UNIÓN INTERNACIONAL DE TELECOMUNICACIONES

UIT-T

SECTOR DE NORMALIZACIÓN
DE LAS TELECOMUNICACIONES
DE LA UIT

Z.100

Anexo F1
(11/2000)

SERIE Z: LENGUAJES Y ASPECTOS GENERALES DE SOPORTE LÓGICO PARA SISTEMAS DE TELECOMUNICACIÓN

Técnicas de descripción formal – Lenguaje de especificación y descripción

---

Lenguaje de especificación y descripción

Anexo F1: Definición formal del lenguaje de especificación y descripción: Visión general

Recomendación UIT-T Z.100 – Anexo F1

RECOMENDACIONES UIT-T DE LA SERIE Z

LENGUAJES Y ASPECTOS GENERALES DE SOPORTE LÓGICO PARA SISTEMAS DE TELECOMUNICACIÓN

TÉCNICAS DE DESCRIPCIÓN FORMAL	
Lenguaje de especificación y descripción	**Z.100–Z.109**
Aplicación de técnicas de descripción formal	Z.110–Z.119
Gráficos de secuencias de mensajes	Z.120–Z.129
LENGUAJES DE PROGRAMACIÓN	
CHILL: el lenguaje de programación del UIT-T	Z.200–Z.209
LENGUAJE HOMBRE-MÁQUINA	
Principios generales	Z.300–Z.309
Sintaxis básica y procedimientos de diálogo	Z.310–Z.319
LHM ampliado para terminales con pantalla de visualización	Z.320–Z.329
Especificación de la interfaz hombre-máquina	Z.330–Z.399
CALIDAD DE SOPORTES LÓGICOS DE TELECOMUNICACIONES	Z.400–Z.499
MÉTODOS PARA VALIDACIÓN Y PRUEBAS	Z.500–Z.599

Para más información, véase la Lista de Recomendaciones del UIT-T.

Recomendación UIT-T Z.100

Lenguaje de especificación y descripción

ANEXO F1

Definición formal del lenguaje de especificación y descripción: Visión general

Resumen

Este anexo F1 establece la motivación, describe la estructura general de la semántica formal y contiene una introducción a la máquina de estados abstractos (ASM, *abstract state machine*), formalismo que es usado para definir la semántica SDL.

Orígenes

El anexo F1 a la Recomendación UIT-T Z.100, revisado por la Comisión de Estudio 10 (2001-2004) del UIT-T, fue aprobado por el procedimiento de la Resolución 1 de la AMNT el 24 de noviembre de 2000.

PREFACIO

La UIT (Unión Internacional de Telecomunicaciones) es el organismo especializado de las Naciones Unidas en el campo de las telecomunicaciones. El UIT-T (Sector de Normalización de las Telecomunicaciones de la UIT) es un órgano permanente de la 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 Asamblea Mundial de Normalización de las Telecomunicaciones (AMNT), 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 1 de la AMNT.

En ciertos sectores de la tecnología de la información que corresponden a la esfera de competencia del UIT-T, se preparan las normas necesarias en colaboración con la ISO y la CEI.

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.

PROPIEDAD INTELECTUAL

La UIT señala a la atención la posibilidad de que la utilización o aplicación de la presente Recomendación suponga el empleo de un derecho de propiedad intelectual reivindicado. La UIT no adopta ninguna posición en cuanto a la demostración, validez o aplicabilidad de los derechos de propiedad intelectual reivindicados, ya sea por los miembros de la UIT o por terceros ajenos al proceso de elaboración de Recomendaciones.

En la fecha de aprobación de la presente Recomendación, la UIT no ha recibido notificación de propiedad intelectual, protegida por patente, que puede ser necesaria para aplicar esta Recomendación. Sin embargo, debe señalarse a los usuarios que puede que esta información no se encuentre totalmente actualizada al respecto, por lo que se les insta encarecidamente a consultar la base de datos sobre patentes de la TSB.

© UIT 2002

Es propiedad. Ninguna parte de esta publicación puede reproducirse o utilizarse, de ninguna forma o por ningún medio, sea éste electrónico o mecánico, de fotocopia o de microfilm, sin previa autorización escrita por parte de la UIT.

ÍNDICE

Page

1 Prefacio ... 1
1.1 Motivación ... 1
1.2 Objetivos principales ... 1
1.3 Referencias ... 2
1.4 Bibliografía ... 2
2 Visión general de la semántica ... 3
2.1 Gramática ... 3
2.2 Condiciones de formación correcta ... 4
2.3 Reglas de transformación ... 4
2.4 Semántica dinámica ... 4
3 Maquinas de estados abstractos ... 6
3.1 Modelo ASM básico ... 6
3.1.1 Vocabulario ... 6
3.1.2 Estados ... 8
3.1.3 Nombres derivados ... 8
3.1.4 Estados iniciales ... 9
3.1.5 Transiciones de estado y ejecuciones ... 9
3.1.6 Reglas de transición ... 10
3.1.7 Abreviaturas ... 12
3.1.8 Programas de ASM ... 13
3.2 ASM distribuida ... 14
3.2.1 Vocabulario ... 14
3.2.2 Agentes y ejecuciones ... 15
3.2.3 Programas de ASM distribuidas ... 16
3.3 El mundo externo ... 17
3.4 Comportamiento en tiempo real ... 19
3.5 Ejemplo: el sistema RMS ... 19
3.6 Nombres definidos previamente ... 20

Recomendación UIT-T Z.100

Lenguaje de especificación y descripción

ANEXO F1

Definición formal del lenguaje de especificación y descripción: Visión general

1 Prefacio

La presente definición formal del lenguaje de especificación y descripción (SDL, *specification and description languages*) proporciona una definición del lenguaje que complementa la definición del texto de la Recomendación. Este anexo está destinado a aquellos que necesitan una definición muy detallada y precisa del SDL como por ejemplo los encargados de mantener el lenguaje SDL, los diseñadores de instrumentos SDL y los usuarios del lenguaje SDL.

La definición formal está constituida por tres anexos:

Anexo F1 Este anexo establece la motivación, describe la estructura general de la semántica formal y contiene una introducción a la *máquina de estados abstractos* (ASM, *abstract state machine*), formalismo que es usado para definir la semántica SDL.

Anexo F2 Este anexo describe las limitaciones y las transformaciones semánticas estáticas, tal como se indica en las secciones Modelo de UIT-T Z.100.

Anexo F3 Define la semántica dinámica del SDL.

1.1 Motivación

Por lo general, los lenguajes naturales son ambiguos, es decir, pueden tener más de una interpretación. Una especificación es formal si su sentido (semántico) no es ambiguo. Para ello, se han desarrollado lenguajes especiales, conocidos como técnicas de descripción formal (FDT, *formal description techniques*). Las FDT se diferencian de los lenguajes formales en general, por el hecho de que tienen una sintaxis formal y una semántica formal. Esto difiere de la mayoría de los lenguajes tales como Java o C++, que tienen solamente una sintaxis formal.

La semántica formal de un lenguaje se define en términos de un formalismo matemático subyacente, es decir, de modo axiomático o funcional. La selección de un formalismo apropiado está influenciada por la expresividad de las FDT, así como por los objetivos inequívocos de la semántica. La semántica formal define, para cada especificación, un modelo matemático correspondiente que recoge su significado de manera precisa y completa.

Los anexos F1, F2 y F3 definen formalmente la semántica del SDL. Si hay alguna incoherencia entre el texto principal de UIT-T Z.100 y estos anexos F, es porque existe un error que debe corregirse. En tal caso ninguno de los dos textos tiene prelación sobre el otro.

1.2 Objetivos principales

Un objetivo fundamental de una semántica formal SDL es su inteligibilidad, requisito previo para que sea correcta, aceptada y conservada. La inteligibilidad se sustenta mediante la construcción de formalismos y notaciones matemáticos bien conocidos, una estrecha correspondencia entre la especificación técnica y la semántica que debe ser formalizada y una documentación concisa y bien estructurada.

La conservación es otro objetivo importante, ya que el SDL es una norma técnica evolutiva. Además de las ampliaciones del lenguaje que habrán de ser incorporadas en esta Recomendación, se están considerando otras características del lenguaje, tales como la expresividad en tiempo real. Por consiguiente, el formalismo matemático tendrá que ser lo suficientemente rico y flexible como para que la semántica formal pueda ser adaptada y ampliada con un esfuerzo razonable.

El SDL se puede clasificar como una FDT basada en un modelo para la especificación de sistemas distribuidos y concurrentes, lo que significa que una especificación SDL define explícitamente un conjunto de cálculos. Para ello hace falta una semántica operativa que permita obtener una estrecha correspondencia con la especificación y mejorar por tanto su inteligibilidad. Además, debido al número de instrumentos disponibles, la semántica operativa ofrece de forma natural una capacidad de ejecución, lo que constituye otro objetivo explícito.

1.3 Referencias

– Recomendación UIT-T Z.100 (1999), *Lenguaje de especificación y descripción*.

1.4 Bibliografía

[1] `http://www.uni-paderborn.de/cs/asm/` and `http://www.eecs.umich.edu/gasm/`.

[2] ESCHBACH (R.), GLÄSSER (U.), GOTZHEIN (R.), PRINZ (A.): On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In *Abstract State Machines: Theory and Applications*. Y. Gurevich, P.W. Kutter, M. Odersky and L. Thiele, editors, vol. 1912 of LNCS, Springer-Verlag, 2000.

[3] GUREVICH (Y.): Evolving Algebra 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger, editor, pages 9-36, Oxford University Press, 1995.

[4] GUREVICH (Y.): ASM Guide 97, *CSE Technical Report CSE-TR-336-97*, EECS Department, University of Michigan-Ann Arbor, 1997.

2 Visión general de la semántica

Para poder definir la semántica formal del SDL, se descompone la definición del lenguaje en varias partes:

- la gramática;
- las condiciones de formación correcta;
- las reglas de transformación; y
- la semántica dinámica.

El punto de inicio para definir la semántica formal del SDL es una especificación SDL sintácticamente correcta, representada como un árbol de sintaxis abstracta (AST, *abstract syntax tree*).

En el contexto del SDL (véase figura F1-1), las primeras tres partes de la semántica formal son denominadas de manera colectiva *semántica estática* o *aspectos estáticos*.

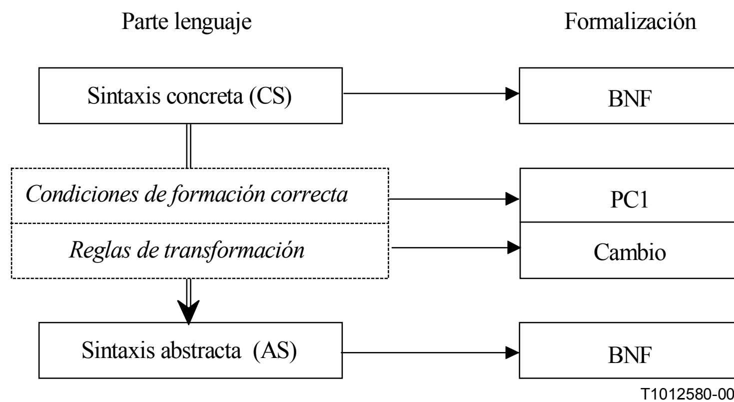


Figura F1-1/Z.100 – Aspectos estáticos del SDL

La *gramática* define un conjunto de especificaciones SDL sintácticamente correctas. En UIT-T Z.100 se definen formalmente una gramática textual concreta, una gramática gráfica concreta, y una gramática abstracta, utilizando la forma Backus-Naur (BNF) con extensiones para poder incorporar las construcciones del lenguaje gráfico. La gramática abstracta se obtiene a partir de las gramáticas concretas eliminando los detalles irrelevantes, tales como los separadores y las reglas léxicas, y aplicando las reglas de transformación (véase a continuación).

Las *condiciones de formación correcta* definen que especificaciones, correctas con respecto a la gramática, lo son también en relación con la información del contexto, tal como los nombres está permitido utilizar en determinado lugar o el tipo de valores que está permitido asignar a las variables. Las condiciones de formación correcta se definen en forma de cálculos de predicados de primer orden (PC1, *first order predicate calculus*).

Además, los constructivos de algunos lenguajes que aparecen en las gramáticas concretas son reemplazados por otros elementos del lenguaje en las gramáticas abstractas utilizando *reglas de transformación* (*transformation rules*) para que el número de conceptos fundamentales siga siendo reducido. Estas transformaciones se describen en los párrafos de modelo de UIT-T Z.100 y se denominan formalmente reglas de cambio.

La *semántica dinámica* sólo se aplica a las especificaciones SDL sintácticamente correctas que satisfacen las condiciones de formación correcta. La semántica dinámica define un conjunto de cálculos asociados con una especificación.

2.1 Gramática

La *gramática* de SDL se formaliza utilizando una representación de la gramática en forma Backus-Naur (BNF, *Backus-Naur form*). Sin embargo, en UIT-T Z.100 la gramática está concebida de modo que sea una gramática de presentación, es decir, no ha sido diseñada para generar automáticamente un análisis sintáctico. Además, algunas restricciones que garantizan finalmente la unicidad de la semántica no pueden ser expresadas en BNF y, por ello, han sido enunciadas en el texto. La gramática, por tanto, define utilizando BNF y algún texto (en particular para las reglas de precedencia). La conversión de la representación SDL textual concreta en representación de la sintaxis abstracta de Z.100 (llamada AS1) se efectúa en dos pasos. El primer paso, de la representación SDL textual concreta a la sintaxis abstracta AS0, no ha sido definido formalmente, pero se deriva de la correspondencia entre las dos gramáticas, que es casi biunívoca y elimina los detalles irrelevantes, tales como los separadores y las reglas léxicas. El segundo paso, que convierte AS0 en AS1, es introducido formalmente por un conjunto de reglas de transformación (véase el anexo F2).

2.2 Condiciones de formación correcta

Las *condiciones de formación correcta* definen las limitaciones suplementarias a las que una especificación debe atenerse. Estas limitaciones no pueden ser expresadas en BNF, pero son estáticas, es decir, pueden ser definidas y verificadas con independencia de la definición semántica dinámica (véase el anexo F2). Una especificación SDL es *válida* solamente si satisface las reglas sintácticas y las condiciones estáticas de SDL. En realidad, las condiciones de formación correcta se refieren a la sintaxis, pero no han sido enunciadas en la sintaxis concreta porque no pueden ser expresadas en una gramática sin contexto.

Básicamente, existen cinco tipos de condiciones de formación correcta :

- *Reglas de ámbito y visibilidad*: La definición de una entidad introduce un identificador que puede ser utilizado como referencia a esa entidad. Solamente se deben usar identificadores visibles. Las reglas de ámbito y visibilidad se aplican para determinar si la definición correspondiente de un identificador es visible o no.
- *Reglas de eliminación de ambigüedades*: Algunas veces, un nombre puede referirse a varios identificadores. Se aplican reglas para discernir el nombre correcto.
- *Reglas de consistencia del tipo de datos*: Estas reglas garantizan que, desde el punto de vista dinámico, no se aplica ninguna operación a los operadores que no corresponda a sus tipos de argumentos. Más específicamente, el tipo de datos de un parámetro concreto debe ser compatible con el del parámetro formal correspondiente; el tipo de datos de una expresión debe ser compatible con el de la variable a la que ha sido asignada la expresión.
- *Reglas especiales*: Existen algunas reglas aplicables a entidades específicas. Por ejemplo, que haga bloques locales o un gráfico dentro de un bloque.
- Re*glas de sintaxis simple*: Existen algunas reglas que se refieren a la corrección de la sintaxis concreta, y que no tienen su equivalente en la sintaxis abstracta. Por ejemplo, en una definición, el nombre del inicio debe concordar con el nombre del fin.

2.3 Reglas de transformación

Para un lenguaje con una sintaxis rica, es importante identificar los conceptos básicos que concuerdan con las intenciones del diseñador del lenguaje. Otros constructivos del lenguaje que son introducidos por conveniencia, pero que no enriquecen la expresividad del lenguaje (tales como notaciones abreviadas), pueden ser reemplazados utilizando estos conceptos básicos. Dado que las sustituciones, que se describen mediante reglas de transformación, pueden ser formalizadas, basta con definir solamente la semántica dinámica para los conceptos básicos, lo que los hace más concisos y más inteligibles.

Para la definición de la semántica formal, es fundamental la elección de los conceptos básicos "adecuados". Si hay demasiados conceptos básicos, la semántica dinámica será menos concisa y menos inteligible. Si hay demasiados pocos o si existen conceptos erróneos, las transformaciones tienden a ser muy complejas. En 1992 se introdujo, en el ámbito de SDL, la orientación al objeto, pero aun así, la definición semántica (formal) y la sintaxis abstracta se basaban en ejemplos y no tenían ninguna noción de clase. El resultado fue una transformación muy engorrosa, que ha sido rectificada en esta definición semántica formal.

UIT-T Z.100 prescribe la transformación de las especificaciones SDL por una secuencia de *pasos de transformación (transformation steps)*. Cada paso de transformación consiste en un conjunto de transformaciones simples, tal como se indica en las cláusulas relativas al modelo, y determina cómo se debe tratar una clase especial de notaciones abreviadas. El resultado de un paso se utiliza como entrada al paso siguiente.

Para formalizar las reglas de transformación de UIT-T Z.100, se utilizan reglas de cambio. Una transformación simple se realiza aplicando una regla de cambio a una especificación concreta, que esencialmente significa reemplazar partes de la especificación por otras partes, tal como define la regla (véase el anexo F2).

2.4 Semántica dinámica

La *semántica dinámica* (anexo F3) consta de siguientes partes (véase la figura F1-2) :

- La *máquina de estados abstractos SDL,* (SAM, *SDL abtract machine*), que se define utilizando ASM. La definición de la SAM se divide en tres partes, en correspondencia con la sintaxis abstracta:
  1) conceptos básicos de flujo de señales (señales, temporizadores, excepciones, puertas, canales),
  2) varios tipos de agente ASM (que modelan agentes SDL correspondientes); y
  3) primitivas de comportamiento (instrucciones SAM).
- La *recopilación*, definida como una función en el árbol de la sintaxis abstracta de una especificación SDL. Los resultados de la recopilación son conjuntos de primitivas de comportamiento, que modelan las acciones de los agentes SDL.

- La *inicialización*, que define el estado previo al inicio de un sistema y varios programas de inicialización. A continuación, se alcanza el estado de sistema inicial creando un agente de sistema SDL y activando este agente en el estado previo al inicio. La inicialización despliega recursivamente la estructura estática del sistema, creando más agentes SDL que los especificados. En realidad, cada vez que se crean los agentes SDL, se inicia el mismo proceso en la fase de ejecución subsiguiente. Desde este punto de vista, la inicialización describe simplemente la instanciación del agente de sistema SDL.
- La *semántica de los datos* que está separada del resto de la semántica por una interfaz. En este lugar, el uso de la interfaz es intencional. Permitirá conmutar el modelo de datos si para algún dominio es más apropiado otro modelo de datos que el modelo integrado. Además, el modelo de datos integrado también puede ser cambiado de esta manera sin afectar al resto de la semántica.

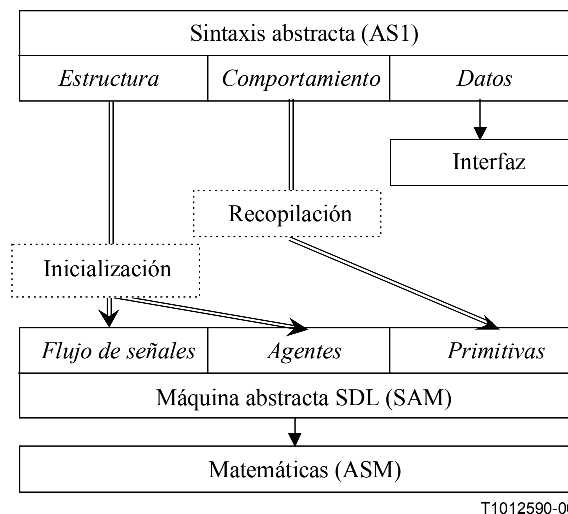


Figura F1-2/Z.100 – Visión general de la semántica dinámica

La semántica dinámica se formaliza a partir de la sintaxis abstracta AS1 de SDL. De esta sintaxis abstracta se deriva un modelo de comportamiento para las especificaciones SDL. El enfoque aquí elegido se basa en una visión de funcionamiento abstracto que utiliza el formalismo ASM como estructura matemática subyacente para una definición semántica rigurosa del modelo SAM. La recopilación define un recopilador abstracto que hace corresponder las partes de comportamiento del SDL con el código abstracto (semántica indicativa). Finalmente, la inicialización describe una interpretación del árbol de sintaxis abstracta para construir la estructura inicial del sistema (semántica operativa).

La *semántica dinámica* asocia con cada especificación SDL, una ASM particular distribuida en tiempo real. De manera intuitiva, esto consiste en un conjunto de agentes autónomos que cooperan en la realización de ejecuciones simultáneas de la máquina. El comportamiento de un agente está determinado por un programa ASM que consiste en una regla de transición. Colectivamente, estas reglas definen un conjunto de posibles ejecuciones de máquina. Cada agente tiene su propia visión parcial del estado global, definido por un conjunto de funciones y dominios estáticos y dinámicos. La acción entre los agentes puede ser modelada haciendo que haya intersecciones no vacías de visiones parciales. En la cláusula 3 se presenta una introducción al modelo ASM y a la notación utilizada en los anexos F1, F2 y F3.

3 Maquinas de estados abstractos

Esta cláusula explica las nociones y conceptos básicos de las *máquinas de estados abstractos (ASM)* así como la notación utilizada en este anexo para definir el modelo de máquina abstracta SDL. El objetivo aquí es proporcionar una comprensión intuitiva del formalismo; para obtener una definición rigurosa de los fundamentos matemáticos de la ASM, el lector debe referirse a [3] y [4]. En [2] se presenta un análisis de la conveniencia del marco semántico aquí utilizado así como los motivos que la justifican. En las páginas Web ASM [1] podrá también encontrar otro material referente a la ASM.

El modelo ASM utilizado para definir la semántica dinámica del SDL se explica en varias etapas. Primero se trata el *modelo ASM básico* con un único agente (cláusula 3.1). A continuación se amplia este modelo para cubrir los *sistemas con múltiples agentes (multi-agent systems)* (cláusula 3.2). Seguidamente se tratan los *sistemas abiertos* (*open systems*), es decir los sistemas que interactúan con un entorno que no pueden controlar, agregando la noción de *mundo externo (external world)* (cláusula 3.3). Por último, el modelo se amplia utilizando la noción de *comportamiento en tiempo real (real-time behaviour)* (cláusula 3.4). Para representar estos pasos, se desarrolla de manera sucesiva un modelo ASM para un sistema simple. En 3.5 se describe el modelo ASM final de este sistema. En 3.6 se resume la notación adicional utilizada para definir la semántica dinámica de SDL.

Ejemplo (descripción oficiosa):

Para describir el modelo ASM, se define un sistema de gestión de recursos (RMS, *resource management system*) sencillo compuesto por un *grupo de agentes, $n > 1$*, que compiten por *un recurso* (por ejemplo, algún dispositivo o servicio). De manera oficiosa, este sistema está caracterizado de la manera siguiente:

- Existe un conjunto de *testigos (tokens), $m < n$*, utilizados para hacer posible un acceso *exclusivo o no exclusivo (compartido)* al recurso.
- Según que el modo de acceso deseado sea exclusivo o compartido, un agente debe poseer todos los testigos o un testigo respectivamente, antes de poder acceder al recurso.
- Un agente está *en reposo* (*idle*) cuando no compite por un recurso, está *esperando* tratando de obtener el acceso al recurso, o está *ocupado* cuando tiene derecho a acceder al recurso.
- Una vez que el agente está en situación de espera (*waiting*) , continúa en la misma hasta que obtiene el acceso al recurso.
- Un agente ocupado libera el recurso cuando ya no lo necesita, lo que se indica mediante una *condición de parada (stop condition)* externamente para ese agente. Al liberar el recurso, se devuelven todos los testigos que posee el agente.
- Las condiciones de parada sólo se indican cuando un agente está ocupado. Existe una *restricción de integridad* (*integrity constraint*) impuesta al comportamiento del mundo externo.
- Inicialmente, todos los agentes están en reposo y todos los testigos están disponibles.

El sistema se definirá paso a paso, según se vayan presentando las explicaciones del modelo ASM, comenzando por el modelo ASM básico con un único agente. En 3.5 se resume el modelo ASM final de este sistema.

3.1 Modelo ASM básico

Un *Abstract State Machine M* (máquina de estados abstractos M) con un *vocabulary V* (*vocabulario V*) dado por sus *states S (estados S)*, sus *initial states* (estados iniciales) $S_0 \subseteq S$, y su *program (programa) P.* En las cláusulas siguientes se explican estos elementos.

3.1.1 Vocabulario

El vocabulario (o signatura) *V* denota un conjunto finito de *function names (nombres de función)* , *predicate names (nombres de predicado)*, y *domain names (nombres de dominio)*, cada uno con una dimensión fija. Los nombres de *V* están clasificados como *basic* (*básicos*) o *derived (derivados)* , y además se dividen en *static* (*estáticos*) o *dynamic (dinámicos)* (véase la figura F1-3). En las cláusulas siguientes se explica el significado asociado a estas clasificaciones.

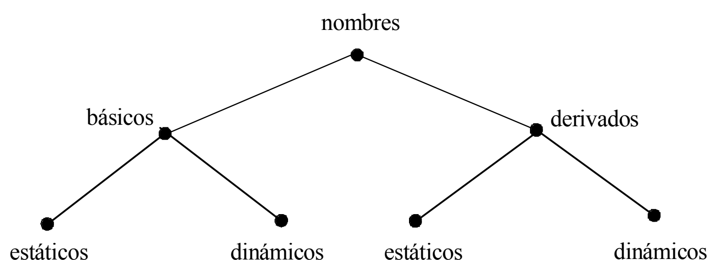


Figura F1-3/Z.100 – Clasificación de los nombres ASM

Se declara *V* cuando se está definiendo una ASM, excepto para un subconjunto de *predefined names (nombres predefinidos)*. Este subconjunto contiene, por ejemplo, un signo de igualdad, los nombres *True*, *False* (*Verdadero, Falso*) del predicario 0-ario, el nombre de la función 0-aria (función constante) *undefined* (*no definido*), y los nombres de dominio *BOOLEAN*, *NAT* y *REAL*, así como los nombres de las funciones estándar utilizadas frecuentemente, tales como las operaciones booleanas $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\Leftrightarrow$, y las operaciones de conjunto $\subseteq$, $\cup$, $\cap$, $\in$, $\notin$, etc.). En 3.6 figura una relación de nombres predefinidos.

Para *declare (declarar)* los nombres cuando se define una ASM concreta se utilizan los convenios siguientes de notación:

- Los *nombres de dominio* se escriben en cursivas mayúsculas (como en *AGENT (AGENTE)*), excepto cuando denotan un no terminal de la gramática abstracta. En este caso, los nombres de dominio se escriben como no terminales, es decir en cursivas, con guión y comenzando por una mayúscula (como en *AgentDefinition*). Un nombre de dominio *D* se declara mediante **domain** *D*.
- Los *nombres de función* se escriben en cursivas comenzando por una letra minúscula (como en *mode (modo)*). Un nombre de función *f* se declara mediante f: $D_1 \times D_2 \times ... \times D_n \rightarrow D_0$, en donde *n* es la dimensión de *f*, y $D_0, D_1, D_2, ..., D_n$ son los nombres de dominio.
- Los *nombres de* predicado también se escriben en cursivas pero comienzan con una letra mayúscula (como en *Available*). Un nombre de predicado *P* se declara mediante P: $D_1 \times D_2 \times ... \times D_N \rightarrow$ *BOOLEAN*.
- Los *nombres estáticos básicos* se califican mediante la palabra clave **static**, cuando se declaran (véase la figura F1-3).
- Los *nombres dinámicos* básicos se califican mediante una de las palabras clave **controlled** (controlado), **shared** (compartido), o **monitored** (supervisado), cuando se declaran (tal como se explica en 3.3).
- Los nombres que no están precedidos por una palabra clave son *nombres derivados* por defecto (véase la figura F1-3).

Ejemplo (vocabulario):

Para definir un modelo ASM del sistema *RMS*, supóngase un vocabulario *V* que incluya los nombres siguientes:

static domain *AGENT*
static domain *TOKEN*
domain *MODE*

shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*
static *ag*: → *AGENT*

Idle: *AGENT* → *BOOLEAN*
Waiting: *AGENT* → *BOOLEAN*
Busy: *AGENT* → *BOOLEAN*
Available: *TOKEN* → *BOOLEAN*

monitored *Stop*: *AGENT* → *BOOLEAN*

Los nombres del dominio estático *AGENT*, *TOKEN*, y *MODE* se introducen para representar el (único) agente del sistema, el conjunto de testigos y, respectivamente, los diferentes modos de acceso (exclusive, shared). Los nombres *mode* y *owner* indican funciones dinámicas, y se usan para modelar, respectivamente, el modo respectivamente, el modo de acceso corriente de un agente y el propietario corriente de un testigo. El nombre *ag* de la función 0-aria se refiere a un valor del dominio *AGENT*. *Idle*, *Waiting*, *Busy*, y *Available* son nombres de predicado dinámico derivado. *Stop* representa un predicado supervisado, que se explicará más adelante.

3.1.2 Estados

Se da un estado $s \in S$ asignando un significado, también llamado *interpretation*, a los nombres de *V* en un conjunto infinito llamado *base set of M* (conjunto de base de M) (al que se hará referencia por el nombre del dominio predefinido *X*)[1]. Es decir, que a cada nombre de dominio, a cada nombre de función y a cada nombre de predicado de *V*, se le debe asociar respectivamente un dominio, una función o un predicado básicos. La interpretación de nombres derivados es consecuencia de la interpretación de nombres básicos. Se señala que el conjunto de base es el mismo para todos los estados de *M*. Es preciso que *True*, *False* y *undefined* representen distintos elementos del conjunto de base. Las operaciones predefinidas tienen su interpretación habitual.

Hay que recordar que los nombres se clasifican como estáticos o dinámicos. Si están clasificados como estáticos se requiere que los nombres tengan la misma interpretación en todos los estados de *M*. Si no es así, podrán tener diferentes interpretaciones en diferentes estados de *M*. Así pues, los estados *S* de *M* vienen dados por el conjunto de todas las interpretaciones de los nombres de *V* en el conjunto de base de *M* que cumplen con éstas y otras limitaciones enunciadas explícitamente.

En sentido estricto, todas las funciones son funciones *total* en el conjunto de base de *M*. Para imitar las *partial functions (funciones parciales)*, los valores de la función "no definida" se marcan con el elemento distintivo *undefined*. Los predicados sólo generan uno de los valores *True* o *False*, es decir, no deben ser parciales.

Cada estado tiene, potencialmente, un número infinito de *reserve elements (elementos de reserva* que permiten extender dinámicamente los dominios (véase 3.1.6.). Por definición los elementos de reservas de un estado son los elementos del conjunto de base que no están ni identificados por una función ni contenidos en uno de los dominios.

3.1.3 Nombres derivados

El significado de los nombres derivados proviene de la interpretación de los nombres básicos y se define en términos de *formulae* (véase 3.6); los nombres derivados pueden interpretarse, por tanto, como abreviaturas. Supóngase que un *DerivedName* es un nombre n-ario y que la $Formula(v_1,...,v_n)$ representa una fórmula del dominio *D* con variables libres $v_1,...,v_n$ de los dominios $D_1,...,D_n$, $n \geq 0$. La forma general de una *derived name definition* es:

$$DerivedNameDefinition ::= DerivedName(v_1{:}D_1,...,v_n{:}D_n){:}D =_{\text{def}} Formula(v_1,...,v_n)$$

El resultado del dominio *D* se omite en el caso de una definición de dominio derivado.

[1] En sentido estricto, los estados ASM son (*many-sorted*) *first-order structures,* es decir, estructuras de primer orden (con clasificación múltiple).

Ejemplo (definiciones):

Se definen los predicados derivados siguientes para referirse al estatus de un agent/token en un estado dado:

$MODE =_{def} \{exclusive, shared\}$

$Idle(a{:}AGENT)$: BOOLEAN	$=_{def}$	$a.mode = undefined \wedge \forall t \in TOKEN: t.owner \neq a$
$Waiting(a{:}AGENT)$: BOOLEAN	$=_{def}$	$a.mode \neq undefined \wedge \forall t \in TOKEN: t.owner \neq a$
$Busy(a{:}AGENT)$: BOOLEAN	$=_{def}$	$a.mode \neq undefined \wedge \exists t \in TOKEN: t.owner = a$
$Available(t{:}TOKEN)$: BOOLEAN	$=_{def}$	$t.owner = undefined$

Por ejemplo, un agente *a* está en reposo si la función *mode* da el valor *undefined* para ese agente, y *a* no contiene ningún testigo. Un testigo *t* está disponible si ningún agente contiene *t*.

Para facilitar la lectura, se utiliza una *notation "."* para las funciones y predicados monarios. Por ejemplo, se escribe *a.mode*, que equivale a escribir *mode*(*a*).

3.1.4 Estados iniciales

El conjunto de *initial states* $S_0 \subseteq S$ se define por las limitaciones impuestas a los dominios, funciones y predicados asociados con los nombres de *V*. Las limitaciones iniciales para las operaciones y los ámbitos predefinidos son dadas implícitamente; véase 3.6. Las limitaciones iniciales tienen la forma general siguiente:

initially *ClosedFormula*

Ejemplo (estados iniciales):

Las limitaciones siguientes definen el conjunto de estados iniciales del sistema *RMS*:

initially $AGENT = \{ag\}$
initially $\forall a \in AGENT: a.Idle \wedge \forall t \in TOKEN: t.Available$

La primera limitación define el conjunto inicial *AGENT* que está compuesto por un único elemento *ag*. La segunda limitación indica que inicialmente el agente de *RMS* está en reposo (*a.mode = undefined*), y que todos los testigos están disponibles (*t.owner = undefined*). Se señala que no se define ninguna limitación en *Stop*.

3.1.5 Transiciones de estado y ejecuciones

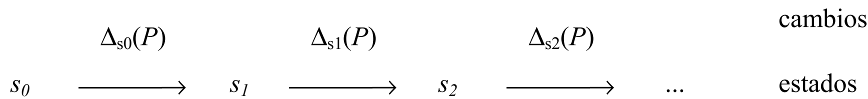
Hay que recordar que un estado (global) $s \in S$ viene dado por una interpretación de los nombres de *V* en el conjunto de base de *M*. Se pueden definir las transiciones de estado en forma de nuevas interpretaciones parciales de dominios, funciones y predicados dinámicos. Esto introduce la noción de *location* como término conceptual para referirse a partes de los estados globales y a la noción de *update* para describir los cambios de estado.

Una *location of a state s of M* es una $loc_S = \langle f, s(x) \rangle$ par, en donde *f* es un nombre dinámico de *V*, y *s*(*x*) es una secuencia de elementos del conjunto de base según la dimensión de *f*. Una *update of s* es una $\delta_s = \langle loc_s, s(y) \rangle$ par, en donde *s*(*y*) identifica un elemento del conjunto de base como el nuevo valor que debe ser asociado con la posición loc_s. *Disparar* δ_S significa transformar *s* en un estado *s'* de *M* tal como $f_{s'}(s(x)) = s(y)$, mientras que las otras posiciones loc'_s de *s*, $loc'_s \neq loc_s$, no son afectadas. En otras palabras, el disparo de una actualización modifica la interpretación de un estado en un modo bien definido.

El comportamiento potencial de una ASM básica se incorpora mediante un *program P*, definido por una *transition rule* (véanse 3.1.6. y 3.1.8.). Para cada estado $s \in S$, un programa *P* de *M* define un *update set* $\Delta_s(P)$ como un conjunto finito de actualizaciones de *s*. $\Delta_s(P)$ es *consistent*, únicamente si no contiene ninguna de las dos actualizaciones δ_s, δ'_s tales como $\delta_s = \langle loc_s, s(y) \rangle$, $\delta'_s = \langle loc_s, s(y') \rangle$, y $s(y) \neq s(y')$. El *firing of a consistent update set* $\Delta_s(P)$ en el estado *s,* significa disparar simultáneamente todos sus miembros, es decir, producir (en un paso atómico) un nuevo estado *s'* de ese tipo, llamado *state transition*, para todas las posiciones $loc_s = \langle f, s(x) \rangle$ de *s*, $f_{s'}(s(x)) = s(y)$, si $\langle\langle f, s(x) \rangle, s(y) \rangle \in \Delta_s(P)$, y $f_{s'}(s(x)) = f_s(s(x))$, de no ser así. Disparar un conjunto de actualización inconsistente[2] no tiene efecto, es decir, $s' = s$.

[2] En el contexto de la semántica SDL, un conjunto de actualización inconsistente indica un error en el modelo semántico. La semántica ASM garantiza que esos errores no destruyen la noción de estado.

El comportamiento de un único agente M de ASM se modela mediante *runs of M* , (finitas o infinitas), en donde una *run* es una secuencia de transiciones de estado de la forma



tal que $s_0 \in S_0$, y s_{i+1} se obtiene de s_i, para $i \geq 0$, disparando $\Delta_{si}(P)$ en s_i, en donde $\Delta_{si}(P)$ denota un conjunto actualizado definido por el programa P de M en s_i (véase 3.1.8.). El significado de una ASM se define como el conjunto de todas sus ejecuciones. La atención se centra a continuación, en las ejecuciones que comienzan en un estado inicial, también llamadas *regular runs* (ejecuciones periódicas).

3.1.6 Reglas de transición

Las reglas de transición especifican conjuntos actualizados en los estados de la ASM. Las reglas complejas se forman a partir de reglas elementales utilizando diversos constructores de reglas. A la forma elemental de una regla de transición se le llama *update instruction* (*instrucción de actualización*).

- *update instruction*

  Rule ::= $f(t_0,...,t_n) := t_0$ $(n \geq 0)$

  En este caso, f es un nombre no estático de V que representa tanto una función como un predicado o un dominio, controlados o compartidos, y $t_0,t_1,...,t_n$ son términos de V que identifican, respectivamente, para un estado dado s, la posición $loc = \langle f,\langle s(t_1),..., s(t_n)\rangle\rangle$ que debe ser cambiada y el nuevo valor $s(t_0)$ que debe ser asignado. En otras palabras, la instrucción de actualización anterior especifica el conjunto de actualización $\{\langle\langle f,\langle s(t_1),..., s(t_n)\rangle\rangle, s(t_0)\rangle\}$, que consiste en una sola actualización. Se señala que, en el lado izquierdo de una instrucción de actualización, sólo pueden producirse posiciones vinculadas a nombres básicos (no estáticos).

> **Ejemplo (instrucción de actualización):**
>
> Sea t una variable que representa un testigo y ag un agente.
>
> *t.owner* := *ag* specifies the update set {<<*owner*, <*s*(*t*)>>, *s*(*ag*)>}
> *ag.mode* := *undefined* specifies the update set {<<*mode*, <*s*(*ag*)>>, *s*(*undefined*)>}

La construcción de reglas de transición complejas a partir de instrucciones de actualización elementales, se define de manera recurrente por medio de *ASM rule constructors*. Para el modelo ASM aplicado para definir la semántica SDL se utilizan seis constructores diferentes. A continuación se indican estos constructores, junto con una descripción oficiosa de su significado. Aquí, *Rule*, $Rule_i$ denotan reglas de transición, g denota un término booleano, y $v,v_1,...,v_n$ denotan variables libres en el conjunto de base de M. El alcance de un constructor de reglas lo expresan las palabras clave apropiadas y puede estar indicado además por un sangrado. Se puede omitir el cierre de las palabras clave, siempre que ello no produzca confusión. Si se omite el cierre de las palabras clave, el constructor correspondiente se amplía tanto como sea posible, pero no hasta la próxima cláusula **where**.

- *if-then*-constructor

  Rule ::= **if** *g* **then**
  $Rule_1$
  [**else**
  $Rule_2$]
  endif

  El conjunto de actualización especificado por *Rule* en un estado dado s se define como el conjunto de actualización de $Rule_1$ o $Rule_2$, según cual sea el valor de g en el estado s. Sin la parte facultativa **else**, el conjunto de actualización definido por *Rule* es el conjunto de actualización de $Rule_1$ o el conjunto de actualización vacío. Algunas veces, **elseif** se utiliza como abreviatura para **else if**.

- do-in-parallel-constructor

Rule ::= [**do in-parallel**]
$Rule_1$
...
$Rule_n$
[**enddo**]

El conjunto de actualización definido por *Rule* en un estado *s* se define como la unión de los conjuntos de actualización de $Rule_1$ a $Rule_n$. En otras palabras, no tiene importancia el orden en el que se enuncian las reglas de transición que pertenecen al mismo bloque. Para abreviar, pueden omitirse las palabras clave **do in-parallel** y **enddo**, siempre que esto no provoque confusión. Por ello, un programa ASM aparece a menudo como una recolección de reglas más que como un conjunto monolítica en bloque.

- *do-forall*-constructor

Rule ::= **do forall** *v*: *g*(*v*)
$Rule_0(v)$
enddo

El efecto de *Rule* es que $Rule_0$ se dispara simultáneamente para todos los elementos *v* del conjunto de base de *M* para el cual la condición booleana *g*(*v*) es válida en el estado *s*, donde *v* es una variable libre en $Rule_0$. Más precisamente, $\Delta_s(Rule)$ es la unión de todos los conjuntos de actualización $\Delta_S(Rule_0(v))$ de manera tal que *g*(*v*) sea válida en el estado *s*. Hay que recordar que los conjuntos de actualización deben ser finitos, por lo que, *g*(*v*) debe ser válida solamente para un número finito de valores.

- *choose*-constructor

Rule ::= **choose** *v: g*(*v*)
$Rule_0(v)$
endchoose

El efecto de *Rule* es que $Rule_0$ se dispara para algún elemento *v* del conjunto de base de *M* para el cual es válida la condición *g*(*v*) en el estado *s*, siendo *v* una variable libre de $Rule_0$. Con mayor precisión, cabe decir que $\Delta_s(Rule)$ es algún conjunto de actualización $\Delta_s(Rule_0(v))$ tal que *g*(*v*) es válido en el estado *s*, o que es un conjunto de actualización vacío si no existe esa *v*.

- *extend*-constructor[3]

Rule ::= **extend** *D* **with** $v_1,...,v_n$
$Rule_0(v_1,...,v_n)$
endextend

El efecto de *Rule* cuando se dispara en el estado *s,* es que *n* elementos de reserva de *s* (véase 3.1.2.) son importados en el dominio dinámico *D* (mientras se eliminan de la reserva), de tal manera que $v_1,...,v_n$ se vincula sucesivamente a cada uno de los elementos importados y a continuación se dispara $Rule_0(v_1,...,v_n)$.

El constructor *extend* se puede utilizar para imitar las definiciones ASM basadas en el objeto, cuando los objetos se crean dinámicamente. Por lo tanto, por cada objeto que se deba crear, se asigna e inicializa un elemento de la reserva al dominio correspondiente.

- *let*-constructor

Rule ::= **let** *v* = *expression* **in**
$Rule_0(v)$
endlet

El efecto de *Rule* cuando se dispara en algún estado *s* es que *v* se vincula al valor de *expression*, y que $Rule_0$ se dispara con este valor.

[3] En sentido estricto, *extend* se puede definir de acuerdo con los términos del constructor *import* (que no aparece aquí); sin embargo, en este anexo no se utiliza el constructor *import*.

Ejemplo (regla de transición):

La siguiente regla de transición define el comportamiento de un agente *ag* cuando solicita un acceso compartido, es decir cuando *ag.mode = shared*. La regla es válida para el constructor condicional, para el constructor seleccionado y para una instrucción de actualización.

```
if ag.mode = shared ∧ ag.Waiting then
    choose t: t ∈ TOKEN ∧ t.Available
        t.owner := ag
endchoose
endif
```

El significado exacto de la regla viene dado por su conjunto de actualización en relación a un estado *s*, que es bien $\{\langle\langle owner, \langle s(t)\rangle\rangle, s(ag)\rangle\}$ para algún testigo $s(t)$ disponible en *s*, si todos los otros predicados enunciados en el constructor condicional son válidos en *s*, o bien, el conjunto de actualización vacío, en caso contrario.

3.1.7 Abreviaturas

Las reglas se pueden estructurar utilizando las *abbreviations* (abreviaturas), que constan de *rule macros* y *derived names* (nombres derivados), que pueden tener parámetros. Esto permite definiciones jerárquicas y la mejora por pasos sucesivos de las reglas complejas, lo que ayuda a comprender las definiciones del modelo ASM.

Los nombres derivados se introducen aplicando los métodos indicados en 3.1.1 y 3.1.3, es decir mediante declaración y definición o, alternativamente, en forma compacta combinando declaración y definición.

- *rule-macro*-definition

  Sea $Rule_0$ la expresión de una regla de transición con variables libres $v_1,...,v_n$ de los dominios $D_1,...,D_n$, $n \geq 0$. La forma general de la definición de los macros de la regla es:

  $RuleMacroDefinition ::= RuleMacroName(v_1{:}D_1,...,v_n{:}D_n) \equiv Rule_0(v_1,...,v_n)$

  Se ha convenido en escribir los nombres de los macros de regla en mayúsculas pequeñas, con una letra mayúscula principal (tal como SHAREDACCESS).

- *where*-part

  Por defecto, los *rule macros* y *derived names* tienen un ámbito global. Sin embargo su ámbito puede ser limitado también a una regla de transición especial, *Rule* utilizando la where-part.

  ```
  Rule ::=  Rule0
              where
                 ( RuleMacroDefinition | DerivedNameDefinition )+
            endwhere
  ```

- Constructor de los macros de la regla

  Los macros de regla se aplican en las reglas de transición de la manera siguiente:

  $Rule ::= RuleMacroName(t_1,...,t_n)$

  De manera formal, los macros de regla son abreviaturas sintácticas, es decir, cada vez que se produce un macro en una regla éste debe ser textualmente reemplazado por una definición del macro vinculado (reemplazando los parámetros formales por los parámetros reales).

Ejemplo (macros de regla):

La regla de transición del ejemplo previo se puede enunciar utilizando macros de regla y puede definir como un macro en sí mismo. En este caso, SHAREDACCESS es una definición macro con un ámbito global que puede ser utilizada en otros lugares de la definición del modelo ASM. GETTOKEN es una definición macro parametrizada de ámbito local restringido a la regla SHAREDACCESS, con un parámetro formal *a*. Cuando se aplica GETTOKEN en SHAREDACCESS, *a* es reemplazada por el parámetro concreto *ag*.

```
SHAREDACCESS ≡
        if ag.mode = shared ∧ ag.Waiting then
            GETTOKEN(ag)
    endif
    donde
            GETTOKEN(a:AGENT) ≡
                choose t: t ∈ TOKEN ∧ t.Available
                        t.owner :=a
                    endchoose
        endwhere
```

3.1.8 Programas de ASM

Un *ASM program P* viene dado por una *transition rule* encuadrada (o *rule* para simplificar) de la siguiente forma:

Rule

Como ya se ha mencionado, las definiciones de los macros de regla pueden tener un ámbito local o global. Para tener un ámbito global, las definiciones de macro pueden darse fuera del programa ASM y por lo tanto pueden aplicarse también en el programa ASM.

En el modelo ASM básico, existe solamente un programa ASM, que está asociado de manera estática con un agente definido implícitamente que ejecuta este programa. En la cláusula próxima se definen varios programas ASM y se asocian con diferentes agentes introducidos dinámicamente durante las ejecuciones de la máquina abstracta.

Ejemplo (programa de la ASM):

El programa *P* de la ASM del sistema *RMS* se define como sigue:

```
do in-parallel
    SharedAccess
    ExclusiveAccess
    ReleaseAccess
enddo
where
        SharedAccess ≡
                if ag.mode = shared ∧ ag.Waiting then
                    choose t: t ∈ Token ∧ t.Available
                        t.owner := ag
                    endchoose
                endif
        ExclusiveAccess ≡
                if ag.mode = exclusive ∧ ∀t ∈ Token: t.Available then
                    do forall t: t ∈ Token
                        t.owner := ag
                    enddo
                endif
        ReleaseAccess ≡
                if ag.Busy ∧ ag.Stop then
                    do in-parallel
                        ag.mode := undefined
                        do forall t: t ∈ Token ∧ t.owner = ag
                            t.owner := undefined
                        enddo
                    enddo
                endif
endwhere
```

El programa de la ASM se define mediante una única regla de transición como se muestra en el recuadro. La regla de transición utiliza el do-in-parallel-constructor y tres macros de regla, lo que da como resultado una definición jerárquica de la regla.

3.2 ASM distribuida

La modelización matemática de sistemas simultáneos y reactivos requiere la ampliación del modelo básico ASM. En esta cláusula, se explica el concepto de *distributed ASM* que generaliza el modelo ASM básico presentado en 3.1.

Una *distributed Abstract State Machine M* se define con un determinado *vocabulary V* por sus *states S*, sus *initial states* $S_0 \subseteq S$, sus agentes *A*, y sus *programs P*. En las cláusulas que siguen se explican estos elementos en la medida en que difiere del modelo ASM básico.

3.2.1 Vocabulario

El vocabulario *V* de una ASM *M* de agentes múltiples incluye nombres de dominio distinguidos

controlled domain *Agent*
static domain *Program*

que representan, respectivamente un conjunto dinámico A de agentes y un conjunto invariable P de programas de ASM. Además, V incluye un nombre de función distinguido

controlled *program*: *Agent* → *Program*

y una función especial 0-aria *Self* (véase 3.2.2).

3.2.2 Agentes y ejecuciones

Una ASM *M* distribuida puede tener cualquier número finito de agentes, pudiendo variar ese número dinámicamente según el estado de que se trate. El comportamiento de cada agente lo determina algún programa de *M*, definido por una regla de transición igual que en el modelo ASM básico. Los agentes operan simultáneamente ejecutando sus programas y actúan entre sí de manera asíncrona mediante posiciones de un estado compartidas globalmente, es decir dos o más agentes podrán leer y escribir en la misma posición. Los pasos de ejecución simultáneos del modelo ASM distribuida están restringidos a operaciones independientes, definiéndose el comportamiento admisible en términos de *partially ordered runs* (ejecuciones ordenadas parcialmente) (véase [3]). Por noción de simultaneidad autoriza una concurrencia verdadera en lugar de una concurrencia aproximada por un modelo entrelazado.

Para asignar un comportamiento a un agente de *M*, la función distinguida *program* (véase 3.2.1.) da para cada agente *a* de *M*, el programa de *P* que tendrá que ser ejecutado por *a*. Por tanto, la función *program* permite definir (o redefinir) dinámicamente el comportamiento de los agentes y de este modo es posible crear nuevos agentes en el momento de la ejecución. Un estado dado *s* de *M*, los agentes de *M* son todos los elementos *a* de s, con los que *a.program* identifica un comportamiento (tal como se define por algún programa de *P*) que debe ser asociado con *a*.

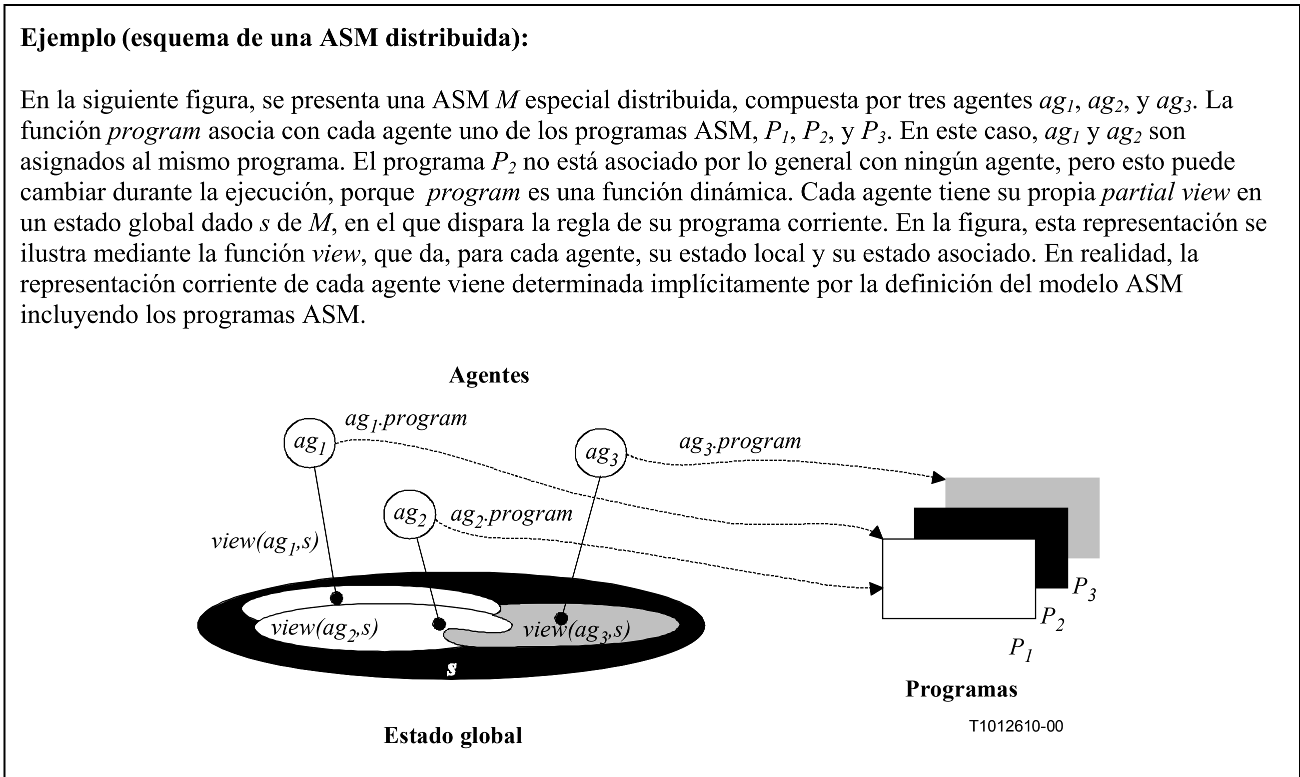
Una función 0-aria *Self* especial sirve como *self reference* (autorreferencia) que identifica al agente respectivo que llama a *Self*:

monitored *Self*: → *AGENT*

Self tiene una interpretación diferente para cada agente. Cuando se utiliza *Self* como un argumento de la función adicional, cada agente *a* puede tener su propia visión parcial de un estado global dado de *M* en el que dispara la regla en *a.program*.

Ejemplo (esquema de una ASM distribuida):

En la siguiente figura, se presenta una ASM *M* especial distribuida, compuesta por tres agentes ag_1, ag_2, y ag_3. La función *program* asocia con cada agente uno de los programas ASM, P_1, P_2, y P_3. En este caso, ag_1 y ag_2 son asignados al mismo programa. El programa P_2 no está asociado por lo general con ningún agente, pero esto puede cambiar durante la ejecución, porque *program* es una función dinámica. Cada agente tiene su propia *partial view* en un estado global dado *s* de *M*, en el que dispara la regla de su programa corriente. En la figura, esta representación se ilustra mediante la función *view*, que da, para cada agente, su estado local y su estado asociado. En realidad, la representación corriente de cada agente viene determinada implícitamente por la definición del modelo ASM incluyendo los programas ASM.



El modelo semántico de simultaneidad subyacente al modelo ASM distribuida, define el comportamiento en términos de ejecuciones ordenadas parcialmente. Una *partially ordered run* representa una cierta clase de ejecuciones de máquina (admisibles) restringiendo el no determinismo con respecto al orden en el que los agentes individuales podrán realizar sus pasos de cálculo, los llamados *moves*. Para evitar que los agentes interfieran entre sí, los movimientos de diferentes agentes sólo han de estar ordenados si tienen dependencias por motivos causales (como se detalla a continuación).

Ejecuciones ordenadas parcialmente

Por lo que se refiere a los movimientos de un agente determinado, dichos movimientos están ordenados linealmente mientras que los movimientos de agentes diferentes sólo necesitan ser ordenados cuando no son *independent* entre sí. Intuitivamente, los movimientos independientes modelan acciones simultáneas que no pueden ser comparadas en

relación con su orden de ejecución. El sentido exacto de independencia está implícito por la condición de coherencia en la definición formal de ejecuciones ordenadas parcialmente (adoptada de [3]).

Una ejecución ρ de una ASM *M* distribuida viene dada por un (Λ,*A*,σ) triple que satisface las cuatro condiciones siguientes:

1) Λ es un conjunto de movimientos ordenados parcialmente, en donde cada movimiento tiene muchos predecesores sólo de manera finita;
2) *A* es una función de Λ asociando agentes a movimientos de manera que los movimientos de un agente simple de *M* estén ordenados linealmente;
3) σ asigna un estado de *M* a cada segmento inicial *Y* de Λ, siendo σ(*Y*) el resultado de la realización de todos los movimientos en *Y*; si *Y* está vacío, $\sigma(Y) \in S_0$;
4) si *y* es el elemento máximo en un segmento inicial finito *Y* de Λ y $Z = Y - \{ y \}$, entonces *A*(*y*) es un agente en σ(*Z*) y σ(*Y*) se obtiene de σ(*Z*) disparando *A*(*y*) en σ(*Z*) (*condición de coherencia*).

Implicaciones

Las ejecuciones ordenadas parcialmente tienen ciertas propiedades características que pueden ser enunciadas en forma de *linearisations* (linealizaciones) de conjuntos ordenados parcialmente. Una linealización Λ de un conjunto ordenado parcialmente es un conjunto ordenado linealmente Λ' con los mismos elementos, de tal manera que si $y < z$ en Λ entonces $y < z$ en Λ'. En consecuencia, el modelo semántico de simultaneidad que implica la noción de ejecución ordenada parcialmente puede caracterizarse además de la manera siguiente [3]:

- Todas las linealizaciones del mismo segmento inicial finito de una ejecución de *M* tienen el mismo estado final.
- Una propiedad es válida en todo estado alcanzable de una ejecución ρ de *M* solamente si es válida en todo estado alcanzable de cada linealización de ρ.

3.2.3 Programas de ASM distribuidas

Una *ASM M* distribuida tiene un conjunto finito *P* de programas. Cada programa $p \in P$ viene dado por un *program name* y una *transition rule* (o *rule* para abreviar). El nombre del programa identifica *p* en *P* de manera exclusiva, y está representado por una función estática monaria[4]. Los programas son enunciados de la siguiente forma:

ASM-PROGRAM:

Rule

Los nombres de los programas se escriben por convenio en letras mayúsculas de tamaño reducido y unidos por guiones, con una letra mayúscula al principio (tal como en RESOURCE-MANAGEMENT-PROGRAM).

Las siguientes condiciones implícitas se aplican por defecto:

initially *PROGRAM* = $\{\text{PROGRAM}_1,\ldots,\text{PROGRAM}_n\}$

en donde $\text{PROGRAM}_1,\ldots,\text{PROGRAM}_n$ son los nombres de los programas que han sido definidos en el modelo ASM.

[4] En sentido estricto, los nombres de programa de *M* están representados por un conjunto distinguido de elementos del conjunto de base.

Ejemplo (programa ASM):

El programa ASM distribuido del sistema *RMS* define un programa simple de la forma siguiente:

RESOURCE-MANAGEMENT-PROGRAM:

```
do in-parallel
    SharedAccess
    ExclusiveAccess
    ReleaseAccess
enddo
where
    SharedAccess ≡
        if Self.mode = shared ∧ Self.Waiting then
            choose t: t ∈ TOKEN ∧ t.Available
                t.owner := Self
            endchoose
        endif
    ExclusiveAccess ≡
        if Self.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
            do forall t: t ∈ TOKEN
                t.owner := Self
            enddo
        endif
    ReleaseAccess ≡
        if Self.Busy ∧ Self.Stop then
            do in-parallel
                Self.mode := undefined
                do forall t: t ∈ TOKEN ∧ t.owner = Self
                    t.owner := undefined
                enddo
            enddo
        endif
endwhere
```

El programa de la ASM distribuida tiene el nombre de RESOURCE-MANAGEMENT-PROGRAM, y es definido como el anterior programa ASM con un único agente, con una diferencia: todos los casos de *ag* han sido reemplazados por la función *Self*. Esto permite asociar al programa con diferentes agentes, mientras se accede al estado local de los mismos.

3.3 El mundo externo

Siguiendo una *open system view* (visión de sistema abierto), las interacciones entre un sistema y el mundo externo, por ejemplo el entorno en el que el sistema está insertado, se modelan en término de diversos mecanismos de interfaz. Considerando la naturaleza reactiva de los sistemas distribuidos, es importante identificar claramente y enunciar de manera precisa:

- las condiciones previas del comportamiento esperado del mundo externo; y
- cómo afectan las condiciones y los eventos externos al comportamiento de un modelo ASM.

Esto se consigue clasificando los nombres ASM *dinámicos* en tres categorías básicas de nombres, lo que amplia la clasificación de nombres mostrada en la figura F1-3:

- *controlled names (nombres controlados)*

  Estos dominios, funciones o predicados sólo pueden ser modificados por agentes del modelo ASM en función de los programas ASM ejecutados. Los nombres controlados van precedidos en su punto de declaración por la palabra clave **controlled** y son visibles al entorno.

- *monitored names (nombres supervisados)*

  Estos dominios, funciones o predicados sólo pueden ser modificados por el entorno pero son visibles a los agentes de ASM. Por lo tanto, el dominio, la función o el predicado supervisados pueden cambiar sus valores de un estado a otro de manera imprevisible, a menos que se limite mediante *integrity constraints* (limitaciones de integridad) (véase más adelante) Los nombres supervisados van precedidos en su punto de declaración por la palabra clave **monitored**.

- *shared names (nombres compartidos)*

  Estos dominios, funciones o predicados son visibles y pueden ser alterados por el entorno, así como por los agentes de ASM. Por lo tanto, una *integrity constraint* en dominios, funciones o predicados compartidos indica que no se debe producir ninguna interferencia en relación con las posiciones recíprocamente actualizadas. Hace falta, en consecuencia que el propio entorno actúe como un agente de ASM (o un conjunto de agentes de ASM). Los nombres compartidos van precedidos en su punto de declaración por la palabra clave **shared**. Véase la figura F1-4.

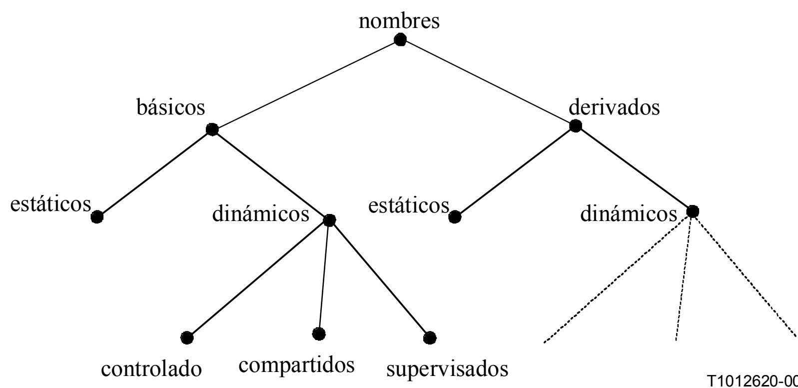


Figura F1-4/Z.100 – Clasificación ampliada de los nombres ASM

Ejemplo (mundo externo):

El vocabulario *V* del sistema *RMS* se amplía mediante una clasificación de las funciones y predicados dinámicos:

shared *mode*: $AGENT \rightarrow MODE$

controlled *owner*: $TOKEN \rightarrow AGENT$

monitored *Stop*: $AGENT \rightarrow BOOLEAN$

La función *mode*, que determina el modo de acceso corriente, es una función compartida. Puede ser afectada por las operaciones del 'conjunto' controlado externamente comuntándola a uno de los valores *exclusive* o *shared*. Además, se reinicia internamente cuando se libera el recurso (véase 3.2.3).

El predicado *Stop* representa un pedido de parada externo, tal como una interrupción y por lo tanto es supervisado.

Por lo general, la influencia del entorno en el sistema a través de los nombres compartidos y supervisados es completamente imprevisible. Sin embargo, se pueden formular condiciones previas sobre el comportamiento previsto del entorno estableciendo *integrity constraints*, que se requiere que sean válidas en *todos* los estados y ejecuciones de *M*. Se señala que las limitaciones de integridad expresan solamente condiciones previas del comportamiento del entorno, pero *no* se supone que tengan propiedades de sistema.

Las limitaciones de integridad se enuncian de la manera siguiente:

IntegrityConstraint ::= **constraint** *ClosedFormula*

Ejemplo (limitaciones de integridad):

La siguiente limitación de integridad indica que las peticiones de paradas sólo son generadas por los agentes ocupados:

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

3.4 Comportamiento en tiempo real

Introduciendo una noción de *real time* e imponiendo limitaciones adicionales a las ejecuciones, obtenemos una clase especializada de ASM llamada *distributed real-time ASM*, (ASM distribuida en tiempo real), con agentes que realizan acciones *instantáneas* en tiempo *continuo*. Esto significa, básicamente, que los agentes disparan sus reglas en el momento en que son habilitados.

Para incorporar un comportamiento en tiempo real en el modelo de ejecución de la ASM subyacente, se introduce una función 0-aria supervisada *currentTime* que toma valores reales. Intuitivamente, *currentTime* se refiere al tiempo físico. Siendo una limitación de la integridad en la naturaleza del tiempo físico, se presume que *currentTime* cambia sus valores de manera monótona aumentando en las ejecuciones ASM.

monitored *currentTime*: → *REAL*

Considérese un vocabulario dado *V* que contiene *REAL* (pero no *currentTime*) y sea V^+ la ampliación de *V* con el símbolo de la función *currentTime*. Atención solamente a los enunciados de V^+ en los que *currentTime* toma como valor un número real, se pueda definir una ejecución *R* del modelo de máquina resultante como el establecimiento de la correspondencia entre el intervalo $[0,\infty)$ y los enunciados del vocabulario V^+ que satisfacen los siguientes *discreteness requirement* (requisitos de discreción), en donde $\sigma(t)$ denota la reducción[5] de $R(t)$ a *V*:

1) para cada $t \geq 0$, *currentTime* toma como valor *t* en el estado $R(t)$;

2) para cada $\tau > 0$, hay una secuencia finita $0 = t_0 < t_1 <\ldots< t_n = \tau$ tal que si $t_i < \alpha < \beta < t_{i+1}$ entonces $\sigma(\alpha) = \sigma(\beta)$.

Utilizando la propiedad de discreción, se obtiene efectivamente alguna representación finita (*historia*) para cada (sub)ejecución finita haciendo abstracción de los estados que se considera que no son lo bastante significativos como para contribuir con alguna información pertinente a la descripción del comportamiento. En concreto, se pueden simplemente ignorar todos los estados que sean idénticos a su estado precedente excepto que *currentTime* ha aumentado. De la definición de ejecución indicada más arriba se sigue que quedan muchos estados sólo de manera finita.

3.5 Ejemplo: el sistema *RMS*

En esta cláusula se ensamblan las piezas de la definición del modelo ASM del sistema *RMS* en su versión final. Para una mejor referencia, se repite además la descripción oficiosa.

Descripción oficiosa

Para representar el modelo ASM, se define un resource management system (sistema de gestión de recursos) *RMS* sencillo compuesto por *un grupo de agentes*, *n > 1* que compiten por un *recurso*, por ejemplo, algún dispositivo o servicio. De manera oficiosa, este sistema está caracterizado por lo siguiente:

- Existe un conjunto de *testigos, m < n*, utilizados para hacer posible un acceso *exclusive (exclusivo)* o *non-exclusive* (*shared*) (*no exclusivo* (*compartido*)) al recurso.
- Según que el modo de acceso deseado sea exclusivo o compartido, un agente debe poseer todos los testigos o un testigo respectivamente antes de poder acceder al recurso.
- Un agente está *idle*, cuando no está compitiendo por un recurso, está *waiting*, cuando está tratando de obtener acceso al recurso, o está *busy* cuando posee el derecho de acceder al recurso.
- Una vez que el agente está en situación de esfera, continúa en la misma hasta que obtiene el acceso al recurso.
- Un agente ocupado libera el recurso cuando no lo necesita, lo que se indica mediante una *stop condition* (condición de parada) fijada externamente para ese agente. Al liberarse el recurso, son devueltos todos los testigos que posee el agente.
- Las condiciones de parada sólo se indican cuando un agente está ocupado.
- Inicialmente, todos los agentes están en reposo y todos los testigos están disponibles.

Vocabulario

static domain *TOKEN*
shared *mode*: *AGENT* → *MODE*
controlled *owner*: *TOKEN* → *AGENT*
monitored *Stop*: *AGENT* → *BOOLEAN*

---

5 Es decir, para un valor *t* dado, se obtiene $\sigma(t)$ de $R(t)$ ignorando la interpretación del nombre de la función *currentTime*.

Nombres derivados

$MODE =_{def} \{exclusive, shared\}$

Idle(*a*:*AGENT*): *BOOLEAN*	$=_{def}$	$a.mode = undefined \wedge \forall t \in TOKEN: t.owner \neq a$
Waiting(*a*:*AGENT*): *BOOLEAN*	$=_{def}$	$a.mode \neq undefined \wedge \forall t \in TOKEN: t.owner \neq a$
Busy(*a*:*AGENT*): *BOOLEAN*	$=_{def}$	$a.mode \neq undefined \wedge \exists t \in TOKEN: t.owner = a$
Available(*t*:*TOKEN*): *BOOLEAN*	$=_{def}$	$t.owner = undefined$

Limitaciones de seguridad

constraint $\forall a \in AGENT: (a.Stop \Rightarrow a.Busy)$

Limitaciones iniciales

initially $|AGENT| > 1$

initially $|TOKEN| < |AGENT|$

initially $\forall a \in AGENT$: *a.program* = RESOURCE-MANAGEMENT-PROGRAM

initially $\forall a \in AGENT: a.Idle \wedge \forall t \in TOKEN: t.Available$

Programas ASM

RESOURCE-MANAGEMENT-PROGRAM:

```
do in-parallel
        SHAREDACCESS
        EXCLUSIVEACCESS
        RELEASEACCESS
enddo
where
        SHAREDACCESS ≡
                if Self.mode = shared ∧ Self.Waiting then
                    choose t: t ∈ TOKEN ∧ t.Available
                        t.owner := Self
                    endchoose
                endif
        EXCLUSIVEACCESS ≡
                if Self.mode = exclusive ∧ ∀t ∈ TOKEN: t.Available then
                    do forall  t: t ∈ TOKEN
                        t.owner := Self
                    enddo
                endif
        RELEASEACCESS ≡
                if Self.Stop then
                    Self.mode := undefined
                    do forall t: t ∈ TOKEN ∧ t.owner = Self
                        t.owner := undefined
                    enddo
                endif

endwhere
```

3.6 Nombres definidos previamente

Para definir un modelo ASM, en particular el modelo ASM que incorpora la semántica de SDL, se definen previamente algunos nombres así como la interpretación que se les quiere dar. Estos nombres figuran agrupados y listados en esta cláusula (en donde *D* se refiere a la categoría sintáctica de los dominios). Para los operadores prefijo, infijo y sufijo, se utiliza un subrayado ("_") que indica la posición de sus argumentos. Además, se indica la precedencia de los operadores por prec(n), siendo n es un número. Los números más altos significan un vínculo más estrecho. Los operadores monádicos tienen una vinculación más fuerte que los binarios. Los operadores binarios son asociativos hacia la izquierda.

Dominios específicos de ASM	
static domain *X*	Conjunto de base ASM (dominio meta)
static domain *BOOLEAN*	Valores booleanos
static domain *NAT*	Valores enteros
static domain *REAL*	Valores reales
shared domain *AGENT*	Agentes ASM
static domain *PROGRAM*	Programas ASM
static domain *TOKEN*	Testigos de sintaxis (cadenas de caracteres)
_ *	Constructor del dominio: secuencias finitas de
_ +	Constructor del dominio: secuencias finitas de
_ **-set**	Constructor del dominio: conjuntos finitos de
_ × _ prec(7)	Constructor del dominio de tupla
_ ∪ _ prec(6)	Constructor del dominio de unión

Funciones específicas de ASM	
static *undefined*: → *X*	Indicador de valores indefinidos
monitored *Self*: → *AGENT*	Autorreferencia para los agentes ASM
controlled *program*: *AGENT* → *PROGRAM*	Programa de un agente ASM
monitored *currentTime*: → *REAL*	Tiempo del sistema corriente

Funciones booleanas y predicadas	
static *True*: → *BOOLEAN*	Literal predefinido
static *False*: → *BOOLEAN*	Literal predefinido
_ = _ prec(4)	Igualdad
_ ≠ _ prec(4)	Desigualdad
_ ∧ _ prec(3)	y lógica
_ ∨ _ prec(2)	o lógica
_ ⇒ _ prec(1)	Implicación
_ ⇔ _ prec(1)	Equivalencia lógica
¬ _	Negación
$\exists x \in D: P(x)$ prec(0)	Cuantificación existencial (por lo menos un elemento)
$\exists! x \in D: P(x)$ prec(0)	Cuantificación existencial única exactamente un elemento
$\forall x \in D: P(x)$ prec(0)	Cuantificación universal

Términos	
X	Aplicación de la función 0-aria (función constante)
$f(t_1,..., t_n)$	Aplicación de la función con n expresiones de argumentos
if *Formula* **then** *Term* **else** *Term* **endif**	Expresión condicional; nuevamente se utiliza **elseif** en lugar de **else if**
s-_(_)	Función de selección tupla (véase tuplas más abajo)
mk-_(...)	Construcción de tupla (véase tuplas más abajo)
inv-_(...)	La inversa de una función o mapa, **inv-***Fun*(*x*) $=_{def}$ *take*({ *a* ∈ *D*: *Fun*(*a*) =*x* })

Funciones y relaciones en enteros	
_ > _, _ ≥ _, _ <_ , _ ≤ _ prec(4)	Operadores de comparación
_ + _ prec(6)	Adición
_ - _ prec(6)	Sustracción
_ *_ prec(7)	Multiplicación
_ / _ prec(7)	División
0, *1*, ...	Literales enteros

Funciones en secuencias	
static *empty*: $\rightarrow D^*$	Secuencia vacía
static *head*: $D^* \rightarrow D$	Inicio de la secuencia (*no definido* cuando está vacía)
static *tail*: $D^* \rightarrow D^*$	Fin de la secuencia (*no definido* cuando está vacía)
static *last*: $D^* \rightarrow D$	Último elemento de una secuencia (*no definido* cuando está vacía)
static *length*: $D^* \rightarrow$ *NAT*	Longitud de una secuencia
static < >: $D^n \rightarrow D^*$	Constructor de una secuencia; los argumentos están listados entre paréntesis, separados por comas
_ ⁀ _ prec(6)	Concatenación de secuencias
toSet: $D^* \rightarrow$ *D*-**set**	Conversión de los elementos de una secuencia en un conjunto
_ [_]	Acceso a un elemento de una lista; el índice entre paréntesis tendrá que ser del tipo *NAT*
_ **in** _ prec(4)	Elemento de?
< <*result*> | <*var*> **in** <*seq*> : <*cond*> >	Comprensión de la secuencia; actúa como un filtro en <seq>, es decir preservando el orden
< <*var*> **in** <*seq*> : <*cond*> > $=_{def}$ < <*var*> | <*var*> **in** <*seq*> : <*cond*> >	Comprensión de una secuencia abreviada
< <*result*> | <*var*> **in** <*seq*> > $=_{def}$ < <*result*> | <*var*> **in** <*seq*> : True >	Comprensión de una secuencia abreviada

Funciones en conjuntos	
_ $\cup$ _ prec(6)	Unión de conjuntos
_ $\cap$ _ prec(7)	Intersección
_ \ _ prec(6)	Sustracción de conjuntos
_ $\in$ _ prec(4)	Elemento de?
_ $\notin$ _ prec(4)	No elemento de?
_ $\subseteq$ _ prec(4)	Subconjunto de?
_ $\subset$ _ prec(4)	Subconjunto apropiado de?
| _ |	Tamaño de un conjunto
U _	Unión grande: unión de todos los conjuntos incluidos en el conjunto de argumentos
$\varnothing$	Conjunto vacío
static { }: $D^n \rightarrow$ *D*-**set**	Constructor de conjunto; lista separada por coma de argumentos entre paréntesis
take: *D*-**set** $\rightarrow D$	Selección de un elemento cualquiera del conjunto, o *indefinido* para un conjunto vacío
_ .. _ prec(5)	Gama de enteros del valor primero al segundo. Conjunto vacío cuando la segunda expresión es menor que la primera
{ <*result*> | <*var*> $\in$ <*set*> : <*cond*> }	Comprensión del conjunto, actúa como un filtro en <set>
{ <*var*> $\in$ <*set*> : <*cond*> } $=_{def}$ { <*var*> | <*var*> $\in$ <*set*>: <*cond*> }	Comprensión del conjunto abreviado
{ <*result*> | <*var*> $\in$ <*set*> } $=_{def}$ { <*result*> | <*var*> $\in$ <*set*>: True }	Comprensión del conjunto abreviado

Patrones y expresiones de casos

Los patrones suministran medios para acceder fácilmente a la estructura de los valores. Se proporcionan los patrones siguientes:

- Variables: una variable concuerda con cualquier valor. Sin embargo, si la variable ya está vinculada sólo concuerda consigo misma.
- Variables anónimas: las variables anónimas se indican mediante "*". Son notaciones abreviadas para introducir una variable no utilizada.
- Constructor: un constructor viene dado por su nombre y los argumentos, que a su vez son patrones. Se concuerda con cualquier valor construido utilizando el constructor y con los argumentos que concuerdan con el patrón correspondiente.
- Patrón nombrado: la notación Variable = Patrón introduce un nombre para (el valor que concuerda con) el patrón.

Los patrones se utilizan para describir las funciones en el árbol de sintaxis. Los nombres de no terminal de la gramática son utilizados como funciones de construcción.

Se utiliza una expresión de caso para determinar un valor según la concordancia con un patrón.

$CaseExpression$ ::= **case** $Term$ **of**
| $Pattern_1$: $Term_1$
| $Pattern_2$: $Term_2$
...
[**otherwise** $Term_0$]
endcase

Si el valor de *Term* concuerda por lo menos con un $Pattern_i$, el resultado de la expresión de caso viene dado por $Term_i$. Si no concierne ningún patrón, el resultado es $Term_0$ (si está presente). De otro modo, el resultado es *undefined.*

Dominios de unión

Los dominios de unión contienen simplemente los valores de los dominios que lo constituyen.

$$D =_{\text{def}} D_1 \cup D_2$$

Tuplas

Para cada dominio de tuplas declarado se definen varias funciones implícitas de constructor y selector. La definición

$$D =_{\text{def}} D_1 \times D_2^* \times D_3\text{-}\mathbf{set} \times D_1 \times (D_1 \cup D_2)$$

define también las funciones siguientes:

$\mathbf{mk}\text{-}D$: $D_1 \times D_2^* \times D_3\text{-}\mathbf{set} \times D_1 \times (D_1 \cup D_2) \to D$
$\mathbf{s}\text{-}D_1$: $D \to D_1$
$\mathbf{s}\text{-}D_2\text{-}\mathbf{seq}$: $D \to D_2^*$
$\mathbf{s}\text{-}D_3\text{-}\mathbf{set}$: $D \to D_3\text{-}\mathbf{set}$
$\mathbf{s2}\text{-}D_1$: $D \to D_1$
$\mathbf{s\text{-}implicit}$: $D \to (D_1 \cup D_2)$

Cuando la tupla incluye el mismo dominio más de una vez, se definen funciones selector similares a $\mathbf{s2}\text{-}D_1$. Para unión, se define la función selector especial **s-implicit**.

Reglas de sintaxis abstracta

Las reglas de sintaxis abstracta de la definición del lenguaje, se convierten directamente a la notación ASM utilizando algunos convenios que se explicarán por medio de ejemplos. Básicamente, una regla de sintaxis abstracta se puede interpretar como la declaración de uno o más dominios (tupla) y la definición de funciones para construir y seleccionar valores de los dominios componentes. Sin embargo, los nodos de sintaxis tienen una identidad, lo que no tienen las tuplas ordinarias. Existen reglas de sintaxis que introducen constructores denominados, así como uniones denominadas y no denominadas. Los constructores que introducen reglas están compuestos de símbolos terminales y no terminales y tienen la forma de:

$Symbol$ :: $Symbol_1$ $Symbol_2^+$ $Symbol_3$**-set** [$Symbol_4$]

que se convierte en

$Symbol\text{-}aux =_{\text{def}} Symbol_1 \times Symbol_2^* \times Symbol_3\text{-}\mathbf{set} \times Symbol_4$
controlled domain *Symbol*
controlled *contents-Symbol*: $Symbol \to Symbol\text{-}aux$
$\mathbf{s}\text{-}Symbol_1(x\text{: }Symbol)\text{: }Symbol_1 =_{\text{def}} \mathbf{s}\text{-}Symbol_1(x.contents\text{-}Symbol)$
$\mathbf{s}\text{-}Symbol_2\text{-}\mathbf{seq}(x\text{: }Symbol)\text{: }Symbol_2^* =_{\text{def}} \mathbf{s}\text{-}Symbol_2\text{-}\mathbf{seq}(x.contents\text{-}Symbol)$
$\mathbf{s}\text{-}Symbol_3\text{-}\mathbf{set}(x\text{: }Symbol)\text{: }Symbol_3\text{-}\mathbf{set} =_{\text{def}} \mathbf{s}\text{-}Symbol_3\text{-}\mathbf{set}(x.contents\text{-}Symbol)$
$\mathbf{s}\text{-}Symbol_4(x\text{: }Symbol)\text{: }Symbol_4 =_{\text{def}} \mathbf{s}\text{-}Symbol_4(x.contents\text{-}Symbol)$

Además, existe una abreviatura **mk-***Symbol*. Esta abreviatura equivale a crear un nuevo objeto del dominio *Symbol* utilizando la primitiva **extend** y a fijar el valor *contents-Symbol* del objeto recién producido en el resultado de **mk-***Symbol-aux*. Se señala que este tipo de abreviatura no es una función sino que en realidad es un elemento de regla. Por lo tanto, se debe usar solamente dentro de las reglas. El hecho de que el $Symbol_4$ facultativo no esté presente, se expresa en el modelo ASM dejando el valor correspondiente *undefined*.

Una secuencia vacía de símbolos (constructor sin partes) se expresa por ( ).

La igualdad de los valores de sintaxis es siempre una igualdad estructural, es decir, se compara el contenido de los símbolos en lugar de comparar los propios símbolos.

Las reglas de sintaxis que introducen uniones denominadas, es decir, los sinónimos, tienen la forma:

$Symbol = Symbol_1 \mid Symbol_2 \mid ... \mid Symbol_n$ $(n \geq 1)$

lo que se convierte en:

$Symbol =_{\text{def}} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$

Se señala que, puesto que *Symbol* es un dominio *union*, la expresión genera una definición de dominio pero no funciones **mk-** or **s-**.

A veces, no es necesario referirse a sinónimos. En tales casos, las uniones no denominadas pueden ser introducidas por:

$Symbol :: Symbol_1 \{ Symbol_{21} \mid ... \mid Symbol_{2n} \}$

en lugar de introducir los sinónimos:

$Symbol :: Symbol_1\ Symbol_2$
$Symbol_2 = Symbol_{21} \mid ... \mid Symbol_{2n}$

Para cada palabra clave **KEYWORD** de SDL, existe un dominio de palabra clave asociado *Keyword* con solo un valor:

static domain *Keyword*

Es preciso que todos los dominios de palabras clave sean mutuamente inconexos.

En la gramática abstracta, existe un dominio derivado llamado *DefinitionAS1*, que está compuesto por todos los dominios de los símbolos de sintaxis abstracta, de la forma siguiente

$DefinitionAS1 =_{\text{def}} Symbol_1 \cup Symbol_2 \cup ... \cup Symbol_n$

donde $Symbol_1, Symbol_2, ..., Symbol_n$ es la lista de *all* terminales y no terminales de la gramática abstracta.

Existe un dominio *DefinitionAS0* similar para la gramática concreta (AS0).

Para navegar descendiendo por un determinado árbol de sintaxis abstracta, se pueden utilizar las funciones **s-** Para navegar ascendiendo se definen dos funciones progenitoras.

controlled *parentAS1*: *DefinitionAS1* → *DefinitionAS1*
controlled *parentAS0*: *DefinitionAS0* → *DefinitionAS0*

Además, se definen dos funciones para encontrar el progenitor de un tipo particular.

parentAS0ofKind(*from*: *DefinitionAS0*, *x*: *DefinitionAS0*-**set**): *DefinitionAS0* $=_{\text{def}}$
  if *from* = *undefined* **then** *undefined*
  elseif *from* ∈ *x* **then** *from*
  else *parentAS0ofKind*(*from*.*parentAS0*, *x*)
  endif
parentAS1ofKind(*from*: *DefinitionAS1*, *x*: *DefinitionAS1*-**set**): *DefinitionAS1* $=_{\text{def}}$
  if *from* = *undefined* **then** *undefined*
  elseif *from* ∈ *x* **then** *from*
  else *parentAS1ofKind*(*from*.*parentAS1*, *x*)
  endif

Las funciones *isAncestorAS1* y *isAncestorAS0* determinan si el primer nodo es el antecesor del segundo:

isAncestorAS1(*n*: *DefinitionAS1* ,*n'*: *DefinitionAS1*): *BOOLEAN* $=_{\text{def}}$
  $n = n'.parentAS1 \lor isAncestorAS1(n, n'.parentAS1)$

isAncestorAS0(*n*: *DefinitionAS0* ,*n'*: *DefinitionAS0*): *BOOLEAN* $=_{\text{def}}$
  $n = n'.parentAS0 \lor isAncestorAS0(n, n'.parentAS0)$

El nodo superior del árbol de sintaxis abstracta o concreta corriente se expresa mediante las siguientes funciones 0-arias:

controlled *rootNodeAS1*: $\rightarrow$ *DefinitionAS1*
controlled *rootNodeAS0*: $\rightarrow$ *DefinitionAS0*

El árbol de sintaxis abstracta puede ser modificado utilizando las funciones derivadas siguientes:

replaceInSyntaxTree: *DefinitionAS0* $\times$ *DefinitionAS0* $\times$ *DefinitionAS0* $\rightarrow$ *DefinitionAS0*

El primer parámetro de la función es el subárbol antiguo, el segundo es el subárbol nuevo y el tercero es el árbol antiguo. El resultado de la función es el árbol nuevo, en el que todos los subárboles antiguos han sido reemplazados por el subárbol nuevo.

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	Redes de cable y transmisión de programas radiofónicos y televisivos, 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**

Ginebra, 2002