UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

CCITT

COMITÉ CONSULTIVO
INTERNACIONAL
TELEGRÁFICO Y TELEFÓNICO

Z.100 Anexo F1

(11/1988)

ANEXO F1 A LA RECOMENDATION Z.100: DEFINICIÓN FORMAL DEL LED

INTRODUCCIÓN

DEFINICIÓN FORMAL DEL LED

Reedición de la Recomendación Z.100 Anexo F1 del CCITT publicada en el Libro Azul, Fascículo X.3 (1988)

NOTAS

1 La Recomendación Z.100 Anexo F1 del CCITT se publicó en el fascículo X.3.4 del Libro Azul. Este fichero es un extracto del Libro Azul. Aunque la presentación y disposición del texto son ligeramente diferentes de la versión del Libro Azul, el contenido del fichero es idéntico a la citada versión y los derechos de autor siguen siendo los mismos (véase a continuación).

2 Por razones de concisión, el término «Administración» se utiliza en la presente Recomendación para designar a una administración de telecomunicaciones y a una empresa de explotación reconocida.

ÍNDICE DEL FASCÍCULO X.3 DEL LIBRO AZUL

Anexo F.1 a la Recomendación Z.100

_____

NOTAS PRELIMINARES

1 Las Cuestiones asignadas a cada Comisión de Estudio para el periodo de estudios 1989-1992 figuran en la contribución N.° 1 de dicha Comisión.

2 En este fascículo, la expresión «Administración» se utiliza para designar, en forma abreviada, tanto una Administración de telecomunicaciones como una empresa privada de explotación de telecomunicaciones reconocida.

FASCÍCULO X.3

Anexo F.1 a la Recomendación Z.100

DEFINICIÓN FORMAL DEL LED

ANEXO F.1 A LA RECOMENDACIÓN Z.100

DEFINICIÓN FORMAL DEL LED

ÍNDICE

1 Prefacio

La presente definición formal del LED proporciona una definición del lenguaje que complementa la definición del texto de la Recomendación. Este anexo está destinado a aquéllos que necesitan una definición muy detallada y precisa del LED, como por ejemplo el personal de mantenimiento del lenguaje LED y los diseñadores de instrumentos LED.

La definición formal está constituida por tres volúmenes:

Anexo F.1 (este volumen)

Establece la motivación, describe la estructura general, facilita directrices sobre cómo utilizar la definición formal y describe la notación utilizada.

Anexo F.2 Define las propiedades estáticas del LED.

Anexo F.3 Define las propiedades dinámicas del LED.

2 Motivación

En general, los lenguajes naturales son ambiguos e incompletos, es decir que se puede dar más de una interpretación a algunas de las sentencias del lenguaje, ya sea un ser humano o un computador quien haga la lectura.

Una definición o especificación es formal si su sentido (semántica) es completo y sin ambigüedad. Debido a que los lenguajes naturales no pueden utilizarse para este fin, se han desarrollado lenguajes especiales, denominados lenguajes de especificación (como LED y LOTOS). Lenguajes de implementación como CHILL o PASCAL pueden también utilizarse como lenguajes de especificación (por ejemplo, un compilador especifica formalmente la semántica de otro lenguaje), pero a menudo es esencial separar los detalles de la realización práctica, irrelevantes para la comprensión, de la semántica de una especificación.

Los lenguajes formales especialmente adecuados para la definición de lenguajes se denominan metalenguajes. Por ejemplo la forma Backus Naur (FBN) es un metalenguaje especialmente adecuado para definir formalmente la sintaxis de lenguajes de programación.

A pesar de la ambigüedad de los lenguajes naturales, éstos son normalmente más legibles para los seres humanos que los lenguajes formales y permiten describir más fácilmente el marco en el que se puede entender la especificación. Por dichas razones, a menudo se dan las definiciones en un lenguaje natural y en un lenguaje formal.

Este anexo es una definición formal del LED. Si cualquiera de las propiedades de un concepto LED aquí definido contradice las propiedades definidas en la Recomendación Z.100, y el concepto está definido consistentemente en dicha Recomendación, la definición dada en ella tiene preferencia y este documento debe ser corregido.

2.1 *El metalenguaje*

El metalenguaje empleado en esta definición formal es el Meta IV [1]. Las razones para elegir este lenguaje son las siguientes:

- Está construido sobre bases matemáticas sólidas y ampliamente estudiadas.
- Tiene facilidades muy adecuadas y poderosas para la manipulación de objetos.
- Tiene una notación de tipo «programación», lo cual significa que está orientado a programadores y diseñadores.
- Está en curso de normalización por parte de la Comunidad Europea.
- Hay gran cantidad de información sobre el mismo en libros, artículos y revistas científicas, habiéndose utilizado en el manual del CCITT «The Formal Definition of CHILL» [2], que también contiene un resumen de la notación del Meta IV.
- Se dispone de instrumentos del Meta IV que permiten verificar la sintaxis, analizar la visibilidad, generar documentos, efectuar referencias cruzadas, etc.

En la sección 5 puede encontrarse una introducción informal a las partes del Meta IV utilizadas en la definición formal. En [1] puede encontrarse una definición completa del Meta IV.

3 Técnica de modelado

Al considerar el significado de «semántica del LED» es conveniente (conceptualmente) descomponer la definición del lenguaje en varias partes:

- La definición de las reglas sintácticas.
- La definición de las reglas semánticas estáticas (denominadas condiciones de formación correcta) tales como los nombres que se permiten utilizar en un determinado lugar, el tipo de valores que es posible asignar a variables, etc.
- La definición de la semántica de las construcciones del lenguaje cuando son interpretadas (semántica dinámica).

No es necesario incluir las reglas sintácticas en definición formal, ya que las reglas FBN y los diagramas sintácticos de la Recomendación Z.100 sirven como definiciones formales de las reglas de sintaxis, lo que significa que la entrada a la definición formal es una especificación LED sintácticamente correcta. La entrada se representa mediante una sintaxis abstracta. Esta sintaxis abstracta se basa en la sintaxis LED textual concreta de análisis arbóreo (reglas FBN) eliminando detalles irrelevantes como separadores y reglas léxicas. Por lo tanto, esta sintaxis abstracta no es la de la Recomendación Z.100 que aparece en las Recomendaciones y que es una abstracción del concepto del modelo LED.

Por ejemplo, la regla de producción en sintaxis abstracta:

1 *Transstring* :: *Actstmt*$^+$ [*Termstmt*]

expresa que una *trans*ition *string* consiste en una lista no vacía de *act*ion *state*ment*s* y de un *term*inator *state*ment opcional (las letras que están en cursiva también lo están en la regla de producción). El conjunto completo de reglas de producción (denominadas definiciones de dominio) que define la sintaxis LED de manera abstracta se denomina AS_0. En cierto modo define la sintaxis del lenguaje a un nivel más básico que las reglas de sintaxis de la Recomendación Z.100, ya que la sintaxis textual concreta de ésta contiene, a diferencia de AS_0, mucha información semántica (es sensible al contexto). Debe señalarse que AS_0 es una abstracción de la sintaxis textual concreta. Por razones de economía de tiempo y espacio, más que por la dificultad del trabajo en sí, no se ha utilizado la sintaxis gráfica concreta.

Por ejemplo, en la Recomendación Z.100, se define una lista de señales como:

<signal list> ::= <signal item> {,<signal item>}
<signal item> ::= <signal identifier> | (<signal list identifier>) | <timer identifier>

mientras que las definiciones correspondientes en AS_0 son:

2 *Signallist* :: *Signalitem*$^+$
3 *Signalitem* = *Id* | *Signallistid*

Una *signallist* consiste en una lista de *signalitem*s. Un *signalitem* es un identificador o un identificador de lista de señales. En contraposición a la producción de <ítem de señal> mediante FBN, que es sensible al contexto, en AS_0 no se hace distinción entre un identificador de señal y un identificador de temporizador, ya que sintácticamente ambos son identificadores, a diferencia de las listas de señales, las cuales se distinguen por el uso del paréntesis.

El punto de partida de la definición formal (DF) consiste en especificaciones LED sintácticamente correctas. La misión de la definición formal consiste en:

- Definir las condiciones de formación correcta para las especificaciones LED. Esta función, denominada semántica estática, constituye el anexo F.2,
- Definir las propiedades dinámicas para las especificaciones LED. Esta función, denominada semántica dinámica, constituye el anexo F.3.

En la figura 1 se muestran los pasos indicados. A continución se explica el resultado de la semántica estática (es decir, AS_1).

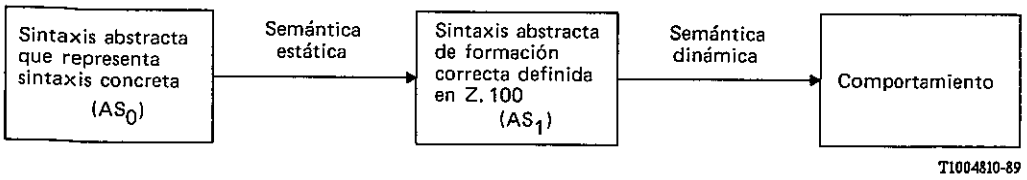

FIGURA 1

Objectivos de la semántica estática y la semántica dinámica

El paso de traducción de la sintaxis textual concreta a AS_0 no está formalmente definida, pero se obtiene de la correspondencia entre los nombres de las dos sintaxis, tal como se ha explicado antes para *Signallist*.

3.1 *Semántica estática*

En la Recomendación Z.100, la semántica dinámica de las distintas construcciones se define en términos de una sintaxis abstracta. Las subsecciones comunes, *gramática textual concreta* y *gramática gráfica concreta*, definen las reglas de la sintaxis concreta, establecen las condiciones de formación correcta adecuadas y relacionan las reglas de la sintaxis concreta con la sintaxis abstracta de Z.100. Se define utilizando el Meta IV (en las subsecciones comunes de *gramática abstracta*). La misma sintaxis abstracta se utiliza en la definición formal (en la cual se denomina AS_1). En el anexo B de la Recomendación Z.100 figura un resumen de esta sintaxis abstracta.

Además de definir las condiciones de formación correcta, la semántica estática debe, por tanto, definir cómo la representación AS_0 de una especificación se transforma en la representación AS_1, es decir, dada una representación AS_0, la semántica estática devuelve una representación AS_1 si la representación AS_0 está formada correctamente. La semántica estática puede considerarse como un «compilador abstracto» en el cual la representación AS_0 es el lenguaje fuente y la representación AS_1 es el lenguaje objeto.

Además de AS_0 y AS_1, la semántica estática utiliza algunos dominios de servicio interno, conocidos como dominios semánticos, que tienen la información requerida sobre una entidad dada en cualquier lugar. Por ejemplo, cuando se transforma una definición de proceso, la información sobre sus parámetros formales se conserva en los dominios semánticos y se recupera durante la transformación de la acción de petición de crear. Los dominios AS_0 podían utilizarse para este propósito, ya que los dominios semánticos se deducen de AS_0, pero una representación en árbol no es útil cuando se requiere información de una entidad determinada (por ejemplo, una definición de proceso). Por lo tanto, los dominios semánticos son normalmente relaciones de correspondencia (aplicaciones) que modelan tablas.

Por ejemplo, los dominios semánticos incluyen una aplicación (se explica más adelante, en el § 5.4.7) de identificadores en descriptores que incluyen información acerca de los identificadores:

```
4  Descriptordict        = Qual m→Descr
```

donde *Qual* es la representación del identificador usado internamente en la definición formal y *Descr* es cualquier descriptor. El descriptor puede serlo, por ejemplo, de un proceso:

```
5  Descr                 = ProcessD | ...
6  ProcessD              :: ParameterD* Validinputset Outputset
```

expresando que un *Process D*escriptor contiene una lista de *Parameter D*escriptors, información sobre el *set* de señales *Valid input* e información sobre las señales *Output*. No se muestran aquí las definiciones de estos tres (sub)descriptores.

La transformación en sí misma se realiza mediante un conjunto de funciones del Meta IV que utilizan los tres dominios AS_0, AS_1 y semánticos.

3.2 *Semántica dinámica*

La misión de la semántica dinámica es definir el comportamiento de una especificación LED en la forma AS_1.

La semántica dinámica se divide en tres partes principales:

– El modelo del sistema subyacente (la máquina LED abstracta).

– La interpretación de los gráficos de proceso.

– La transformación de AS_1 en un modelo más apropiado; es decir se contruye una aplicación (un dominio semántico) que contiene la información requerida durante la interpretación, tal como el género de las variables, los posibles trayectos de comunicación entre procesos, las clases de equivalencia para tipos, etc. La aplicación se denomina *Entity-dict* (o más correctamente el dominio de la aplicación se denomina *Entity-dict*).

En la semántica dinámica, la concurrencia en LED se modela utilizando **metaprocesos**; esto se realiza ejecutando concurrentemente metaprocesos en el modelo Meta IV que ejecuta concurrentemente procesos en LED.

Se utilizan seis tipos distintos de metaprocesos:

– *system*

Para tratar el encaminamiento de la señal y la creación de *sdl-process*os.

– *path*

Para tratar el retardo no determinístico de los canales.

– *timer*

Para tener en cuenta el tiempo (la hora) actual y manejar temporizaciones.

– *view*

Para ocuparse de todas las variables reveladas.

– *sdl-process*

Para interpretar el comportamiento de un proceso LED.

– *input-port*

Manejar la cola de señales en un proceso LED. Para cada instancia de *sdl-process* existe una instancia de *input-port*.

Puede considerarse que los cuatro tipos de metaprocesos, *system, path, timer* y *view* en su conjunto, modelan el sistema subyacente.

No se comparten datos entre metaprocesos; éstos interaccionan transmitiendo valores mediante instancias (objetos) de **dominios de comunicación** (corresponden al concepto de señales en LED).

Los dominios de comunicación se definen de la misma forma que otros dominios; por ejemplo, los objetos del dominio de comunicación *Input-Signal* se dirigen hacia una instancia de *sld-process* a partir de la instancia de *input-port* asociada. El dominio de comunicación se define de la manera siguiente:

7 *Input-Signal* :: *Signal-Identifier*$_1$ [*Value*]* *Sender-Value*

Las instancias de *Input-Signal* transportan el identificador de la señal LED enviada, la lista de valores transportados por la señal LED y el valor PID del emisor.

La figura 2 incluye el «esquema de interacción de metaprocesos» completo. El mecanismo de comunicación es síncrono y la notación se conoce como CSP (véase [3] y [4]) (Communicating Sequential Processes).

3.3 *Ejemplo*

La figura 3 muestra la comunicación entre metaprocesos en la definición formal para el siguiente proceso (parcial) LED, cuando llega una señal («b») del entorno y el proceso responde devolviendo una señal («a») al entorno:

```
. . .
state S;
  input b;
  output a;
. . .
```

La comunicación se ilustra de manera informal mediante un diagrama de secuencia de mensajes. Path(1) y Path(2) son dos instancias del procesador path, correspondientes al trayecto desde el entorno al proceso LED (Path(1)) y viceversa (Path(2)).

3.4 *Estructura física de la definición formal*

La semántica estática (anexo F.2) se divide en tres partes principales:

1. Las definiciones de dominio para AS_0
2. Las definiciones de dominio para los dominios semánticos
3. Las funciones Meta IV que comprueban las condiciones de formación correcta y definen cómo AS_0 se transforma en AS_1

Las definiciones de dominios para AS_1 que se utilizan en las partes 2 y 3 se encuentran en la Recomendación Z.100 y están resumidas en el anexo B de la Recomendación Z.100. No se repiten en la definición formal. El anexo F.2 también incluye índices cruzados de los nombres de funciones y dominios Meta IV (que definen tanto las ocurrencias como las ocurrencias aplicadas) y un índice cruzado de las condiciones de formación correcta aplicadas.

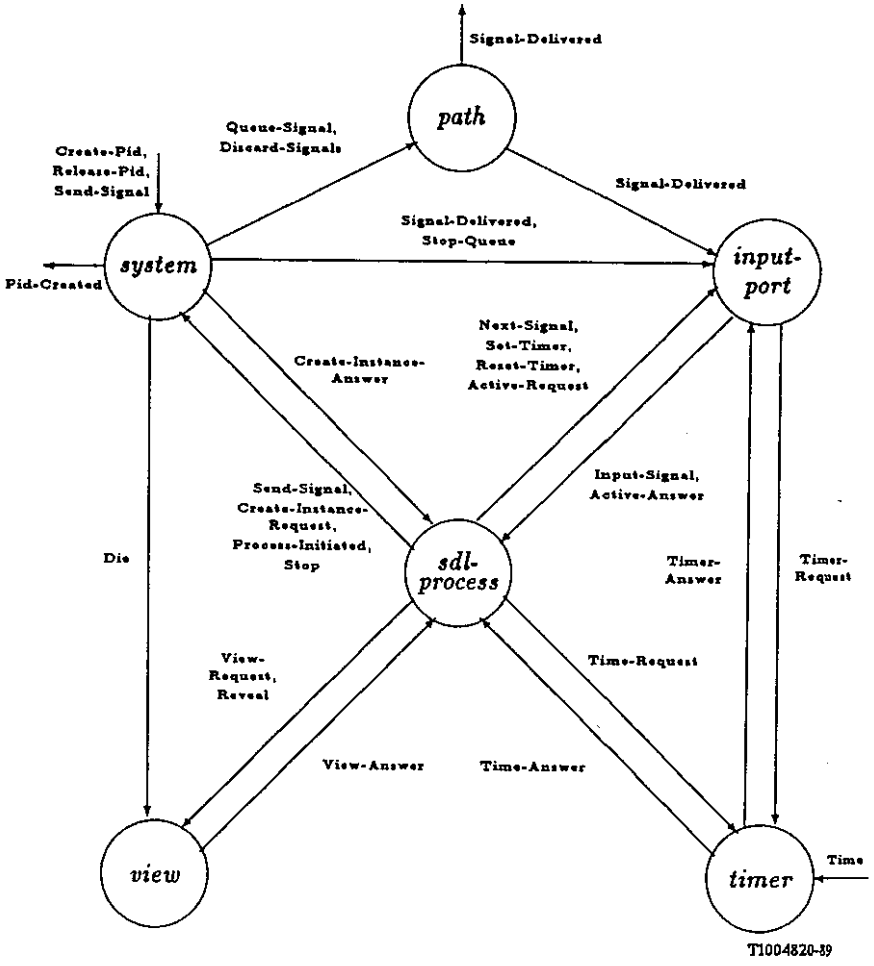

FIGURA 2

Esquema de comunicación

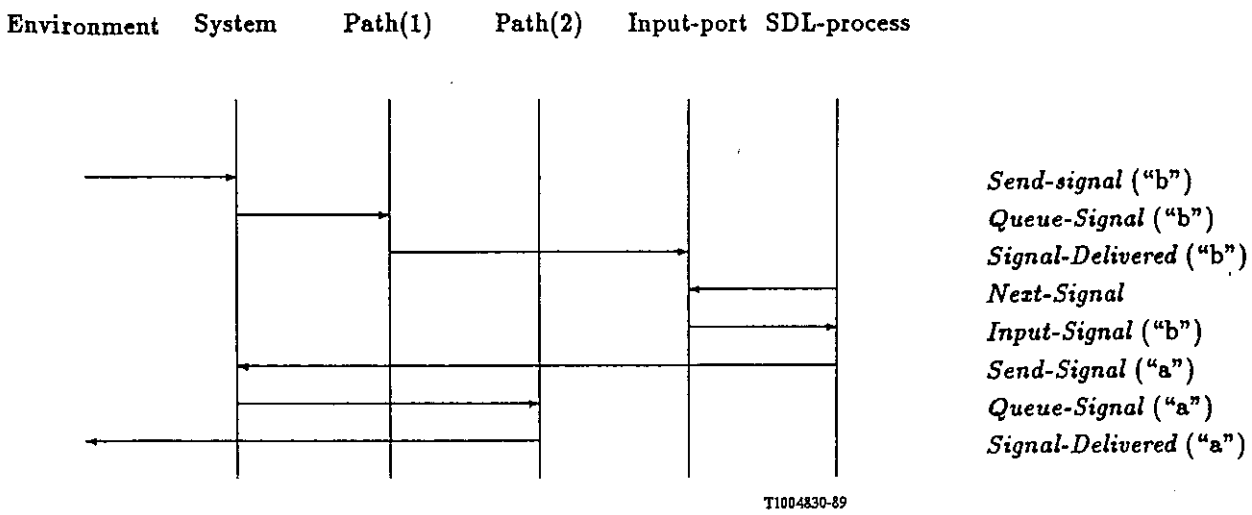

FIGURA 3

Ejemplo de comunicación entre metaprocesos

La semántica dinámica (anexo F.3) se divide en cinco secciones principales:

1. Definiciones de dominio para los dominios de comunicación
2. Definiciones de dominio para los dominios semánticos (*Entity-dict*)
3. Las definiciones de metaprocesos y las funciones asociadas para el modelo del sistema subyacente
4. Las definiciones de metaprocesos y las funciones asociadas para la interpretación del proceso LED
5. La creación del dominio interno *Entity-dict*. Los procesos LED utilizan *Entity-dict*, que por lo tanto se crea antes de que se interpreten los procesos LED.

El anexo F.3, como el anexo F.2, contiene asimismo una serie de índices relativos a los nombres de dominios, nombres de funciones, nombres de metaprocesos, condiciones de error, etc.

Si bien el número de páginas puede aparecer enorme a primera vista (sobre todo en el anexo F.2), más de la mitad del espacio contiene anotaciones para definiciones de dominios, funciones y procesos. La disposición de una definición de función o de proceso sigue el siguiente esquema:

1. En primer lugar, se especifica la definición de la función o del proceso mediante:
   a) un encabezamiento que define el nombre del proceso o de la función y los nombres de sus parámetros formales,
   b) el cuerpo (algoritmo),
   c) una *cláusula de tipo* que especifica el tipo (dominio) de los parámetros formales y el tipo del resultado (si es que lo hay).
2. A continuación, vienen las anotaciones (en lenguaje natural), por conceptos, asociadas a la definición de proceso o función:

Objective	explica el propósito de la función o proceso;
Parameters	explica el propósito de cada parámetro formal de la función o proceso;
Result	explica el objeto devuelto (si lo hay);
Algorithm	explica, línea a línea, el algoritmo utilizado en la función o proceso.

Ejemplo

La función *definition-of-SDL* más externa del anexo F.2 que enlaza la semántica estática (*transform-system*) con la semántica dinámica (comenzando por el *system* del metaproceso) es como sigue:

```
definition-of-SDL(extparms, systemdef, predefsorts) ≜
1  (let (as1, auxinf) = transform-system(systemdef, predefsorts, extparms) in
2   if as1 = nil then
3     undefined
4    else
5     (let subsetcut = select-consistent-subset(as1, extparms) in
6      start system(as1, subsetcut, auxinf)))

type: External-Information Sys0 Datadef0+ ⇒
```

Objective Define las propiedades del LED.

Parameters

extparms Alguna *External-Information* (véase el anexo F.2, § 2.3).

systemdef El árbol AS_0 representa el sistema LED.

predefsorts Los datos predefinidos en la forma AS_0.

Algorithm

línea 1: Transforma el sistema en la forma de sintaxis abstracta (forma AS_1).

línea 2: Si se encuentran errores estáticos (es decir, si no se obtiene representación AS_1) el comportamiento no está definido.

línea 4: Si no se encuentran errores estáticos, entonces

línea 5: Seleccionar el conjunto de *Block-identifier*$_1$s que indican el subconjunto consistente.

línea 6: Crear una instancia del sistema, es decir, crear un proceso Meta IV que se comporte como el sistema subyacente.

4 Cómo utilizar la definición formal

4.1 *Usuarios del LED*

La definición formal no pretende ser un manual de referencia para usuarios del LED. Los usuarios principiantes del LED pueden encontrar adecuadas las Directrices para el Usuario (anexo D de la Recomendación Z.100) para tener una visión general de los conceptos del lenguaje, mientras que la Recomendación Z.100 por sí misma sirve como manual de referencia sobre el LED, pudiendo, no obstante, haber algunos casos en los que la Recomendación Z.100 resulte inadecuada. Por ejemplo:

- si faltan algunas propiedades (por ejemplo, alguna condición estática esperada), si algunas propiedades enunciadas contradicen otras propiedades, o
- si el significado exacto de algunas propiedades enunciadas es difícil de entender, o
- si algunas propiedades son difíciles de encontrar (debido a que en la Recomendación Z.100 no hay un índice cruzado), o
- si el usuario desea tener un conocimiento más profundo de materias más complejas como, por ejemplo, la máquina LED abstracta, cuándo y cómo seleccionar un subconjunto consistente, la resolución por el contexto, el mecanismo de herencia, etc.

En dichos casos la definición formal puede ser un documento de apoyo útil. El usuario debe, en primer lugar, conocer internamente la estructura de la definición formal, cómo se organizan las funciones y para qué se utilizan los dominios. También se requiere un cierto conocimiento sobre la notación del Meta IV, pero al estar las funciones ampliamente documentadas , puede resultar posible leer el Meta IV leyendo las funciones junto con las anotaciones después de haber leido la introducción al Meta IV (§ 5 siguiente). Al estudiar la definición formal los usuarios pueden utilizar ventajosamente el índice de materias y los índices cruzados.

4.2 *Implementadores*

Como se ha indicado antes, el enfoque del Meta IV permite a los implementadores obtener sistemáticamente una realización (es decir, un analizador estático, simulador, etc.) a partir de la especificación Meta IV. Es posible obtener un analizador estático y un simulador para el LED a partir de los anexos F.2 y F.3 respectivamente. Es aconsejable utilizar la representación AS_1 (generada por el analizador estático) como base para la simulación. Las razones para ello son que en AS_0 falta información de contexto para los identificadores (normalmente éstos no se califican en AS_0) y que la semántica dinámica de una especificación en la forma AS_0 es difícil de obtener debido al gran número de notaciones taquigráficas del LED (especialmente para conceptos como tipos de datos).

Debe señalarse que la obtención de una realización práctica es sistemática pero no automática.

Deben considerarse los siguientes puntos:

– Deben encontrarse tipos de datos adecuados para representar los tipos de datos ideales (dominios) del Meta IV tales como aplicaciones (relaciones de correspondencia), listas y conjuntos utilizados en AS_0, AS_1 y los dominios semánticos.

– Debido a las reglas de visibilidad del LED (el hecho de que los identificadores puedan utilizarse antes de ser definidos), se utilizan (por conveniencia) las denominadas «ecuaciones de coma fija» en la semántica estática (véase el § 3.1 del anexo F.2). En una relación práctica los dominios semánticos pueden crearse gradualmente pasando varias veces por el árbol AS_0 (por ejemplo, los descriptores para señales deben crearse antes de crear descriptores para canales ya que los canales hacen referencia a las señales en sus definiciones).

– El enfoque del álgebra inicial implica que la definición formal manipula objetos infinitos. También AS_1 contiene objetos infinitos. Es por tanto necesario modificar ligeramente AS_1 e imponer restricciones al uso de tipos de datos o utilizar algunas técnicas de abstracción en la que sea posible codificar dichos objetos.

5 Introducción al Meta IV

Esta sección presenta una introducción informal al Meta IV y sobre cómo se ha utilizado éste en la definición formal, es decir, se explica el Meta IV en términos de la definición formal (abreviadamente DF) lo cual significa que sólo se explican aquellas partes del Meta IV utilizadas en la DF.

5.1 *Estructura general*

La DF consiste en:

– Un conjunto de definiciones de funciones y procesos que definen la semántica del LED. Los procesos (en el Meta IV y en la DF se denominan procesadores) se utilizan para modelar la concurrencia y por lo tanto sólo se utilizan en la semántica dinámica. Sintácticamente, las definiciones de procesadores se parecen a las definiciones de funciones (excepto por la palabra clave **processor** que sigue al nombre del procesador); por tanto, la siguiente descripción del concepto de función se aplica también a los procesadores.

– Un conjunto de definiciones de dominios que definen el tipo de los objetos manipulados por las funciones. Se introducen términos que denotan ciertos grupos de definiciones de dominios a fin de clasificarlos lógicamente. Los dominios AS_0 denotan la representación de la sintaxis concreta, los dominios AS_1 denotan la sintaxis abstracta del LED y los conjuntos de dominios *Dict y Entity-Dict* denotan los dominios de servicios internos (dominios semánticos) de las semánticas estática y dinámica respectivamente. En esta sección se usará a menudo «valor» como sinónimo de objeto y «tipo» como sinónimo de dominio.

– Un conjunto de definiciones de constantes globales. En la DF sólo hay dos de dichas definiciones. Se definen en el § 3.13 de la semántica estática. No son esenciales para la comprensión de la DF.

Las definiciones pueden especificarse en cualquier orden y los nombres introducidos en las mismas pueden usarse antes de ser definidos textualmente.

5.2 *Definición de funciones*

Una definición de función consta de tres partes:

1) El encabezamiento, que comienza con el nombre de la función y va seguido por una o dos listas de parámetros formales. Cada lista de parámetros formales se encierra entre paréntesis. La división de los parámetros en dos listas carece de importancia formal. A menudo algunos parámetros aparecen en una (segunda) lista de parámetros separada si es que son de importancia secundaria para la evaluación, como por ejemplo en el caso de los dominios semánticos que frecuentemente son utilizados por las funciones y se suministran en una lista de parámetros separada.

2) El cuerpo de la función, que puede ser una expresión o una secuencia de sentencias. Una función no tiene que producir ningún resultado (véase más adelante).

3) La cláusula de tipo, que especifica el tipo de los parámetros formales y el tipo del resultado. Primero se especifica el tipo de la primera lista de parámetros, a continuación el tipo de la segunda lista de parámetros (si la hay) separada de la primera por una flecha (→ o =>), a continuación otra flecha y después el resultado.

Ejemplo

$f(a, b)(d) \triangleq$

1 /* *expression* */

type: $DomX\ DomY \rightarrow DomZ \rightarrow DomW$

En este ejemplo se tiene

f	nombre de la función
a, *b*, *d*	parámetros formales. *a* y *b* están en la primera lista de parámetros formales y *d* en la segunda. El tipo de *a* es *DomX*, el de *b* es *DomY* y el de *d* es *DomZ* . El tipo del resultado es *DomW*. Los dominios *DomX*, *DomY*, *DomZ* y *DomW* deben definirse en definiciones de dominios.

Si los parámetros formales o el resultado no se utilizan de acuerdo con la cláusula de tipo, hay un error en la especificación Meta IV. En el ejemplo anterior se utiliza texto Meta IV informal (el que está encerrado en /* */) para indicar alguna expresión Meta IV que por razones de economía de espacio no se ha incluido. El texto informal en Meta IV es semejante al texto informal en LED y se utiliza ampliamente en los ejemplos de esta sección.

Normalmente se hace una distinción entre funciones aplicativas e imperativas. Las funciones aplicativas no se refieren a partes del estado global (variables), es decir, el resultado de dichas funciones sólo depende del valor de los parámetros efectivos aplicados. El cuerpo de una función aplicativa está restringido a ser una expresión ya que las sentencias imponen algunos cambios de estado. Las funciones aplicativas siempre tienen que producir un resultado. Las funciones imperativas se refieren o incluso cambian el estado global (funciones con efectos laterales). Si una función es imperativa, esto debe reflejarse en la cláusula de tipo utilizando => en lugar de → al especificar el resultado. Es decir:

$f(a, b)(d) \triangleq$

1 /* *expression referring to the global state or sequence of statements* */

type: $DomX\ DomY \rightarrow DomZ \Rightarrow DomW$

En la DF, la semántica estática y la creación del dominio interno *Entity-dict* en la semántica dinámica son aplicativas.

5.3 *Definición de variables*

Las variables globales se definen en el nivel más exterior de las definiciones de procesadores. Son visibles a todas las funciones utilizadas por el procesador que define la variable, aunque las funciones normalmente se definen fuera de las definiciones de procesador. Sin embargo, no se permite que una función utilizada por dos o más procesadores tenga acceso a variables. Cuando existen varias instancias de un determinado procesador, también existen varias instancias de las variables definidas por el mismo (no hay variables compartidas).

Las definiciones de variables se presentan especificando la palabra clave **dcl** seguida de una lista de nombres de variables, seguida opcionalmente por una expresión inicial y terminada por el tipo de la variable.

Ejemplo

```
dcl v1 := 5 type Intg;
dcl v2 type DomD;
```

Se han definido dos variables v1 y v2; v1 es de tipo entero y se inicializa a 5. v2 es de tipo *DomD* . Nótese que las variables siempre pueden distinguirse sintácticamente de otros nombres, ya que no están en itálica. Una sintaxis alternativa de definición de variable es:

```
dcl v1 := 5 type Intg,
    v2 type DomD;
```

Se accede al valor asociado a las variables utilizando el operador de contenido, que es la palabra clave **c**.

Ejemplo

```
f() ≙
  1   c v1 + c v2
type:  () ⇒ Intg
```

5.4 *Dominios*

Los dominios se definen normalmente al principio de un documento. Los nombres de los dominios pueden distinguirse sintácticamente de otros nombres ya que la primera letra es mayúscula. Un dominio se define especificando el nombre del mismo seguido de un símbolo «::» (o mediante «=» en el caso de un nombre sinónimo tal como se explica en el § 5.4.1), seguido por una expresión de dominio que refleja sus propiedades (véase también el § 1.5.1 de la Recomendación Z.100 para una introducción a la notación de los dominios).

Ejemplo

```
8  Output-node1          :: Signal-identifier1
                             [Expression1]*
                             [Signal-destination1]
                             Direct-via1
```

Este ejemplo está tomado de la sintaxis abstracta del LED (a efectos de claridad, todos los nombres de AS_1 tienen el sufijo «$_1$» en la DF). Define un árbol con nombre, es decir, un tipo de datos semejante a un registro en el que el nombre del tipo de registro es *Output-node*$_1$ y sus campos son de tipo *Signal-identifier*$_1$, [*Expression*$_1$]*, [*Signal-destination*$_1$] y *Direct-via*$_1$.

El operador más importante para árboles con nombre es el operador **mk- (**make**)** que se utiliza para componer y descomponer objetos árbol (es decir, valores de registros).

Por ejemplo, si un nombre *sigid* denota un objeto de dominio *Signal-identifier*$_1$, un nombre *exprlist* denota un objeto de dominio [*Expression*$_1$]*, un nombre *dest* denota un objeto de tipo [*Signal-destination*$_1$] y un nombre *via* denota un objeto de dominio *Direct-via*$_1$, entonces un objeto de dominio *Output-node*$_1$ se construye escribiendo:

mk-*Output-node*$_1$ (*sigid, exprlist, dest, via*)

que puede utilizarse en una expresión Meta IV. Obsérvese que el orden en que se especifican los argumentos en el operador **mk-** es significativo. Esto también se aplica a llamadas de funciones.

De forma similar, si se tiene un objeto denominado *outputnode* del dominio *Output-node*$_1$ y se desea acceder a los campos, pueden introducirse nombres para los campos descomponiéndolo (se han elegido aquí los mismos nombres que antes):

let mk-*Output-node*$_1$ (*sigid, exprlist, dest, via*) = *outputnode* **in**
/* *some expression using the fields* */

Por medio de la construcción **let** hemos introducido nombres para indicar los campos del objeto *outputnode*. La manera general de introducir nombres consiste en utilizar la construcción **let** (no sólo en combinación con el operador **mk-**). En el § 5.5 se explica la construcción **let**.

Si algunos de los campos no se utilizan en la expresión, pueden omitirse los nombres correspondientes en la descomposición. Por ejemplo, si *sigid* no se utiliza en la expresión, puede escribirse:

let mk-*Output-node*$_1$ (,*exprlist, dest, via*) = *outputnode* **in**
/* *some expression using exprlist and dest* */

Si sólo se desea utilizar el *Signal-Identifier*$_1$ en la expresión, puede, alternativamente, utilizarse el operador de selección de campo **s-**:

let *sigid* = **s-***Signal-Identifier*$_1$ (*outputnode*) **in**
/* *some expression using sigid* */

El operador de selección de campo sólo puede utilizarse si el campo se determina unívocamente mediante el nombre del dominio.

Si se gana legibilidad, los parámetros formales pueden descomponerse (es decir, pueden introducirse nombres para los elementos contenidos) en el encabezamiento de la función y no en el cuerpo de la misma. Es decir,

$$\textit{int-create-node}(\textbf{mk-}\textit{Create-request-node}_1(\textit{pid}, \textit{exprl}))(\textit{dict}) \triangleq$$
$$1 \quad /* \textit{ body of int-create-node } */$$
$$\textbf{type:} \quad \textit{Create-request-node}_1 \rightarrow \textit{Entity-dict} \Rightarrow$$

es equivalente a:

$$\textit{int-create-node}(\textit{createnode})(\textit{dict}) \triangleq$$
$$1 \quad (\textbf{let } \textbf{mk-}\textit{Create-request-node}_1(\textit{pid}, \textit{exprl}) = \textit{createnode } \textbf{in}$$
$$2 \quad /* \textit{ body of int-create-node } */)$$
$$\textbf{type:} \quad \textit{Create-request-node}_1 \rightarrow \textit{Entity-dict} \Rightarrow$$

Nótese que en este ejemplo también se tiene una segunda lista de parámetros que contiene el parámetro formal *dict* del dominio *Entity-dict*.

5.4.1 *Sinónimos*

Sólo es posible utilizar el operador de selección de campo si en la definición del dominio el campo se representa por un nombre. Si por ejemplo desea utilizarse el operador de selección sobre el segundo campo de objetos del dominio *Output-node*$_1$, éste debe definirse de manera ligeramente diferente:

$$9 \quad \textit{Output-node}_1 :: \textit{Signal-identifier}_1 \; \textit{Valuelist} \; [\textit{Signal-destination}_1] \; \textit{Direct-Via}_1$$
$$10 \quad \textit{Valuelist} = [\textit{Expression}_1]^*$$

Este *Output-node*$_1$ es exactamente el mismo dominio que el *Output-node*$_1$ definido previamente. La única diferencia es que se ha dado un nombre al segundo campo, es decir, se ha definido un sinónimo o una notación taquigráfica para la expresión de dominio, a saber [*Expression*$_1$] * (el símbolo «=» se utiliza al definir sinónimos). Frecuentemente hay otras razones para definir sinónimos como por ejemplo si la misma expresión de dominio se utiliza en diversos lugares o en aras de una mayor claridad. Por ejemplo, en la sintaxis abstracta del LED, *Channel-name*$_1$, *Block-name*$_1$, *Process-name*$_1$, etc., son todos sinónimos del dominio *Name*$_1$, pero transmiten al lector información sobre la pertenencia a ciertas clases de entidad de los objetos representados por los distintos *Name*$_1$s. Otro caso típico es cuando se dispone de una larga lista de alternativas. Por ejemplo, la sintaxis abstracta de *Expression*$_1$ es

$$11 \quad \textit{Expression}_1 = \textit{Ground-expression}_1 \mid \textit{Active-expression}_1$$
$$12 \quad \textit{Active-expression}_1 = \textit{Variable-access}_1 \mid \textit{Conditional-expression}_1 \mid \textit{Operator-application}_1 \mid \textit{Imperative-operator}_1$$
$$13 \quad \textit{Imperative-operator}_1 = \textit{Now-expression}_1 \mid \textit{Pid-expression}_1 \mid \textit{View-expression}_1 \mid \textit{Timer-active-expression}_1$$

que refleja mejor la agrupación de las distintas clases de expresiones que

```
14  Expression1          = Ground-expression1 |
                           Variable-access1 |
                           Conditional-expression1 |
                           Operator-application1 |
                           Now-expression1 |
                           Pid-expression1 |
                           View-expression1 |
                           Timer-active-expression1
```

5.4.2 *Árboles sin nombre*

En algunos casos no es necesario dar nombre a una definición de árbol. Los árboles sin nombre son ampliamente utilizados en la DF, pero se mantienen anónimos ya que a menudo no es necesario definirlos explícitamente.

Ejemplo

La primera línea de la definición de *Entity-dict* en la semántica dinámica es:

```
15  Entity-dict          = (Identifier1 Entityclass) m→Entitydescr
```

que expresa que la *Entity-dict* incluye una aplicación de los dos dominios *Identifier*$_1$ y *Entityclass* en algún descriptor (*Entitydescr*). Ambos dominios constituyen un árbol sin nombre. Si se deseara utilizar un árbol con nombre habría que cambiar la definición por la siguiente:

```
16  Entity-dict          = Pair m→Entitydescr
17  Pair                 :: Identifier1 Entityclass
```

Ejemplo

En la semántica dinámica *Reachability* se define como

```
18  Reachability         = (Process-identifier1 | ENVIRONMENT)
                           Signal-identifier1-set Path
```

en donde se ha definido un sinónimo para un árbol sin nombre que contiene tres campos:

1) Un campo que puede contener un identificador de proceso o el literal citación **ENVIRONMENT**.

2) Un campo que contiene un conjunto de identificadores de señales.

3) Un campo del dominio *Path*.

Como puede verse, los paréntesis se utilizan en las definiciones de los dominios para definir árboles sin nombre y para agrupar alternativas.

Ejemplo

La función de la semántica dinámica *make-formal-parameters* se define como:

```
make-formal-parameters(parml, level) ≜
  1  /* The body, which is not shown here */
type:  Procedure-formal-parameter1* Qualifier1 → FormparmD* Entity-dict
```

Esta función devuelve dos objetos, *FormparmD** y *Entity-dict*, lo que significa que de hecho devuelve un árbol sin nombre que consta de dos objetos.

El operador **mk-** no puede utilizarse en árboles sin nombre. La composición y descomposición de éstos se hace encerrando los campos entre paréntesis.

Ejemplo

Composición de un objeto *Reachability* en el que *a* denota un *Process-Identifier*$_1$, *b* denota un conjunto de identificadores de señales y *d* denota un *Path*:

(*a, b, d*)

Si, en aras de la legibilidad, desea utilizarse un nombre para denotar un objeto (es más fácil manejar un nombre que (*a, b, d*), especialmente si (*a, b, d*) se utiliza varias veces en una expresión), puede usarse de nuevo la construcción **let**, es decir, la expresión:

/* *some expression using «(a, b, d)»* */

que es equivalente a

(let *reach* = (*a, b, d*) **in**
/* *some expression using «reach»* */

La construcción **let** también se utiliza para descomponer objetos de árboles sin nombre. Por ejemplo, una descomposición de un objeto *Reachability* denominado *reach* en el que por alguna razón no se usa el conjunto de identificadores de señales, es la siguiente:

let (*a, ,d*) = *reach* **in**
/* *some expression using a and d* */

Cuando se llama a una función, es habitual descomponer los árboles sin nombre que resultan de haber llamado a la función, es decir:

let (*parmlist, pathlist*) = *make-formal-parameters* (. . . , . . .) **in**
/* *some expression using the function results parmlist and pathlist* */

es equivalente a:

let *parminf* = *make-formal-parameters* (. . . , . . .) **in**
let (*parmlist, pathlist*) = *parminf* **in**
/* *some expression using the function results parmlist and pathlist* */

5.4.3 *Construcciones de ramificación*

En algunos casos debe ser posible distinguir un cierto número de objetos de árboles unos de otros. Por ejemplo, objetos del sinónimo *Imperative-operator*$_1$ previamente definido pueden ser *Now-expression*$_1$, *Pid-expression*$_1$, *View-expression*$_1$, etc. Al examinar un *Imperative-operator*$_1$, debe determinarse en primer lugar el tipo de *Imperative-operator*$_1$ antes de evaluarlo. Para ello puede utilizarse la sentencia/expresión «case». Por ejemplo, la función que evalúa las expresiones LED imperativas podría ser:

```
eval-imperative-expression(expr) ≙

1  cases expr:
2   (mk-Now-expression1()
3      → eval-now-expression(),
4    mk-View-expression1(vid, pidexpr)
5      → eval-view-expression(vid, pidexpr),
6    mk-Timer-active-expression1(tid, actlist)
7      → eval-timer-expression(tid, actlist),
8    T → eval-pid-expression(expr))

type:  Imperative-operator1 ⇒
```

Nótese que la ramificación se hace según el tipo del *Imperative-operator* y no según el valor efectivo de los campos del árbol. **T** denota una cláusula de «otro caso» que se ha utilizado debido a que la alternativa final en *Imperative-operator*$_1$ (*Pid-expression*$_1$) es un sinónimo que representa otras cuatro alternativas que no se distinguen en este ejemplo. La evaluación de dichas alternativas se hace posteriormente, en *eval-pid-expression*.

Otra manera de hacerlo es utilizando el operador booleano **is-** que devuelve el valor **true** (**verdadero**) si el objeto dado como argumento es de un cierto dominio, por ejemplo:

```
eval-imperative-expression(expr) ≜

 1  if is-Now-expression1(expr) then
 2     eval-now-expression()
 3    else
 4     if is-View-expression1(expr) then
 5        eval-view-expression(s-Variable-identifier1(expr), s-Expression1(expr))
 6       else
 7        if is-Timer-active-expression1(expr) then
 8          (let mk-Timer-active-expression1(tid, actlist) = expr in
 9           eval-timer-expression(tid, actlist))
10          else
11           eval-pid-expression(expr)

type :  Imperative-operator1 ⇒
```

Nótese que se ilustran tanto el acceso a los campos por descomposición (línea 8) como el acceso por medio del operador de selección de campo (línea 5).

Como en la mayoría de los lenguajes de especificación y programación, se requiere que las alternativas que figuran en la sentencia /expresión «case» sean «constantes» (como lo son cuando se ramifica según el tipo de árbol), lo que significa que si las alternativas son de naturaleza dinámica (variables o parámetros formales) debe utilizarse la construcción «if-then-else». Hay, sin embargo, otra notación para la construcción «if-then-else», la denominada construcción Mc-Carthy, que es más recomendable si hay muchas alternativas:

```
eval-imperative-expression(expr) ≜

 1  (is-Now-expression1(expr)
 2     → eval-now-expression(),
 3   is-View-expression1(expr)
 4     → (let mk-View-expression1(vid, pidexpr) = expr in
 5        eval-view-expression(vid, pidexpr)),
 6   is-Timer-expression1(expr)
 7     → (let mk-Timer-expression1(tid, actlist) = expr in
 8        eval-timer-expression(tid, actlist)),
 9   T → eval-pid-expression(expr))

type :  Imperative-operator1 ⇒
```

Nótese que algunos nombres de funciones DF también empiezan por «is-». Estos casos pueden distinguirse fácilmente del operador «**is-**» ya que no están en letra negrita.

5.4.4 *Dominios elementales*

El Meta IV proporciona una serie de dominios elementales predefinidos. En los puntos siguientes se describe su notación y los operadores asociados.

5.4.4.1 *Booleano*

El nombre *Bool* del Meta IV denota el dominio de los valores de verdad, es decir, el conjunto {**true, false**}.

Operadores para booleanos:

Notación	Tipo	Operación
¬	*Bool* → *Bool*	negación
∧	*Bool* → *Bool*	y lógico
∨	*Bool* → *Bool*	o lógico
⊃	*Bool* → *Bool*	contiene
=	*Bool Bool* → *Bool*	igual
≠	*Bool Bool* → *Bool*	diferente

Ejemplo

En términos de expresiones Meta IV, las propiedades de los operadores *Bool* ¬, ∧, ∨ y ⊃ pueden ilustrarse como sigue:

$\neg a = (\textbf{if } a \textbf{ then false else true})$
$a \vee b = (\textbf{if } a \textbf{ then true else } b)$
$a \wedge b = (\textbf{if } a \textbf{ then } b \textbf{ else false})$
$a \supset b = (\textbf{if } a \textbf{ then } b \textbf{ else true})$

5.4.4.2 *Entero*

Se predefinen tres nombres de dominios para valores enteros:

– El nombre *Intg* denota el dominio de todos los valores enteros, es decir, el conjunto {. . ., –2,–1,0,1,2, . . .}

– El nombre N_0 denota el dominio de los valores enteros no negativos, es decir, el conjunto {0,1,2, . . .}

– El nombre N_1 denota el dominio de los valores enteros positivos, es decir, el conjunto {1,2, . . .}

Operadores para enteros:

Notación	Tipo	Operación
–	*Intg* → *Intg*	negación
–	*Intg Intg* → *Intg*	substracción
+	*Intg Intg* → *Intg*	suma
*	*Intg Intg* → *Intg*	multiplicación
/	*Intg Intg* → *Intg*	división entera
mod	N_0 N_1 → N_0	módulo
=	*Intg Intg* → *Bool*	igual
≠	*Intg Intg* → *Bool*	diferente
<	*Intg Intg* → *Bool*	inferior a
≤	*Intg Intg* → *Bool*	inferior o igual a
>	*Intg Intg* → *Bool*	superior a
≥	*Intg Intg* → *Bool*	superior o igual a

5.4.4.3 *Carácter*

El nombre *Char* del Meta IV denota el dominio de valores de caracteres ASCII. Para los caracteres imprimibles existen representaciones de objetos que se ponen entre comillas, por ejemplo, “a”, “Z”, “ ”.

Operadores para caracteres:

Notación	Tipo	Operación
=	*Char Char* → *Bool*	igual
≠	*Char Char* → *Bool*	diferente
<	*Char Char* → *Bool*	inferior a
⩽	*Char Char* → *Bool*	inferior o igual a
>	*Char Char* → *Bool*	superior a
⩾	*Char Char* → *Bool*	superior o igual a

Los operadores relacionales se aplican a los valores numéricos ASCII asociados.

En aras de mayor claridad, los objetos del dominio *Char* $^+$ pueden representarse por una secuencia de caracteres entre comillas, por ejemplo, "abc" es lo mismo que ("a", "b", "c") (véase el § 5.4.6).

5.4.4.4 *Citación*

El nombre *Quot* del Meta IV denota el dominio de citaciones. Son objetos elementales distintos y se representan como cualquier secuencia de mayúsculas y números en negrita, por ejemplo, **ENVIRONMENT**, **REVERSE**.

Operadores para citaciones:

Notación	Tipo	Operación
=	*Quot Quot* → *Bool*	igual
≠	*Quot Quot* → *Bool*	diferente

En contraposición a otros dominios, los objetos de *Quot* pueden figurar en definiciones de dominios en los que sólo cierto(s) objeto(s) de *Quot* es(son) posible(s) en un contexto dado; por ejemplo, en la sintaxis abstracta de la Recomendación Z.100, *Originating-block*$_1$ se define como:

1 *Originating-block*$_1$ = *Block-identifier*$_1$| **ENVIRONMENT**

Alternativamente, *Originating-block*$_1$ puede definirse usando *Quot*:

2 *Originating-block*$_1$ = *Block-identifier*$_1$ | *Quot*

Sin embargo, resulta más preciso utilizar **ENVIRONMENT** en la definición de dominio, ya que este objeto es el único valor posible de *Quot* en ese contexto.

5.4.4.5 *Testigo (Token)*

El nombre *Token* del Meta IV denota el dominio de testigos (tokens). Puede considerarse que este dominio consiste en un conjunto potencialmente infinito de objetos elementales distintos para los que no se requiere representación.

Operadores para testigos:

Notación	Tipo	Operación
=	*Token Token* → *Bool*	igual
≠	*Token Token* → *Bool*	diferente

Ejemplo

Name$_1$ se define, según la sintaxis abstracta de la Recomendación Z.100, como:

1 *Name*$_1$ :: *Token*

La única propiedad necesaria para *Name*$_1$ durante la interpretación es la igualdad. Por lo tanto *Name*$_1$ consiste en un valor *Token* (la ortografía real de los nombres es irrelevante).

5.4.4.6 *Elipsis*

El dominio de elipsis (representado por …) denota una construcción no especificada. Se utiliza en definiciones de dominios o en expresiones:

– cuando la expresión o dominio efectivo no es de importancia para la semántica, o

– cuando la elaboración del dominio o de la expresión está fuera del ámbito de la especificación.

Ejemplo

En la sintaxis abstracta de la Recomendación Z.100, *Informal-text*$_1$ se define como:

1 *Informal-text*$_1$ :: …

Informal-text$_1$ no puede interpretarse utilizando el Meta IV. Por lo tanto *Informal-text*$_1$ contiene algunos objetos no especificados.

5.4.5 *Dominios conjunto*

Un dominio conjunto se construye añadiendo como posfijo al dominio de elementos la palabra clave **-set** (el guión es significativo). Por ejemplo,

2 *State-node*$_1$ :: *State-name*$_1$
Save-signalset$_1$
Input-node$_1$ **-set**

3 *Save-signalset* :: *Signal-Identifier*$_1$ **-set**

expresa que los objetos del dominio *State-node*$_1$ constan de un nombre de estado, un conjunto de señales de conservación que contiene un conjunto de identificadores de señal, y un conjunto de nodos de entrada. Los valores de los conjuntos pueden construirse utilizando un constructor de conjuntos explícito, que es una lista de expresiones encerradas entre llaves; es decir:

{ 1,3,5,1 }

denota un objeto del dominio *Intg-set* y contiene los tres valores *Intg* 1,3,5. Una forma más habitual es la denominada constructor implícito de conjuntos, en la que el conjunto incluye todos los elementos que satisfacen una cierta condición (predicado). Por ejemplo:

$$\{i \in Intg \mid 0 \leq i \leq 5 \vee i \bmod 2 = 0\}$$

define el conjunto

{ 0,1,2,3,4,5,6,8,10,12,14,16, . . . }

Significa: El conjunto de valores situados a la izquierda de la barra vertical (calificados posiblemente por un valor o por un dominio) para los cuales se cumple la expresión que figura a la derecha de la barra vertical.

El conjunto vacío se representa mediante { }.

En la explicación que sigue de la semántica de los operadores de conjuntos, *s* denota el conjunto { 1,3,5 }:

$\in$ Operador de pertenencia.

Comprueba si un elemento del dominio de elementos está contenido en un conjunto; es decir, 1 $\in$ *s* ≡ **true** y 2 $\in$ *s* ≡ **false**.

$\notin$ Comprueba si un elemento del dominio de elementos está incluido de un conjunto; es decir, 1 $\notin$ *s* ≡ **false** y 2 $\notin$ *s* ≡ **true**.

$\cup$ Operador unión.

Une dos conjuntos; es decir, { 2,3 } $\cup$ *s* ≡ { 1,2,3,5 } y *s* $\cup$ *s* ≡ *s*.

$\cap$ Operador intersección. Devuelve la intersección de dos conjuntos; es decir, { 2,3 } $\cap$ *s* ≡ { 3 } y { } $\cap$ *s* ≡ { }.

$\setminus$ Operador complemento.

Excluye un conjunto determinado de valores de otro conjunto; es decir, *s* $\setminus$ { 1,2 } ≡ { 3,5 } y { 1,2 } $\setminus$ *s* ≡ { 2 }.

$\subseteq$ Operador de subconjunto propio.

Comprueba si los elementos de un conjunto dado están contenidos en un conjunto; es decir, { 1,5 } $\subseteq$ *s* ≡ **true**, *s* $\subseteq$ { 1,5 } ≡ **false** y *s* $\subseteq$ *s* ≡ **false**.

$\subset$ Operador de subconjunto.

Comprueba si los elementos de un conjunto determinado están contenidos o son iguales a un conjunto; es decir, { 1,5 } $\subset$ *s* ≡ **true**, *s* $\subset$ { 1,5 } ≡ **false** y *s* $\subset$ *s* ≡ **true**.

card Operador de cardinalidad.

Devuelve el número de elementos de un conjunto; es decir **card** *s* ≡ 3 y **card** { } ≡ 0.

union Operador de unión distribuida.

El argumento es un conjunto de conjuntos y el resultado es la unión de todos los conjuntos contenidos en el argumento; es decir, **union** { *s*, { 5,6 }, { 1,5,8 }} ≡ *s* $\cup$ { 5,6 } $\cup$ { 1,5,8 } ≡ { 1,3,5,6,8 }.

=,≠ Prueba de igualdad y desigualdad de conjuntos.

Ejemplo

En términos de expresiones Meta IV, las propiedades de los operadores de conjuntos $\notin$, $\cup$, $\cap$, $\subseteq$, $\subset$, **card** y **union** pueden ilustrarse como sigue:

$$element \notin s1 = (\neg(element \in s1))$$
$$s1 \cup s2 = \{element \mid element \in s1 \vee element \in s2\}$$
$$s1 \cap s2 = \{element \mid element \in s1 \wedge element \in s2\}$$
$$s1 \setminus s2 = \{element \mid element \in s1 \wedge element \notin s2\}$$
$$s1 \subseteq s2 = (\forall element \in s1)(element \in s2) \wedge s1 \neq s2$$
$$s1 \subset s2 = (\forall element \in s1)(element \in s2)$$
$$\textbf{card}\ s1 = (\textbf{if}\ s1 = \{\}\ \textbf{then}\ 0\ \textbf{else}\ (\textbf{let}\ element \in s1\ \textbf{in}\ 1 + \textbf{card}\ (s1 \setminus \{element\})))$$
$$\textbf{union}\ s1 \equiv \{element \mid (\exists set \in s1)(element \in set)\}$$

Les quantificateurs ($\forall$ et $\exists$) sont expliqués dans le § 5.6.

5.4.6 *Dominios lista*

Un dominio lista o tupla se construye añadiendo como posfijo al dominio de elementos un «*» en el caso de una lista posiblemente vacía y un «$^+$» en los demás casos.

Ejemplo

4 *Signal-definition*$_1$ :: *Signal-name*$_1$
Sort-reference-identifier$_1$ *

Esta definición de dominio expresa que una definición de señal consiste en un nombre de señal y una lista posiblemente vacía de identificadores de género.

Un valor de lista puede construirse utilizando un constructor de tupla explícito. Este es una lista de expresiones encerradas en tres paréntesis angulares; por ejemplo,

<11,12,13,14>

denota un objeto del dominio $Intg^+$ (o *Intg**) y contiene 5 elementos ordenados.

La lista vacía se denota mediante < >.

Existen también constructores de lista implícitos similares a los de conjuntos. Por ejemplo, en la función de la semántica dinámica *int-output-node* se construye una tupla (*vall*) que contiene los valores de todos los parámetros efectivos (*exprl*) de un nodo de salida:

let *vall* = <*eval-expression* (*exprl*[*i*]) (*dict*) | **1** £ *i* $ **len** *exprl*> **in**

que es equivalente a una enumeración explícita de todos los elementos de la lista:

let *vall* = <(*eval-expression*(*exprl*[1] (*dict*),
eval-expression(*exprl*[2]) (*dict*),
eval-expression(*exprl*[3]) (*dict*),
...> **in**

Nótese que los paréntesis de tupla (< y >) tienen una forma diferente a los operadores relacionales (< y >).

En la explicación que sigue de la semántica de los operadores de listas, *l* denota la lista <11,12,11,13,14>:

hd	Devuelve el primer elemento (la cabeza de la lista). Es decir, **hd** *l* ≡ 11. El argumento de **hd** no debe ser una lista vacía (< >).
tl	Devuelve la lista de la que se ha eliminado el primer elemento («tail», cola). Es decir, **tl** *l* ≡ <12,11,13,14>
[*i*]	Devuelve el elemento número *i* de una lista. Es decir *l*[3]≡11 y *l*[5]≡14. El valor del índice no debe ser menor que uno ni mayor que la longitud de la lista.
len	Devuelve la longitud de la lista. Es decir, **len** *l* ≡ 5.
elems	Devuelve un conjunto que consta de los elementos que figuran en la lista. Es decir, **elems** *l* ≡ { 11,12,13,14 }
ind	Devuelve el conjunto de objetos enteros que son los valores de índices válidos de una lista. Es decir, **ind** *l* ≡ { 1,2,3,4,5 }
⌢	Concatena dos listas. Es decir, *l*⌢ <0,1> ≡ <11,12,11,13,14,0,1>
conc	Concatena todas las listas que son elementos de la lista dada como argumento. Es decir, **conc** ≪0,7>,*l*,<9≫ ≡ <0,7,11,12,11,13,14,9>
=, ≠	Prueba de igualdad y desigualdad de listas.

Ejemplo

En términos de expresiones Meta IV, las propiedades de los operadores de listas **hd**, **tl**, **ind**, **elems** y **conc** pueden ilustrarse como sigue:

$$\mathbf{hd}\ l = (\mathbf{if}\ l = \langle\rangle\ \mathbf{then\ undefined\ else}\ l[1])$$
$$\mathbf{tl}\ l = \langle l[i] \mid 2 \leq i \leq \mathbf{len}\ l\rangle$$
$$\mathbf{ind}\ l = \{i \mid 1 \leq i \leq \mathbf{len}\ l\}$$
$$\mathbf{elems}\ l = \{l[i] \mid i \in \mathbf{ind}\ l\}$$
$$\mathbf{conc}\ l = (\mathbf{if}\ l = \langle\rangle\ \mathbf{then}\ \langle\rangle\ \mathbf{else}\ \mathbf{hd}\ l \frown \mathbf{conc}\ \mathbf{tl}\ l)$$

5.4.7 *Dominios aplicación (correspondencia)*

Un dominio de correspondencia (es decir, un cuadro) se construye especificando el dominio de los objetos de entrada, seguido por el operador $\overrightarrow{m}$ y por el dominio de los objetos contenidos en la aplicación (los valores de llegada).

Ejemplo

```
5  Entity-dict          = (Identifier1 Entityclass) m→ Entitydescr ∪
                          ENVIRONMENT m→ Reachability-set ∪
                          EXPIREDF    m→ Is-expired ∪
                          PIDSORT     m→ Identifier1 ∪
                          NULLVALUE   m→ Identifier1 ∪
                          TRUEVALUE   m→ Identifier1 ∪
                          FALSEVALUE  m→ Identifier1
```

En él se da la definición completa de la aplicación *Entity-dict*. Se muestra cómo se utiliza el operador $\overset{m}{\rightarrow}$ y también que pueden construirse aplicaciones compuestas usando el operador de fusión de dominios $\overset{m}{\rightarrow}$; es decir, dada una relación de correspondencia de dominio *Entity-dict*, se consulta de la manera siguiente:

- se aplica un objeto del árbol sin nombre (*Identifier*$_1$ *Entityclass*) y el resultado es un objeto del dominio *Entitydescr*, o
- se aplica el valor **ENVIRONMENT** de *Quot* y el resultado es un objeto del dominio *Reachability*-set, o
- se aplica el valor **EXPIREDF** de *Quot* y el resultado es un objeto del dominio *Is-expired*, o
- se aplica el valor **PIDSORT** de *Quot* y el resultado es un objeto del dominio *Identifier*$_1$, o
- se aplica el valor **NULLVALUE** de *Quot* y el resultado es un objeto del dominio *Identifier*$_1$, o
- se aplica el valor **TRUEVALUE** de *Quot* y el resultado es un objeto del dominio *Identifier*$_1$, o
- se aplica el valor **FALSEVALUE** de *Quot* y el resultado es un objeto del dominio *Identifier*$_1$.

Sólo puede aplicarse un valor si éste se ha incluido previamente en el objeto de aplicación, en contraposición a las funciones en las que la correspondencia entre los valores de los argumentos y los valores de los resultados se fija y define cuando se define la función.

Los valores de la aplicación pueden construirse utilizando un constructor explícito, que es una lista de pares de valores de entrada y valores de llegada encerrados entre corchetes; es decir:

[1 → **D**,
[2 → **AA**,
[4 → **BB**,
[9 → **ABC**,
[5 → **XYZ**]

denota un valor de aplicación de dominio *Intg* $\overset{m}{\rightarrow}$ *Quot*.

También pueden construirse aplicaciones implícitas. Por ejemplo la aplicación implícita.

$$[a \mapsto b \mid a \in N_1 \wedge a * a = b]$$

es equivalente a la aplicación infinita.

[1 → 1,
[2 → 4,
[3 → 9,
... → ...]

En la explicación que se da a continuación de la semántica de los operadores de aplicación, *m* denota la primera de las aplicaciones explícitamente especificadas anteriormente:

m(entryvalue) Devuelve un valor de una aplicación; es decir, *m* (1) ≡ **D** y *m* (9) ≡ **ABC**

\+ Escribe una aplicación sobre otra. Este operador no es conmutativo; es decir

m + [0 → **XX**, 1 → **B**] ≡
[0 → **XX**,1 → **B**,2 → **AA**,4 → **BB**,9 → **ABC**,5 → **XYZ**]

mientras que

[0 → **XX**,1 → **B**] + *m* ≡
[0 → **XX**,1 → **D**,2 → **AA**,4 → **BB**,9 → **ABC**,5 → **XYZ**]

Excluye de una aplicación un determinado conjunto de valores de entrada; es decir
m { 1,2,3 }≡

[4 → **BB**,9 → **ABC**,5 → **XYZ**]

dom Devuelve el conjunto que contiene exactamente los valores de entrada presentes en una aplicación dada; es decir:

dom *m* ≡ { 1,2,4,5,9 }

rng Devuelve el conjunto que contiene exactamente los valores de llegada presentes en una aplicación dada; es decir:

rng *m* ≡ { **D**,**AA**,**BB**,**ABC**,**XYZ**}

=,≠ Prueba de la igualdad y desigualdad de dos aplicaciones.

merge De un conjunto dado de aplicaciones devuelve la que está construida fusionando todas las aplicaciones contenidas en el conjunto; es decir:

{ *m*,[0 → **WE**],[10 → **D**] }≡
[0 → **WE**,10 → **D**,1 → **D**, 2 → **AA**,4 → **BB**,9 → **ABC**,5 → **XYZ**]

Si dos o más aplicaciones cualesquiera contenidas en el conjunto tienen entradas superpuestas, se elige un valor arbitrario entre los valores posibles.

La aplicación vacía se denota mediante [ ] (dos corchetes muy próximos el uno al otro).

Ejemplo

En términos de expresiones Meta IV, las propiedades de los operadores de aplicación \ , + y **merge** pueden ilustrarse como sigue:

$$m1 \setminus s = [a \mapsto b \mid a \in \mathbf{dom}\ m1 \setminus s \wedge m1(a) = b]$$
$$m1 + m2 = [a \mapsto b \mid (a \in \mathbf{dom}\ m2 \wedge m2(a) = b) \vee (a \in \mathbf{dom}\ m1 \setminus \mathbf{dom}\ m2 \wedge m1(a) = b)]$$
$$\mathbf{merge}\ m1 = (\mathbf{if}\ m1 = \{\}\ \mathbf{then}\ [] \ \mathbf{else}\ (\mathbf{let}\ element \in m1\ \mathbf{in}\ element + \mathbf{merge}\ m1 \setminus \{element\}))$$

5.4.8 *Dominios Pid*

Un dominio Pid (correspondiente al género Pid del LED) se construye por medio del símbolo **Π**. Opcionalmente, puede calificarse por el tipo de procesador para indicar que clase de valores Pid denota el dominio; por ejemplo:

6 *Discard-Signals* :: *Π(input-port)*

El dominio *Discard-Signals* (definido en la semántica dinámica) contiene objetos Pid calificados por el tipo de procesador *input-port*. Los valores Pid del Meta IV no deben confundirse con los valores Pid del LED que son *Ground-term*$_1$. En la semántica dinámica se define el dominio de los valores Pid del LED como:

7 *Pid-Value* = *Value*

8 *Value* = *Ground-term*$_1$

Los valores Pid de Meta IV se crean aplicando la sentencia/expresión **start** (arranque). Corresponde a la acción de petición de creación en el LED. Por ejemplo, cuando el procesador *system* crea una instancia de un procesador *timer* con el parámetro efectivo *timer*, tenemos:

Ejemplo

start *timer (timerf)*

Cuando se utiliza la construcción de arranque como una expresión, ésta crea una instancia de procesador y devuelve el valor Pid Meta IV de dicha instancia (correspondiente al valor OFFSPRING del LED). Por ejemplo, cuando el procesador *sdl-process* arranca su procesador *input-port*:

start *input-port* (*selfp, dict*(**EXPIRED**))

Se crea una instancia del procesador *input-port* y el valor Pid de Meta IV resultante es utilizado por el *sdl-process* para identificar el *input-port*. Los parámetros *selfp* y *dict* (**EXPIRED**) se entregan a la instancia creada.

La comunicación se realiza mediante las primitivas de comunicación síncrona **input** (entrada) y **output** (salida). En la construcción de salida puede elegirse entre comunicar con una instancia específica de procesador o con una instancia no especificada de un tipo específico de procesador.

Ejemplo

output mk-*Some-tree* (*somevalue, someothervalue*, . . .) **to** *p*

donde *p* denota un valor Pid o bien es el nombre de un tipo de procesador. Los valores enviados por el procesador normalmente se incluyen en un objeto de árbol con nombre (de algún dominio de comunicación) y dichos árboles pueden, por tanto, ser equiparados al concepto de señal del LED, es decir, *Some-tree* puede considerarse como una señal.

En la construcción de entrada se especifican el objeto de la comunicación que se desea recibir y la acción que debe tomarse cuando se recibe el objeto. Además, puede especificarse un nombre que después de la recepción del objeto denote el valor Pid del procesador emisor (corresponde a SENDER en LED) o que restrinja los emisores posibles. Por ejemplo,

```
input mk-Some-tree(a, b, d) from p
   ⇒  /* some statements or an expression */
```

Después de recibir *Some-tree a, b* y *d* denotarán los valores transportados por *Some-tree*, pudiendo haber tres interpretaciones posibles para *p*:

– si *p* es un nombre de tipo de procesador, la entrada debe recibirse de una instancia de ese tipo particular de procesador;

– si *p* es un nombre aún no definido, esta ocurrencia es la que define al nombre y es visible en la expresión o sentencia que sigue a la cláusula de entrada. Denota el valor Pid Meta IV del emisor;

– si *p* es una expresión, debe ser del tipo **Π** y la entrada se recibirá de la instancia de procesador indicada por la expresión.

Si se puede recibir una entrada de entre varias, se especifican una serie de construcciones de entrada separadas por comas y se encierra el todo entre llaves, a saber:

```
{input mk-Some-tree(a, b, d) from p
   ⇒  /* some statements or an expression */,
 input mk-Some-other-tree(a, b, d) from p
   ⇒  /* some statements or an expression */}
```

En algunos casos puede ser deseable especificar que debe realizarse una entrada o una salida, dependiendo de qué comunicación es posible realizar primero (no se aplica en LED debido a que en éste la comunicación es asíncrona). En dichos casos las construcciones de salida se incluyen en el conjunto de eventos de comunicaciones, es decir:

```
{input mk-Some-tree(a, b, d) from p
   ⇒  /* some statements or an expression */,
 input mk-Some-other-tree(a, b, d) from p
   ⇒  /* some statements or an expression */,
 output mk-Something(/* expression */, /* expression */) to pi}
```

Si la comunicación debe repetirse, se usa a menudo la construcción **cycle** (ciclo) junto con las de entrada y de salida, es decir:

```
cycle {input mk-Some-tree(a, b, d) from p
          ⇒  /* some statements or an expression */,
       input mk-Some-other-tree(a, b, d) from p
          ⇒  /* some statements or an expression */,
       output mk-Something(/* expression */, /* expression */) to pi}
```

lo cual significa que después de un evento de comunicación, la instancia de procesador tomará las acciones apropiadas, quedando a la espera de que ocurran nuevos eventos.

5.4.9 *Dominios referencia*

Cuando una variable de Meta IV se declara mediante

dcl v **type** *Intg*;

se asigna una posición de almacenamiento Meta IV y la variable (v) indicará una referencia a dicha posición. Cuando se accede al contenido de dicha posición, el operador **c** (operador de contenido) se utiliza como se ha mostrado anteriormente. Cuando la variable se utiliza sin el operador de contenido, el resultado es un valor del dominio **ref**, es decir, una referencia a la posición de almacenamiento. Los dominios **ref** se especifican utilizando la palabra clave **ref**, seguida del dominio pertinente. Por ejemplo:

9 *VarD* :: *Variable-identifier*$_1$ *Sort-reference-identifier*$_1$,
[**REVEALED**] **ref** *Stg*

El descriptor de parámetro IN/OUT para procedimientos incluye una referencia al dominio *Stg*. El descriptor *VarD* se define en la semántica dinámica y se describe con más detalle en las anotaciones asociadas.

5.4.10 *Dominios opcionales*

Los corchetes, que se utilizan ampliamente en la definición de dominios, indican carácter opcional.

Ejemplo

10 *VarD* :: *Signal-name*$_1$
Sort-reference-identifier$_1$ *
Signal-refinement$_1$

indica que en los objetos del árbol *Signal-definition*$_1$, el objeto del dominio *Signal-refinement* puede o no estar presente. Si no está presente el campo contendrá el valor sin tipo **nil**.

Ejemplo

```
(let mk-Signal-definition1(name, sort, refinement) = /* some Signal-definition1 object */ in
 if refinement = nil then
    /* some actions */
    else
     (let mk-Signal-refinement1(...) = refinement in
      /* some other actions using the signal refinement */))
```

5.5 *Las construcciones* **let** *y* **def**

Como se ha indicado antes, la construcción **let** puede utilizarse para componer y descomponer objetos. Más generalmente, la construcción **let** se utiliza cuando se desea un nombre para denotar algún objeto específico (frecuentemente es sólo para evitar expresiones demasiado complicadas e ilegibles). El miembro izquierdo de la construcción **let** incluye los nombres que se definen (excepto para los nombres de dominios, que siempre deben definirse en una definición de dominio). Un nombre introducido puede figurar también en el lado derecho del signo igual (en este caso el nombre se define recursivamente) y en la expresión que sigue a la construcción **let**. En el ejemplo que sigue, *name1* es visible (es decir, puede utilizarse) en /* *expression1* */, /* *expression2* */, /* *expression3* */, /* *expression4* */, *name2* es visible en /* *expression2* */, /* *expression3* */ y /* *expression4* */ y *name3* es visible en /* *expression3* */ y /* *expression4* */. A fin de restringir la visibilidad de los nombres introducidos por **let**, la construcción **let** se encierra entre paréntesis. En el ejemplo anterior un refinamiento de la señal constituye una expresión y comienza con paréntesis a la izquierda porque se utiliza una construcción **let**.

Hay dos modos de especificar una secuencia de varios **let**:

```
let name1 = /* expression1 */ in
let name2 = /* expression2 */ in
let name3 = /* expression3 */ in
/* expression4 */
```

o

```
letname1 = /* expression1 */,
  name2 = /* expression2 */,
  name3 = /* expression3 */in
/* expression4 */
```

La primera forma que incluye tres **let** se utiliza generalmente en la DF cuando el orden es importante, es decir, si /* *expression2* */ utiliza *name1* y /* *expression3* */ utiliza *name2*, mientras que la segunda forma se utiliza cuando los diversos **let** son independientes.

Existen varias formas diferentes para una construcción **let**. Ya se ha visto cómo puede utilizarse para descomponer objetos. Otras formas de interés son:

```
let name ∈ setorname1 in
/* some expression using name */
let name be s.t. /* condition using name */ in
/* some expression using name */
let name ∈ setorname2 be s.t. /* condition using name */ in
/* some expression using name */
let name(parameters) = /* function body */ in
/* some expression applying name */
```

La primera forma quiere decir: extraer un valor arbitrario que pertenezca al conjunto o al dominio denominado *setorname1*, denominando *name* al valor.

La segunda forma quiere decir: construir un valor, es decir, sea *name* tal que la condición especificada se cumpla para el valor.

La tercera forma es una combinación de las anteriores, en la que se aplican ambas restricciones. Si no existe dicho valor, la especificación es errónea.

La cuarta forma quiere decir: construir una función local (llamada *name*) que tiene un cuerpo y algunos parámetros formales (*parameters*).

Ejemplo

Definir la raíz cuadrada de 3:

```
let r ∈ Real be s.t. r > 0 ∧ r * r = 3 in
```

Ejemplo

Definir la función factorial en la que *n* es el parámetro formal:

let *fact* (*n*) = **if** *n* < 0 **then error else if** *n* = 0 **then** 1 **else** *n* * *fact* (*n* - 1) **in**

Cuando se define un nombre para un objeto que se construye haciendo referencia al estado global (es decir, si el nombre se define en términos de una expresión imperativa) la notación **def** se utiliza en lugar de la notación **let**, es decir, la palabra clave **let** se sustituye por la palabra clave **def**, el símbolo igual se sustituye por dos puntos y la palabra clave **in** se sustituye por un punto y coma (debido a que la construcción **def** se utiliza en el contexto de las sentencias, véase el § 5.7). Por ejemplo, si se desea indicar mediante un nombre un valor de instancia de procesador creado, se escribe:

(**def** *pid*: **start** *input-port* (*somevalue*);

/* *some statements using the pid value* */)

o, si se desea descomponer el resultado de una función imperativa, se escribe:

(**def mk-***Some-tree* (*a*, *b*): *some-imperative-function* (...);

/* *some statements using a and b* */)

También existe una versión con **def** de la construcción «be such that» (sea tal que):

```
(def r ∈ Real s.t. r > 0 ∧ r * r = cvl;
 /* some statements using r */)
```

donde se utiliza **def** debido a que se usa una variable (v1) en la evaluación de *r*. Ello quiere decir: definir un valor *Real r* tal que el cuadro de *r* sea igual al contenido de la variable v1.

Debe observarse que los nombres introducidos en **let** y **def** no son variables. Son nombres que representan un valor específico y no está permitido asignar un nuevo valor a dichos nombres.

5.6 *Cuantificación*

El Meta IV también proporciona los cuantificadores matemáticos, a saber, el cuantificador universal representado por el símbolo ∀ , el cuantificador de existencia representado por el símbolo ∃ y el cuantificador de unicidad representado por el símbolo ∃. Estos cuantificadores pueden utilizarse en expresiones cuantificadas que devuelven el valor booleano true (verdadero) si se satisface una condición (un predicado) especificada con respecto a un objeto.

Ejemplo

$$\begin{array}{l} \textit{identifiers-defined-on-system-level}(p) \triangleq \\ \quad 1 \quad (\forall \mathbf{mk}\text{-}\textit{Identifier}_1(q,) \in p)(\mathbf{len}\ q = 1) \\ \mathbf{type:} \quad \textit{Identifier}_1\text{-}\mathbf{set} \rightarrow \textit{Bool} \end{array}$$

Esta función devuelve el valor verdadero si, y sólo si, para todos los identificadores (*Identifier*$_1$) del conjunto *p* se cumple que la longitud de su calificador (*q*) es igual a 1 (el segundo par de paréntesis encierra la expresión predicado).

Ejemplo

$$\begin{array}{l} \textit{one-identifier-defined-on-system-level}(p) \triangleq \\ \quad 1 \quad (\exists \mathbf{mk}\text{-}\textit{Identifier}_1(q,) \in p)(\mathbf{len}\ q = 1) \\ \mathbf{type:} \quad \textit{Identifier}_1\text{-}\mathbf{set} \rightarrow \textit{Bool} \end{array}$$

Esta función devuelve verdadero si, y sólo si, existe al menos un identificador (*Identifier*$_1$) en el conjunto *p* para el cual la longitud de su calificador (*q*) es 1.

Ejemplo

$$\begin{array}{l} \textit{exactly-one-identifier-defined-on-system-level}(p) \triangleq \\ \quad 1 \quad (\exists! \mathbf{mk}\text{-}\textit{Identifier}_1(q,) \in p)(\mathbf{len}\ q = 1) \\ \mathbf{type:} \quad \textit{Identifier}_1\text{-}\mathbf{set} \rightarrow \textit{Bool} \end{array}$$

Esta función devuelve verdadero si, y sólo si, existe exactamente un identificador (*Identifier*$_1$) en el conjunto *p* para el cual la longitud de su calificador (*q*) es 1.

Alternativamente, puede descomponerse el identificador en la expresión del predicado en lugar de hacerlo en la cuantificación, es decir:

$$\begin{array}{l} \textit{identifiers-defined-on-system-level}(p) \triangleq \\ \quad 1 \quad (\forall p' \in p) \\ \quad 2 \quad ((\mathbf{let}\ \mathbf{mk}\text{-}\textit{Identifier}_1(q,) = p'\ \mathbf{in} \\ \quad 3 \quad\quad \mathbf{len}\ q = 1)) \\ \mathbf{type:} \quad \textit{Identifier}_1\text{-}\mathbf{set} \rightarrow \textit{Bool} \end{array}$$

Nótese que el apóstrofo y el guión son caracteres válidos en nombres de Meta IV.

5.7 *Sentencias auxiliares*

– Sentencia de identidad

  La palabra clave **I** indica una sentencia vacía, es decir, una sentencia que no hace nada.

– Sentencia/expresión indefinida

  La palabra clave **undefined** indica que no se puede dar ninguna semántica.

– Sentencia de retorno

  La palabra clave **return** seguida de una expresión termina la elaboración de una función imperativa, siendo el resultado la expresión dada.

– Sentencia/expresión de error

  La palabra clave **error** indica en la DF un error LED dinámico.

– Sentencia de asignación

  Como en LED. Cuando se asigna variables no se utiliza el operador de contenido (**c**).

– Sentencias for y while

  Es el mismo bien conocido concepto usado en CHILL. Las sentencias que se repiten se ponen entre paréntesis.

– Sentencias/expresiones trap y exit

  Se trata (trap) cualquier salida (exit) causada por una sentencia/expresión exit. Si se da un argumento a la sentencia exit ésta sólo se trata si la expresión dada concuerda con el valor dado en la sentencia trap exit. En las funciones *int-process-graph* e *int-procedure-graph* se ha utilizado una versión especial del mecanismo trap exit - la construcción **tixe**, que se explica en las anotaciones asociadas.

5.8 *Divergencias respecto a la notación utilizada en la definición formal del CHILL*

– En la definición formal del CHILL los nombres de dominios predefinidos están en letras mayúsculas en negrita (por ejemplo, **BOOL**, **INTG**), pudiendo los nombres de dominios semánticos estar sólo en letras mayúsculas.

  En la definición formal del LED los nombres de todos los dominios están en itálico, siendo mayúscula la primera letra y conteniendo al menos una minúscula.

– En la definición formal del CHILL todos los objetos son finitos.

  En la definición formal del LED los objetos pueden ser infinitos. La semántica de algunos de los operadores no está bien definida cuando se aplican a dichos objetos; por ejemplo, operadores como el de cardinalidad y el de igualdad no se han utilizado con objetos potencialmente infinitos.

  Además, en el anexo F.2 se ha utilizado una constante especial, *infinite*, en *transform-process*, para representar el «número ilimitado de instancias» en AS_1.

– En la definición formal del LED, se ha ampliado la notación del Meta IV para incluir el dominio elemental *Char* y los objetos cadenas de caracteres (véase el § 5.4.4.3).

– En el procesador *path* del anexo F.3 se ha utilizado una denominada «guarda de salida». El concepto se describe en las anotaciones asociadas al procesador *Path* así como en [4].

5.9 *Ejemplo: juego Demon especificado en Meta IV*

A continuación se muestra cómo puede utilizarse el Meta IV para definir la semántica del juego Demon. Para más detalles sobre el juego Demon, véase el § 2.9 de la Recomendación Z.100.

```
Communication demon → monitor and monitor → game
    11  Bump                    :: ()
Communication user → monitor
    12  Newgame                 :: ()
Communication game → monitor
    13  Gameover                :: Π
Communication monitor → game
    14  Gameoverack             :: ()
Communication game → user
    15  Gameid                  :: ()
    16  Win                     :: ()
    17  Lose                    :: ()
    18  Score                   :: Intg
Communication user → game
    19  Probe                   :: ()
    20  Result                  :: ()
    21  Endgame                 :: ()
int-demon-game() ≜
    1  start monitor()
type: () ⇒ ()
```

```
monitor processor () ≜

 1  (dcl userset := {} type Π-set,
 2       gameset := {} type Π-set;
 3   cycle (input mk-Newgame() from sender
 4              ⇒  if sender ∉ c userset then
 5                    (def offspring : start game(sender);
 6                     gameset := c gameset ∪ {offspring};
 7                     userset := c userset ∪ {sender})
 8                   else
 9                    I,
10          input mk-Gameover(player) from sender
11             ⇒  (gameset := c gameset \ {sender};
12                 userset := c userset \ {player};
13                 output mk-Gameoverack() to sender),
14          input mk-Bump() from demon
15             ⇒  for all pid ∈ gameset do
16                 output mk-Bump() to pid))

type :  () ⇒

game processor (player) ≜

 1  (dcl count := 0 type Intg;
 2   dcl even := true type Bool;
 3   output mk-Gameid() to player;
 4   cycle (input mk-Probe() from user
 5              ⇒  if c even
 6                    then (output mk-Win() to player;
 7                          count := c count + 1)
 8                    else (output mk-Lose() to player;
 9                          count := c count − 1),
10          input mk-Result() from user
11             ⇒  output mk-Score(count) to player,
12          input mk-Endgame() from user
13             ⇒  (output mk-Gameover(player) to monitor;
14                 input mk-Gameoverack() from monitor
15                    ⇒  stop),
16          input mk-Bump() from monitor
17             ⇒  even := ¬c even))

type :  Π ⇒ ()
```

Referencias

[1] BJØRNER (D.) y JONES (C. B): Formal specification and software development, Prentice-Hall, 1982.

[2] *The Formal Definition of CHILL*, CCITT Manual, ITU, Geneva, 1981.

[3] FOLKJAER (P.) y BJØRNER (D.): A formal model of a generalized CSP-like language, IFIP 8th World Computer Conference, Proceedings, North-Holland, 1980.

[4] HOARE (C. A. R.): Communicating Sequential Processes, Prentice-Hall, 1985.

SERIES DE RECOMENDACIONES DEL UIT-T

Serie A	Organización del trabajo del UIT-T
Serie B	Medios de expresión: definiciones, símbolos, clasificación
Serie C	Estadísticas generales de telecomunicaciones
Serie D	Principios generales de tarificación
Serie E	Explotación general de la red, servicio telefónico, explotación del servicio y factores humanos
Serie F	Servicios de telecomunicación no telefónicos
Serie G	Sistemas y medios de transmisión, sistemas y redes digitales
Serie H	Sistemas audiovisuales y multimedios
Serie I	Red digital de servicios integrados
Serie J	Transmisiones de señales radiofónicas, de televisión y de otras señales multimedios
Serie K	Protección contra las interferencias
Serie L	Construcción, instalación y protección de los cables y otros elementos de planta exterior
Serie M	RGT y mantenimiento de redes: sistemas de transmisión, circuitos telefónicos, telegrafía, facsímil y circuitos arrendados internacionales
Serie N	Mantenimiento: circuitos internacionales para transmisiones radiofónicas y de televisión
Serie O	Especificaciones de los aparatos de medida
Serie P	Calidad de transmisión telefónica, instalaciones telefónicas y redes locales
Serie Q	Conmutación y señalización
Serie R	Transmisión telegráfica
Serie S	Equipos terminales para servicios de telegrafía
Serie T	Terminales para servicios de telemática
Serie U	Conmutación telegráfica
Serie V	Comunicación de datos por la red telefónica
Serie X	Redes de datos y comunicación entre sistemas abiertos
Serie Y	Infraestructura mundial de la información y aspectos del protocolo Internet
Serie Z	**Lenguajes y aspectos generales de soporte lógico para sistemas de telecomunicación**

Impreso en Suiza
Ginebra, 2008