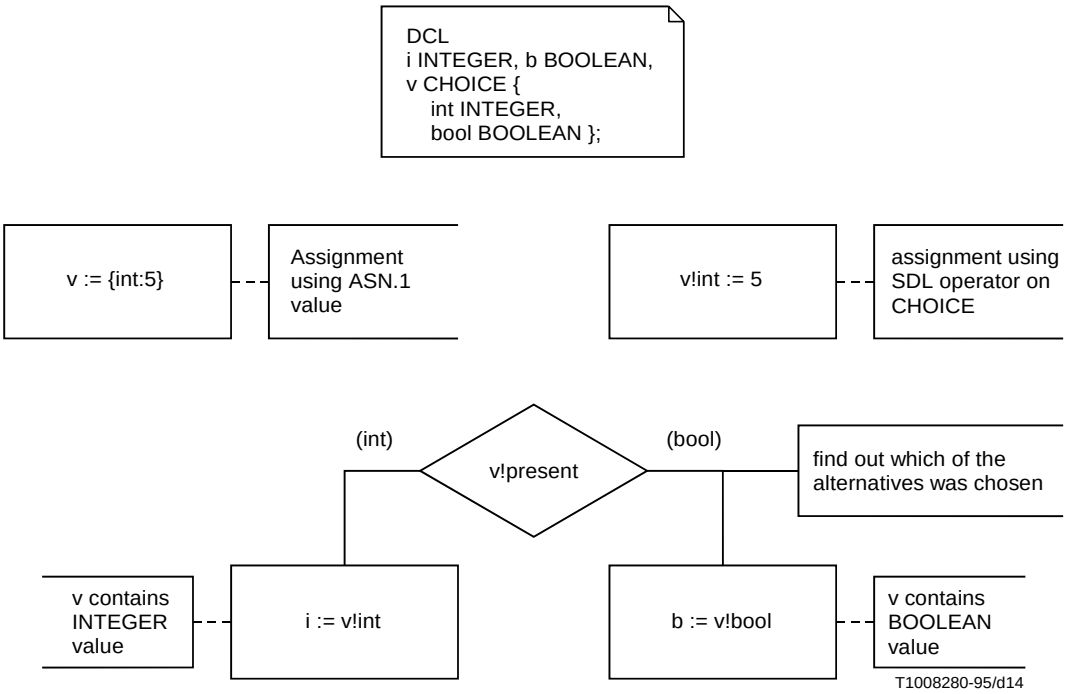

FIGURA II.3.5/Z.105

Utilización de CHOICE en combinación con el SDL

II.4 Directrices para soslayar las restricciones a la ASN.1

A continuación se proporcionan algunas directrices para tratar algunos constructivos ASN.1 que no se admiten en combinación con el SDL. La finalidad no es proporcionar todas las soluciones posibles.

II.4.1 Clase de objeto de información ANY DEFINED BY/TYPE-IDENTIFIER

ANY DEFINED BY (Recomendación X.208, sustituido en la Recomendación X.681 por la clase de objeto de información útil TYPE-IDENTIFIER) se emplea a menudo para especificar que el tipo de un parámetro depende del valor de algún otro parámetro. Por ejemplo: supóngase que un proceso tiene varios atributos, posiblemente de tipos diferentes. Un atributo es direccionado por su identificador. Es posible tener una operación general para leer un atributo. El tipo de la respuesta de la operación Leer depende del atributo que fue leído.

```
ReadArg ::= OBJECT IDENTIFIER;

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

En la presente Recomendación no se sustentan operaciones para ANY DEFINED BY, salvo = y /=. Esto hace imposible proporcionar una especificación de la operación Leer en SDL.

Es posible especificar la operación Leer si se conocen los diferentes tipos de atributo. En ese caso, ANY DEFINED BY se puede sustituir por un tipo CHOICE en los diferentes atributos. Supóngase que hay en total dos atributos, uno de tipo INTEGER y otro de tipo BOOLEAN. El tipo ReadResult se puede volver a escribir como sigue:

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

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

Utilizando el ReadResult redefinido, la operación Leer se puede especificar en SDL como se muestra en la Figura II.4.1.

II.4.2 Macro/clase de objeto de información OPERATION

No se admite el mecanismo de macro ASN.1 para utilización en combinación con el SDL. Tampoco se sustenta el mecanismo para la especificación de objeto de información (Recomendación X.681). Esto significa que el objeto de información OPERATION que se utiliza con frecuencia, no se puede emplear en combinación con el SLD.

En esta subcláusula se muestran dos formas de modelar una operación en SDL:

1) mediante intercambio de señales SDL;
2) mediante un procedimiento remoto SDL;

Como ejemplo, se toma una operación Leer. Su definición en ASN.1 es:

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

ReadArg y ReadResult se definen en II.4.1)

II.4.2.1 Modelado de una operación mediante intercambio de señales

La operación Leer se puede modelar mediante la introducción de tres señales diferentes:

1) ReadInvoke, con parámetro de tipo ReadArg, para modelar la invocación de la operación Leer.
2) ReadResult, con parámetro de tipo ReadResult, para modelar la devolución del resultado de la operación Leer.
3) ReadError, con parámetro de un tipo «Enumerado» para los diferentes errores de lectura para modelar el fallo de la operación Leer.

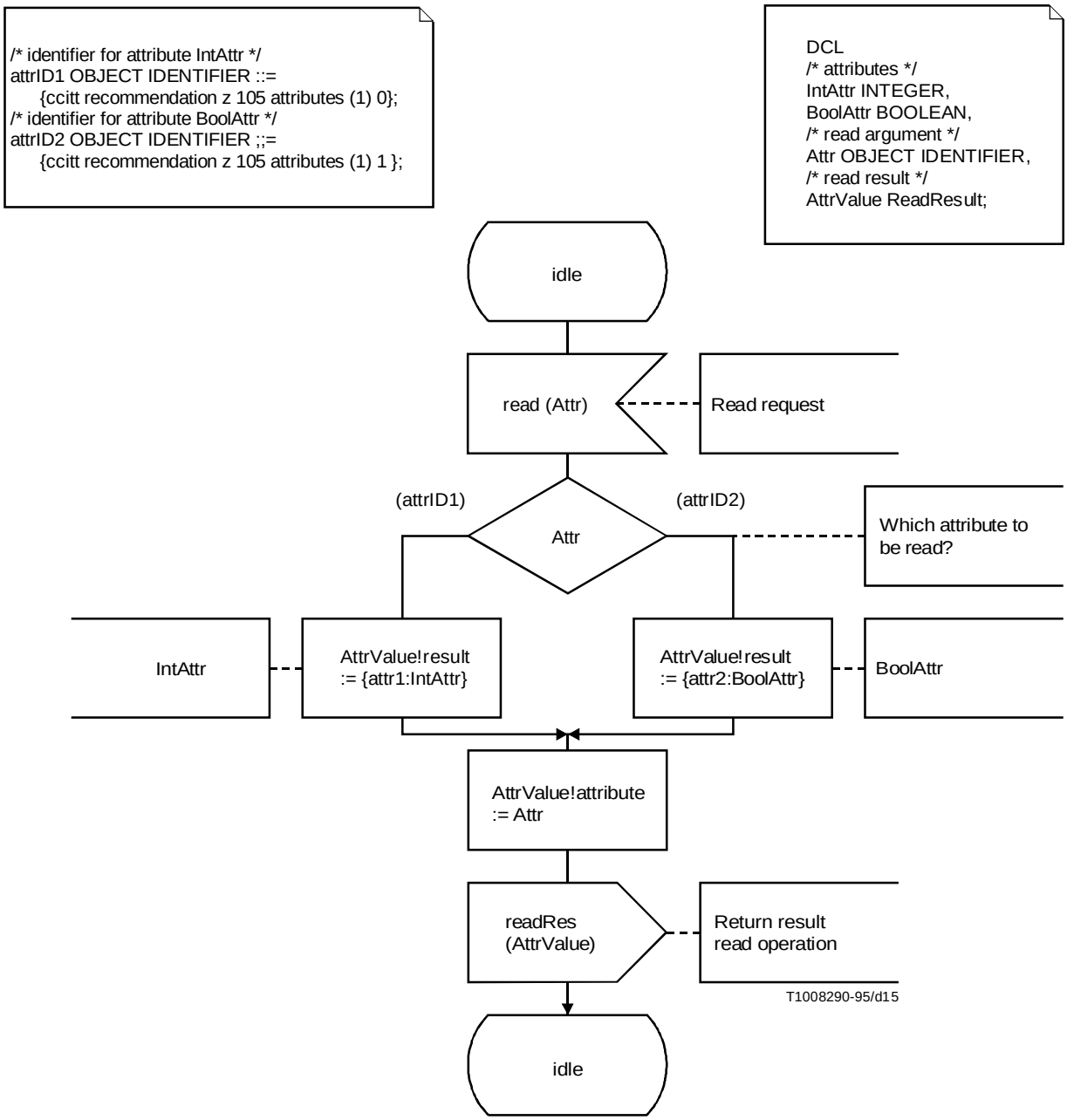

FIGURA II.4.1/Z.105

Especificación de la operación Leer

Para ReadError, es preciso definir un tipo enumerado para los diferentes errores de lectura:

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

El código de operación no tiene que estar presente en el modelo SDL: se utiliza en cambio el nombre de la señal para identificar la operación.

La operación real se puede modelar como una transición de proceso, como se muestra en la Figura II.4.2.1. El diagrama contiene texto informal. En II.4.1 se muestra cómo se puede sustituir el texto informal por SDL formal.

La operación se invoca mediante el envío de la señal ReadInvoke al proceso que puede realizar la operación, y esperar el resultado, según se ilustra en la Figura II.4.2.2

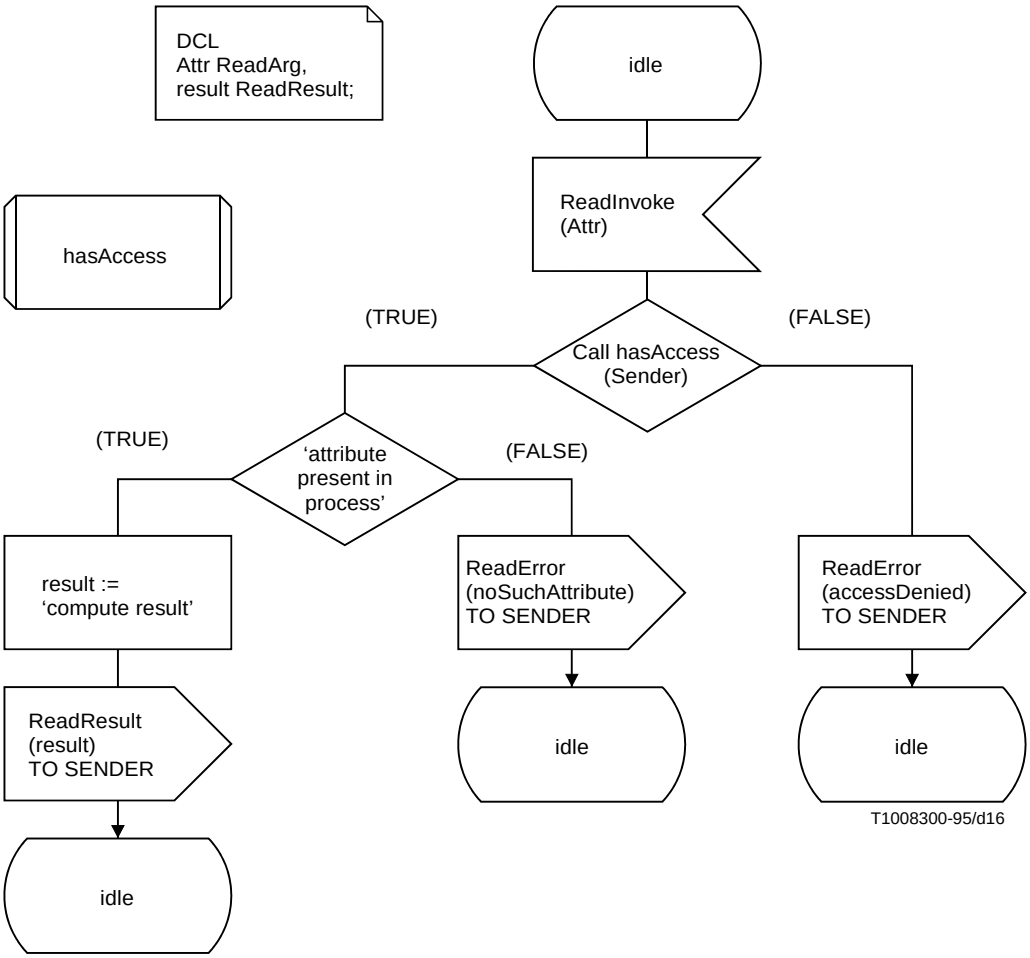

FIGURA II.4.2.1/Z.105

Una transición SDL que modela la operación Leer

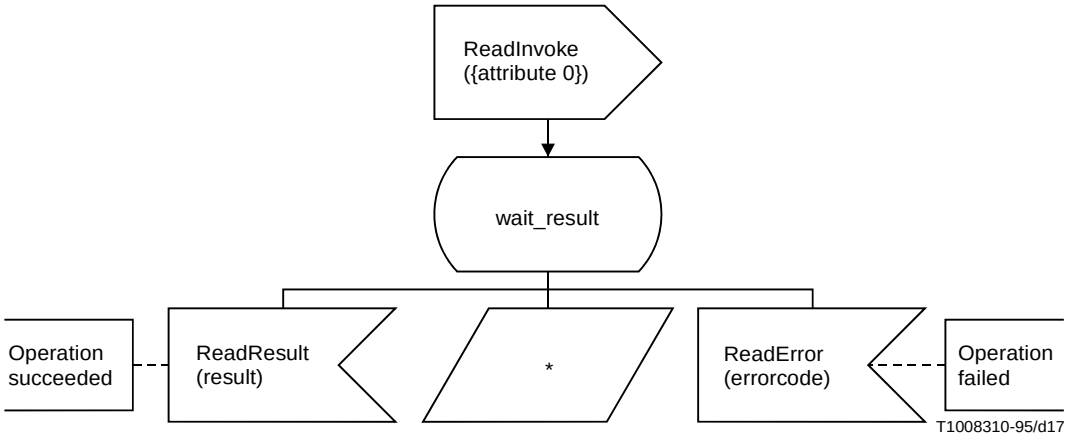

FIGURA II.4.2.2/Z.105

Invocación de la operación Leer

II.4.2.2 Modelado de una operación mediante un procedimiento remoto

Una forma alternativa consiste en introducir un procedimiento remoto SDL para la operación de lectura. El procedimiento remoto tiene dos parámetros:

- un parámetro **in** para el argumento de la operación de lectura (tipo ReadArg)
- un parámetro en **in/out** para el resultado de la operación. El resultado puede ser que la operación tenga éxito, en cuyo caso es del tipo ReadResult, o que la operación fracase, en cuyo caso es del tipo ReadError, como se define en II.4.2.1.

Para el parámetro **in/out** se introduce un nuevo tipo:

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

El procedimiento que modela la operación Leer se muestra en la Figura II.4.2.3

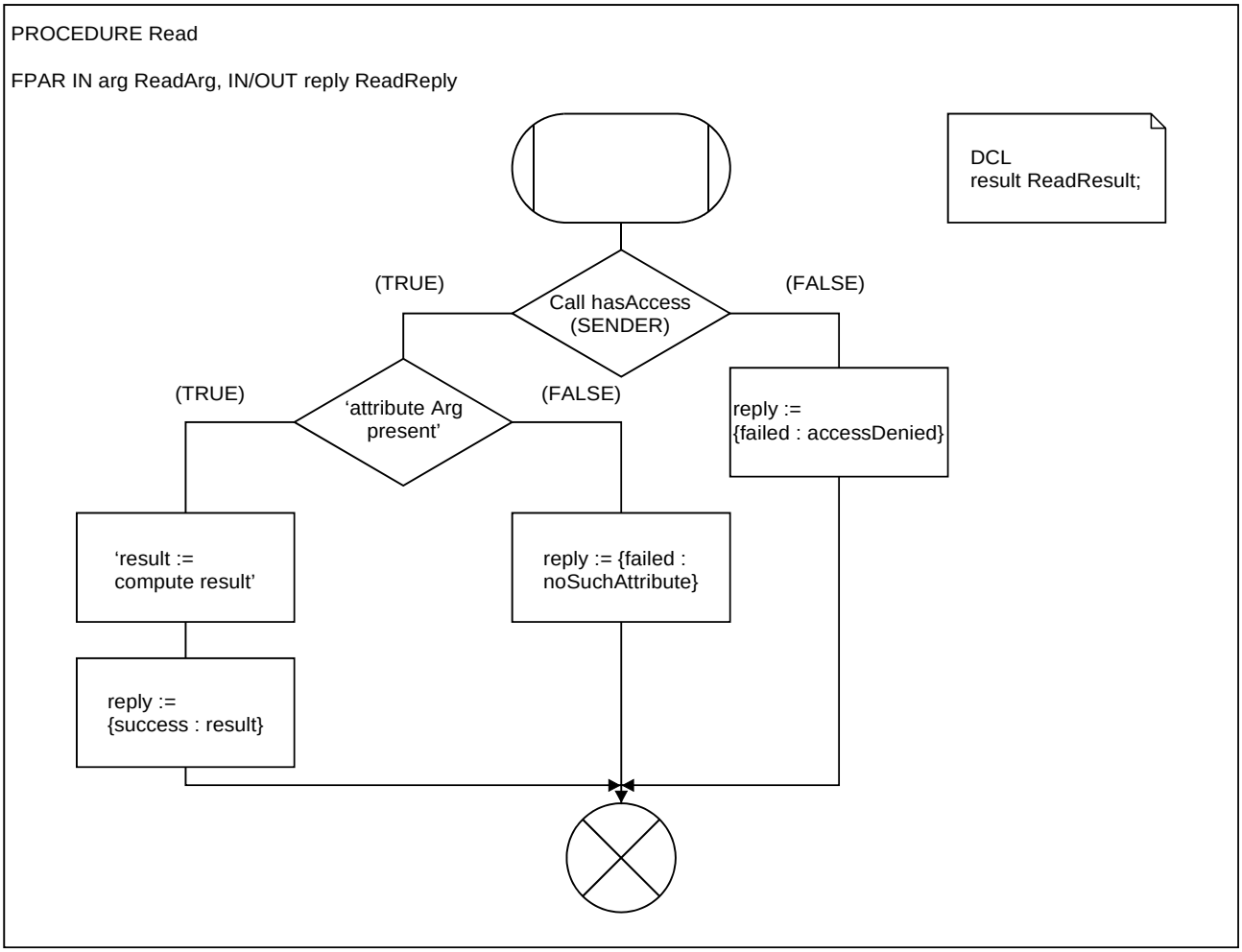

FIGURA II.4.2.3/Z.105

Procedimiento remoto modela la operación Leer

La operación se invoca llamando al procedimiento remoto Read, como se muestra en la Figura II.4.2.4. El procedimiento remoto tiene que ser importado por el proceso llamante. Esto también se muestra en la Figura.

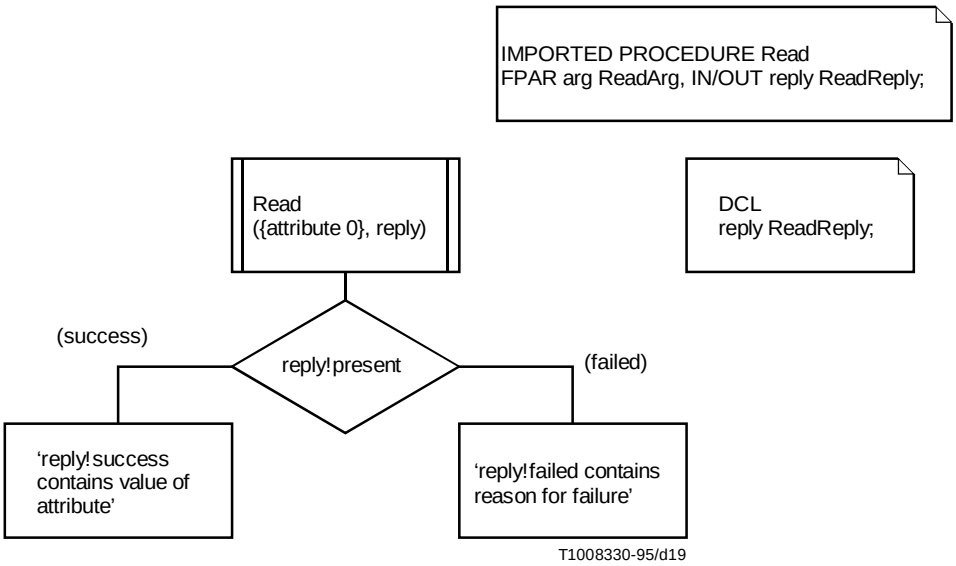

FIGURA II.4.2.4/Z.105

Invocación de la operación Leer

II.5 Algunas directrices para la utilización combinada de SDL y ASN.1

El lenguaje que se define en esta Recomendación es una mezcla de SDL y ASN.1. Como los constructivos ASN.1 corresponden con los constructivos SDL pertinentes, en principio es posible mezclar ASN.1 y SDL completamente. Es posible, por ejemplo, definir diagramas de procesos en un módulo ASN.1, y datos ASN.1 en un lote SDL. Sin embargo, no se considera que esa sea la forma adecuada de utilizar esta Recomendación.

En esta subcláusula se proporcionan algunas directrices sobre la manera de utilizar la combinación de SDL y ASN.1 tal como se prevé en esta Recomendación. El principio general es:

Utilizar la sintaxis ASN.1 para/en construcciones ASN.1 y sintaxis SDL para/en construcciones SDL

Las directrices que figuran a continuación dimanan de este principio:

- Utilizar únicamente notaciones de tipo y de valor ASN.1 en módulos. Los constructivos SDL se deben definir en lotes. De manera similar, los lotes SDL no deben contener definiciones de tipo ASN.1.

  Por ejemplo, el módulo ASN.1 WrongUse que se define más abajo contiene constructivos SDL. Aunque esta utilización es correcta de acuerdo con esta Recomendación, se debe evitar.

```
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
```

- Utilizar en la mayor medida posible la notación de valor ASN.1 para tipos definidos con la notación de tipo ASN.1, y la notación SDL para tipos definidos con la notación SDL.

  Excepción: para operadores en tipos ASN.1, se debe utilizar la sintaxis SDL, porque en la ASN.1 no hay operadores.

  Por ejemplo:

  – Para S ::= SEQUENCE { a INTEGER} : utilizar { a 5} como notación de valor, y no (. 5 .)
  – Para **newtype** s **struct** a : integer **endnewtype**: utilizar (. 5 .) como notación de valor, y no { a 5}
  – Para IA5String: utilizar "abc" como notación de valor, no 'abc'
  – Para Charstring:utilizar 'abc' como notación de valor, no "abc"
  – BIT STRING: es correcto utilizar '10'B // '0A3C'H, porque la // (concatenación) es un operador que no está disponible en ASN.1.

- Se recomienda observar la sensibilidad a los casos de ASN.1, aunque la sensibilidad a los casos no se sustenta en la presente Recomendación.

  Por ejemplo:

– No utilizar c ::= Choice { A Integer, B Boolean },

  y utilizar en cambio C ::= CHOICE { a INTEGER, b BOOLEAN }

- Evitar en la mayor medida posible la utilización de tipos específicos de SDL dentro de la definición de tipos ASN.1, y utilizar también tipos específicos de ASN.1 dentro de la definición de tipos de SDL. Tratar de evitar tipos que mezclan tipos específicos de SDL con tipos específicos de ASN.1. Ejemplos de tipos específicos de SDL son PId, Time, Duration, Character. Tipos específicos de ASN.1 son, por ejemplo, NULL, ANY y BIT STRING.

  Por ejemplo:

  – No utilizar S ::= SEQUENCE { p PId, t Time }

    pero utilizar en cambio **newtype** S **struct** p: PId; t : Time **endnewtype**
  – No utilizar newtype S struct a : ANY; n: NULL endnewtype,

    pero utilizar en cambio S ::= SEQUENCE { a ANY, n NULL }
  – Tratar de evitar S ::= SEQUENCE { n Null, p PId }, aunque esto no puede ser posible.

Apéndice III

Operadores admitidos para tipos ASN.1

(Este apéndice no es parte integrante de esta Recomendación)

Este apéndice proporciona una visión de conjunto de los tipos ASN.1 admitidos, y de los operadores disponibles en el SDL para dichos tipos.

Para cada tipo, están presentes por lo menos dos operadores: = (igualdad) y /= (desigualdad).

Any

Operators:

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

Bit string

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

Some operators are illustrated using type:

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

Operators:

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

Length(...),

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

Mkstring(...),

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

... // ...,

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

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

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

...(...),

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

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

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

Boolean

Operators:

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

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

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

Character string types

The character strings are strings of the SDL type Char.

Operators:

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

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

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

Num (...)

/* returns canonical number of a character */

Choice

The examples are based on:

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

Operators:

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

...!<identifier>,

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

<identifier>Present,

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

...!present,

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

... < ...

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

Enumerated

The examples are based on:

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

Operators:

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

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

num (...),

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

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

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

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

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

Integer

Operators:

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

/* comparison operators */

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

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

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

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

Null

Operators:

… = …, … /= …

Object identifier

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

Operators on OBJECT IDENTIFIER:

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

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

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

Octet string

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

Operators for OCTET STRING:

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

Length(...),

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

Mkstring(...),

/* OCTET to OCTET STRING conversion */

... // ...,

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

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

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

...(...),

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

BIT STRING (...),

/* OCTET STRING to BIT STRING conversion */

OCTET STRING (...)

/* BIT STRING to OCTET STRING conversion */

Real

Operators:

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

/* similar as INTEGER operators */

Power(..., ...),

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

float(...),

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

fix(...)

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

Sequence

The examples are based on:

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

Operators:

```
... = ..., ... /= ...,
...!<identifier>,
    /*  gives the value of one component, e.g. s!field1 = 105,
        s!field2 = 7, s!field3 results in a dynamic error */
<identifier>Present(...)
    /*  for optional components: indicates whether the identifier is
        present or not: field3Present(s) = FALSE. For default components,
        present always gives TRUE, i.e. field2Present(s) = TRUE */
```

Sequence of

Operators:

```
... = ..., ... /= ...,
Length(...),  Mkstring(...), ... // ..., SubString(..., ..., ...), ...(...)
    /*  operators are similar to those of BIT STRING, but indexing starts with 1 instead of 0 */
```

Set

See SEQUENCE.

Set of

The examples are based on SET OF INTEGER

Operators:

```
... = ..., ... /= ..., ... < ... , ... > ..., ... < = ..., ... > = ...,
    /*  comparison operators. < is subset of, > superset operator,
        e.g. {3, 3} /= {3},          {3, 3} > {3},       {1, 2} = {2, 1} */
... and ..., ... or ...,
    /*  union and intersection of sets, e.g. {1} or {0, 1} = {1, 0},
        {1, 1} and {4, 1} = {1} */
... in ...,
    /*  is element in set, e.g. 7 in {4, 7} = TRUE, 7 in {4, 8} = FALSE */
Makebag (...),
    /*  makes a set of one element, e.g. Makebag (1) = {1} */
Incl (..., ...)
    /*  add element to set, e.g. Incl (5, {2, 3}) = {5, 2, 3} */
```

Del (..., ...)

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

Length (...)

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

Take (...)

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

Subtypes

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

Tagged types

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

Useful types

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

Apéndice IV

Resumen de la sintaxis

(Este apéndice no es parte integrante de esta Recomendación)

A continuación se enumeran las producciones de sintaxis que en esta Recomendación son diferentes de las correspondientes producciones de sintaxis de la Recomendación Z.100. Las producciones indicadas, pero no definidas, son idénticas a las de la Recomendación Z.100. Todas las reglas sintácticas enumeradas provienen del cambio de las siguientes reglas de producciones sintácticas de la Recomendación 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>}

ÍNDICE

Nota de la TSB – La traducción del índice de esta Recomendación Z.105 no se aplica. Favor de referirse a la versión inglesa.

_____